# DianFreya Math Physics Simulator with C++

DS Glanzsche[1]

Freya

Hamzst

Berlin-Sentinel Academy of Science, AliceGard

$1^{st}$ Edition

[1]A thank you or further information

# Contents

**Preface** **12**

**1 About DianFreya Math Physics Simulator (DFSimulatorC++)** **17**

**2 Using C++ in GFreya OS** **21**
I Introduction . . . 21
II How to Compile C++ Source Code into Executable Binary . . . 21
III Know-How in C++ . . . 78

**3 Mathematics and Physics in Computer Graphics** **81**
I Mathematics . . . 81
i Geometry . . . 81
ii Linear Algebra . . . 82
II Physics . . . 89

**4 Computer Graphics: OpenGL, SFML, GLFW, GLEW, And All That** **91**
I OpenGL . . . 91
II GLAD . . . 95
III GLEW . . . 96
IV GLFW . . . 97
V GLM . . . 97
VI GLSL . . . 99
VII SDL . . . 99
VIII SFML . . . 99
IX SOIL . . . 100
X Gnuplot . . . 101
XI CMake . . . 101

**5 Box2D, Bullet3, and ReactPhysics3D** **105**
I Box2D . . . 105
i Install Box2D Library and Include Files / Headers . . . 105
ii Build Box2D Testbed . . . 108
iii Know-How in Box2D . . . 110
II Bullet . . . 112
i Install Bullet Library and Include Files / Headers . . . 112
ii Know-How in Bullet . . . 113
iii Create an Example with Bullet . . . 114
III ReactPhysics3D . . . 114

i Install ReactPhysics3D Library and Include Files / Headers . . . . . 114
ii Know-How in ReactPhysics3D . . . . . 115

**6 DFSimulatorC++ 0: C+++ = C++ + Eigen + Gnuplot + SymbolicC++ and All That 117**
I Gnuplot . . . . . 119
i Plot a Slope with Gnuplot from C++ . . . . . 119
ii Plot a 2D Function with Gnuplot from C++ . . . . . 124
iii Plot a 3D Function with Gnuplot from C++ . . . . . 126
iv Plot a 3D Curve / Line from User Defined Points with Gnuplot from C++ . 128
v Plot Data From Textfile in 2D with Gnuplot . . . . . 131
vi Plot Data and Fitting Curve From Textfile in 2D with "gnuplot-iostream.h" . 134
II SymbolicC++ Library . . . . . 137
i Compile and Install SymbolicC++ Library . . . . . 137
ii Symbolic Derivation and Integration with SymbolicC++ Library . . . . . 139
III Eigen Library . . . . . 141
i Compile and Install Eigen3 Library . . . . . 144
IV GiNaC Library . . . . . 145
i Compile and Install GiNaC Library . . . . . 145
V Armadillo Library . . . . . 146
i Compile and Install Armadillo Library . . . . . 146
VI GSL Library . . . . . 148
i Compile and Install GSL Library . . . . . 148
VII Tomsolver Library (Nonlinear System Solver) . . . . . 149
i Compile and Install Tomsolver Library . . . . . 149
VIII Assimp (Open Asset Import Library) . . . . . 153
i Compile and Install Assimp . . . . . 154

**7 DFSimulatorC++ I: Motion in Two Dimensions 155**
I Position, Displacement, Velocity, and Acceleration . . . . . 155
II Projectile Motion . . . . . 157
III Simulation for Projectile Motion with Box2D . . . . . 157
IV Simulation for Projectile Dropped from Above with Box2D . . . . . 162
V Uniform Circular Motion . . . . . 178
VI Simulation for Uniform Circular Motion with Box2D . . . . . 180

**8 DFSimulatorC++ II: Force and Motion 189**
I Newtonian Mechanics . . . . . 189
II Simulation for Newton's First Law with Box2D . . . . . 190
III Simulation for Newton's First Law and 2 Dimensional Forces with Box2D . . . . . 200
IV Some Particular Forces . . . . . 208
V Simulation for Applying Newton's Law with Box2D . . . . . 209
VI Friction, Drag Force, and Centripetal Force . . . . . 218
VII Simulation for Static Frictional Forces with Box2D . . . . . 219
VIII Simulation for Kinetic Friction with Box2D . . . . . 226
IX Drag Force and Terminal Speed . . . . . 233
X Simulation for Downhill Skiing with Box2D . . . . . 233
XI Uniform Circular Motion with Force . . . . . 242
XII Simulation for Airplane Uniform Circular Motion with Box2D . . . . . 243

**9 DFSimulatorC++ III: Kinetic Energy and Work 251**
I Kinetic Energy . . . 251
II Simulation for Kinetic Energy, Train Crash with Box2D . . . 251
III Work and Kinetic Energy . . . 258
IV Simulation for Work By Two Constant Forces with Box2D . . . 259
V Work Done By The Gravitational Force . . . 265
VI Simulation for Work in Pulling and Pushing a Sack up a Frictionless Slope . . . 266
VII Work Done By A Spring Force . . . 275
VIII Simulation for Spring Work with Box2D . . . 276
IX Work Done By A General Variable Force . . . 283
X Simulation for Work Done by a General Variable Force with Box2D . . . 283
XI Power . . . 290
XII Simulation for Power: Moving Elevator with Box2D . . . 293
XIII Simulation for Power: Pendulum Crate with Box2D . . . 301
XIV Simulation for Power: Vertical Facing Upward Spring with Box2D . . . 307
XV Simulation for Power: Boxes on Conveyor Belt with Box2D . . . 316

**10 DFSimulatorC++ IV: Potential Energy and Conservation of Energy 327**
I Potential Energy . . . 327
II Simulation for Potential Energy: Slide Along the Frictionless Loop-the-Loop with Box2D . . . 331
III Conservation of Mechanical Energy . . . 343
IV Simulation for Conservation of Mechanical Energy: Marble Fired Upward Using a Spring with Box2D . . . 344
V Simulation for Conservation of Mechanical Energy: Runaway Truck with Failed Brakes with Box2D . . . 352
VI Simulation for Conservation of Mechanical Energy: Spring Attached to a Breadbox on Frictionless Incline with Box2D . . . 358
VII Simulation for Conservation of Mechanical Energy: Marble Spring Gun with Box2D 368
VIII Reading a Potential Energy Curve . . . 379
IX Work Done On A System By An External Force . . . 384
X Conservation of Energy . . . 387
XI Simulation for Conservation of Energy: a Block Slides to a Higher Level with Box2D 396
XII Simulation for Conservation of Energy: Elevator Snaps with Box2D . . . 401
XIII Simulation for Conservation of Energy: Particle Slides till Stop with Box2D . . . 410

**11 DFSimulatorC++ V: Center of Mass and Linear Momentum 417**
I Center of Mass . . . 417
II Simulation for Center of Mass for Three Particles with Box2D . . . 421
III Newton's Second Law for a System of Particles . . . 426
IV Simulation of A Shell that Explodes into Two Fragments with Box2D . . . 431
V Linear Momentum . . . 440
VI Collision and Impulse . . . 443
VII Simulation for Race Car Wall Collision with Box2D . . . 451
VIII Conservation of Linear Momentum . . . 461
IX Simulation for Two-dimensional Airplane Explosion with Box2D . . . 465
X Momentum and Kinetic Energy in Collisions . . . 474
XI Simulation for Bullet Shot Upward with Box2D . . . 479

XII Elastic Collisions in One Dimension . . . 484
XIII Simulation for Elastic Collision of a Steel Ball Pendulum with Box2D . . . 491
XIV Simulation for Elastic Collision of Baseball and Basketball with Box2D . . . 497
XV Collisions in Two Dimensions . . . 502
XVI System with Varying Mass: A Rocket . . . 503
XVII Simulation for with Box2D . . . 507

**12 DFSimulatorC++ VI: Rotation 509**
I Simulation for Momentum with Box2D . . . 509

**13 DFSimulatorC++ VII: Rolling, Torque, and Angular Momentum 511**
I Simulation for Momentum with Box2D . . . 511

**14 DFSimulatorC++ VIII: Equilibrium and Elasticity 513**
I Simulation for Momentum with Box2D . . . 513

**15 DFSimulatorC++ IX: Gravity 515**
I Mathematical Physics Formula for Gravity . . . 515
II Simulation for Gravity with Bullet3, GLEW, GLFW and OpenGL . . . 517
III Simulation for Gravity with Box2D . . . 524
i Build DianFreya Modified Box2D Testbed . . . 524
IV Simulation for Gravity with ReactPhysics3D . . . 531

**16 DFSimulatorC++ X: Fluids 533**
I Simulation for Momentum with Box2D . . . 533

**17 DFSimulatorC++ XI: Oscillations 535**
I Simple Harmonic Motion . . . 535
i The Force Law for Simple Harmonic Motion . . . 536
II Energy in Simple Harmonic Motion . . . 537
III An Angular Simple Harmonic Oscillator . . . 538
IV Pendulums, Circular Motion . . . 538
V Nonlinear Pendulum . . . 542
VI (optional?)The Equation of Motion for Simple Pendulum . . . 543
VII Simulation of Simple Pendulum with Box2D . . . 543
i Build DianFreya Modified Box2D Testbed . . . 543
VIII Horizontal Spring-Mass System . . . 550
IX Simulation of Horizontal Spring-Mass System with Box2D . . . 551
X Vertical Spring-Mass System . . . 559
XI Simulation of Vertical Spring-Mass System with Box2D . . . 560
XII Oscillation of a Spring-Mass System . . . 568
XIII A Two-Massses Oscillator . . . 573
XIV Simulation of Two Masses Spring System with Box2D . . . 575
XV Friction and Oscillations . . . 584

**18 DFSimulatorC++ XII: Waves 587**
I Simulation for Momentum with Box2D . . . 587

**19 DFSimulatorC++ XIII: Temperature, Heat, and the First Law of Thermodynamics 589**
I Simulation for Momentum with Box2D . . . 589

**20 DFSimulatorC++ XIV: The Kinetic Theory of Gases** **591**
I Simulation for Momentum with Box2D . . . 591

**21 DFSimulatorC++ XV: Entropy and the Second Law of Thermodynamics** **593**
I Simulation for Momentum with Box2D . . . 593

**22 DFSimulatorC++ XVI: Probability and Data Analysis** **595**
I Statistics and Data Analysis . . . 595
II Variability . . . 595

**23 DFSimulatorC++ XVII: Numerical Linear Algebra** **597**
I Matrix . . . 597
II Matrix-Vector Multiplication . . . 604
i C++ Computation: A Matrix Times a Vector . . . 605
ii C++ Computation: A Matrix Times a Vector with Data from Textfile . . . 607
iii C++ Computation: Matrix Sum and Subtraction with Data from Textfiles . . 613
iv C++ Computation: Matrices Multiplication with Data from Textfile . . . 616
III Determinants of a Matrix . . . 620
i C++ Computation: Find the Determinant of a Square Matrix . . . 623
ii C++ Computation: Using the Adjoint to Find Numerical Inverse of a Matrix 627
iii C++ Computation: Symbolic Inverse of a Matrix . . . 635
iv C++ Computation: Solving Linear Systems with Cramer's Rule . . . 637
v C++ Computation: Solving Linear Systems with Gauss-Jordan Elimination . 643
IV Euclidean Vector Spaces . . . 648
i C++ Plot: Vector Addition in $\mathbb{R}^2$ . . . 665
ii C++ Plot: Vector Addition in $\mathbb{R}^3$ . . . 668
iii C++ Computation: Linear Combination . . . 671
iv C++ Computation: Norm, Angle, Dot Product and Distance Between Vectors in $\mathbb{R}^n$ . . . 672
v C++ Plot: Line and Its Normal in $\mathbb{R}^2$ . . . 675
vi C++ Plot: Plane and Its Normal in $\mathbb{R}^3$ . . . 679
vii C++ Plot: Distance Between a Point and the Plane in $\mathbb{R}^3$ . . . 683
viii C++ Plot: Direction Cosines in $\mathbb{R}^3$ . . . 687
ix C++ Plot: Parametric Equation of Plane in $\mathbb{R}^3$ . . . 692
x C++ Computation: Numerical Orthogonal Projection . . . 695
xi C++ Computation: Symbolic Orthogonal Projection . . . 700
xii C++ Plot and Computation: Cross Product in 3D . . . 701
V General Vector Spaces . . . 709
i C++ Computation: Wronskian for Symbolic Matrix . . . 782
ii C++ Plot and Computation: 3D Plot of Standard Basis Reflected toward a Line into New Bases . . . 785
iii C++ Computation: Particular Solution of a Nonhomogeneous Linear System $A\boldsymbol{x} = \boldsymbol{b}$ and General Solution of a Homogeneous Linear System $A\boldsymbol{x} = \mathbf{0}$ . . . 795
iv C++ Computation: Vector Form of the General Solution of a Homogeneous Linear System $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{x} = \mathbf{0}$ . . . 803
v C++ Computation: Compute Basis for a Column Space by Row Reduction . 810
vi C++ Computation: Compute Basis for the Row and Column Space by Row Reduction . . . 818

vii C++ Computation: Compute Basis for a Row Space by Row Reduction and Transposing Matrix . . . 828
viii C++ Computation: Row Space and Column Space Application in Material Science . . . 837
ix C++ Computation: Rank and Nullity of a Matrix . . . 844
x C++ Computation: Gauss-Jordan Elimination (Manual) for Overdetermined Symbolic Matrix . . . 855
xi C++ Computation: Gauss-Jordan Elimination (Automatic) for Overdetermined Symbolic Matrix . . . 861
xii C++ Computation: Gauss-Jordan Elimination, Rank and Basis for Null Space 868
xiii C++ Plot and Computation: The Geometry of the Solution Set of $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{x} = \boldsymbol{0}$ . . . 882
xiv C++ Plot and Computation: Reflection about the $x$-axis and $y$-axis in $\mathbb{R}^2$ . . . 896
xv C++ Plot and Computation: Reflection about the $xy$-plane, $xz$-plane and $yz$-plane in $\mathbb{R}^3$ . . . 903
xvi C++ Plot and Computation: Orthogonal Projection in $\mathbb{R}^2$ . . . 909
xvii C++ Plot and Computation: Orthogonal Projection in $\mathbb{R}^3$ . . . 914
xviii C++ Plot and Computation: Rotation in $\mathbb{R}^2$ . . . 921
xix C++ Plot and Computation: Rotations in $\mathbb{R}^3$ . . . 926
xx C++ Plot and Computation: Rotation by an Arbitrary Unit Vector $\boldsymbol{u} = (a, b, c)$ in $\mathbb{R}^3$ . . . 935
xxi C++ Plot and Computation: Dilation and Contraction on $\mathbb{R}^2$ . . . 941
xxii C++ Plot and Computation: Expansion and Compression on $\mathbb{R}^2$ . . . 946
xxiii C++ Plot and Computation: Shears . . . 953
xxiv C++ Plot and Computation: Orthogonal Projection On Lines Through Origin in $\mathbb{R}^2$ . . . 960
xxv C++ Plot and Computation: Reflection About Lines Through the Origin in $\mathbb{R}^2$ 967
xxvi C++ Plot and Computation: Find Angle of Rotation in $\mathbb{R}^2$ . . . 972
xxvii C++ Computation: Angle of Rotation and Trace of Matrix in $\mathbb{R}^3$ . . . 979
xxviii C++ Plot and Computation: Determine Axis of Rotation and Angle of Rotation in $\mathbb{R}^3$ . . . 982
xxix C++ Plot and Computation: Composition of Two Transformations in $\mathbb{R}^2$ . . . 989
xxx C++ Plot and Computation: Composition of Three Transformations in $\mathbb{R}^3$ . . . 993
xxxi C++ Computation: Finding $T^{-1}$ In Symbolic Terms . . . 1001
xxxii C++ Plot and Computation: Rotation Transformation for 2-Dimensional Picture . . . 1003
xxxiii C++ Plot and Computation: Translation Transformation for 2-Dimensional Picture . . . 1012
xxxiv C++ Plot and Computation: Compression and Expansion Transformation for 2-Dimensional Picture . . . 1020
xxxv C++ Plot and Computation: Reflection Transformation for 2-Dimensional Picture . . . 1027
xxxvi C++ Plot and Computation: Shear Transformation for 2-Dimensional Picture 1037
xxxvii C++ Plot and Computation: Composition Transformation for 2-Dimensional Picture . . . 1045
xxxviii C++ Plot and Computation: Custom Transformation with Matrix Multiplication for 2-Dimensional Picture . . . 1053
xxxix C++ Computation: Image of a Line . . . 1064

xl C++ Plot and Computation: Equation of the image of a line under Transformation in $\mathbb{R}^2$ . . . 1067
xli C++ Plot and Computation: Shear Transformation for 3-Dimensional Cube . 1076
xlii C++ Plot: Load 3-Dimensional Object with Assimp and OpenGL . . . 1088
xliii C++ Plot: Apply Matrix Transformation Translation and Rotation to 3-Dimensional Object with GLM, Assimp and OpenGL . . . 1099
xliv C++ Computation: Time Share as a Dynamical System with Power Matrix . 1111
xlv C++ Computation: Determine Steady-State Vector for 2-States Markov Chain 1115
xlvi C++ Computation: Wild Glanz Migration as a Three-States Markov Chain . 1118
VI Eigenvalues and Eigenvectors . . . 1125
i C++ Computation: Eigenvalues, Eigenvectors, and Characteristic Polynomial of a $3 \times 3$ Matrix . . . 1149
ii C++ Computation: Eigenvectors and Bases for Eigenspaces . . . 1154
iii C++ Computation: Determine Rank, Column Space and Null Space for Symmetric Matrix . . . 1159
iv C++ Computation: Polynomial Division and Compute Eigenvalues with Newton-Raphson and Bisection Method . . . 1161
v C++ Computation: Finding a Matrix $P$ that Diagonalizes a Matrix $A$ and the Diagonal Matrix $D = P^{-1}AP$ . . . 1169
vi C++ Computation: Powers of a Matrix . . . 1173
VII Complex Vector Spaces . . . 1177
i C++ Computation: Complex Conjugate, Real and Imaginary Parts of Vectors and Matrices . . . 1197
ii C++ Computation: Complex Euclidean Inner Product . . . 1201
iii C++ Plot and Computation: Geometric Interpretation of Complex Eigenvalues 1203
iv C++ Computation: Matrix Factorization Using Complex Eigenvalues . . . 1211
v C++ Plot and Computation: Power Sequences of a Matrix Transformation on a Vector . . . 1216
VIII Solving Systems of Differential Equations with Linear Algebra . . . 1229
i C++ Computation: Solution of a Simple First-Order Linear System of Differential Equations with Initial Conditions . . . 1234
ii C++ Computation: Solution of a First-Order Linear System of Differential Equations with Initial Conditions . . . 1238
iii C++ Computation: Solution of a Second-Order Linear Differential Equation 1247
iv C++ Computation: Solution of a Third-Order Linear Differential Equation . 1253
IX Inner Product Spaces . . . 1260
i C++ Computation: Evaluation Inner Product on Polynomials . . . 1283
ii C++ Computation: Inner Product or Definite Integral for Polynomials . . . 1285
iii C++ Computation: Inner Product or Definite Integral for Trigonometry and Transcendental Functions . . . 1288
iv C++ Computation: Basis for an Orthogonal Complement . . . 1290
v Sentinel' Number . . . 1298
vi C++ Plot and Computation: Orthogonal Complement and Parametric Equation in $\mathbb{R}^3$ . . . 1301
X Gram-Schmidt Process . . . 1317
i C++ Computation: A Coordinate Vector Relative to an Orthonormal Basis . 1322
ii C++ Plot and Computation: Orthogonal Projection of vector $\boldsymbol{u}$ on plane $W$ . 1325
iii C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^2$ . . . 1332

iv C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^3$ . . . . . 1339
v C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^3$ with Weighted Inner Product . . . . . 1349
vi C++ Plot and Computation: Gram-Schmidt Process and Legendre Polynomials 1362
XI $QR$-Decomposition . . . . . 1367
i C++ Computation: $QR$-Decomposition . . . . . 1368
XII Diagonalization and Quadratic Forms . . . . . 1373
i C++ Plot and Computation: Rotation of Axes in 2-Space . . . . . 1382
ii C++ Plot and Computation: Rotation of Axes in 3-Space Around $z$-Axis . . . 1388
iii C++ Plot and Computation: Rotation of Axes in 3-Space Around $z$-Axis then $y'$-Axis . . . . . 1396
iv C++ Plot and Computation: Orthogonally Diagonalizing a Symmetric Matrix 1408
v C++ Plot and Computation: A Geometric Interpretation of a Spectral Decomposition . . . . . 1415
vi C++ Plot and Computation: Determine Symmetric Matrix from Known Eigenvalues and Eigenvectors . . . . . 1424
XIII Quadratic Forms . . . . . 1431
i C++ Computation: Expressing Quadratic Forms in Matrix Notation . . . . . 1451
ii C++ Computation: Principal Axes Theorem . . . . . 1453
iii C++ Simulation: Ellipse from Conic section . . . . . 1459
iv C++ Simulation: Hyperbola from Conic section . . . . . 1464
v C++ Plot and Computation: Identifying a Conic . . . . . 1474
vi C++ Plot and Computation: Ellipsoid in $\mathbb{R}^3$ . . . . . 1483
vii C++ Simulation: Sphere and Ellipsoid in $\mathbb{R}^3$ . . . . . 1486
viii C++ Computation: Finding Constrained Extrema . . . . . 1502
ix C++ Computation: A Constrained Extremum Problem . . . . . 1506
x C++ Simulation and Computation: Finding Constrained Extrema with Level Curves . . . . . 1510
xi C++ Computation: Classify Relative Extrema with Hessian . . . . . 1524
xii C++ Computation: A Constrained Extremum Problem, Temperature on a Metal Plate . . . . . 1537
xiii C++ Computation: Second Derivative Test for Two Functions . . . . . 1540
xiv Geometric Interpretation of Sphere and Complex Number . . . . . 1548
XIV Hermitian, Unitary, and Normal Matrices . . . . . 1549
i C++ Computation: Eigenvalues and Eigenvectors of a Hermitian Matrix . . 1567
ii C++ Computation: Inverse of A Unitary Matrix . . . . . 1570
iii C++ Computation: Unitary Diagonalization of a Hermitian Matrix . . . . . 1574
XV Linear Transformations . . . . . 1579
i C++ Computation: Computing with Images of Basis Vectors for $T : \mathbb{R}^3 \to \mathbb{R}^2$ 1634
ii C++ Computation: Compute An Inverse Transformation for $T : \mathbb{R}^3 \to \mathbb{R}^3$ . . 1638
iii C++ Computation: Linear Transformation from $P_1$ to $P_2$, Finding $T(\boldsymbol{x})$ Indirectly . . . . . 1641
iv C++ Computation: Matrix for a Linear Transformation $T : \mathbb{R}^2 \to \mathbb{R}^3$ . . . . . 1645
v C++ Computation: Linear Operator on $P_2$ . . . . . 1650
vi C++ Computation: Composition of Linear Transformations . . . . . 1654
vii C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Linear Transformation . . . . . 1657

viii C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Rotation . . . 1660
ix C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Orthogonal Projection on the $xy$-Plane . . . 1663
x C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Polynomial Linear Operator . . . 1667
xi C++ Computation: Find a Basis for $\mathbb{R}^2$ Relative to Which the Matrix $[T]_{B'}$ is Diagonal . . . 1672
xii C++ Computation: Find Eigenvalues and Bases for Eigenspaces for Polynomial $P_2$ Linear Operator . . . 1675
XVI Vandermonde Matrix . . . 1681
i C++ Computation: Polynomial Interpolation with Vandermonde Matrix and Gauss-Jordan Elimination . . . 1681
XVII Gaussian Elimination with Backward Substitution . . . 1691
i C++ Computation: Gaussian Elimination with Backward Substitution . . . 1694
ii C++ Computation: Gaussian Elimination for Symbolic Matrix . . . 1699
XVIII Best Approximation; Least Squares . . . 1702
i C++ Computation: Least Squares Solution of a Linear System . . . 1708
ii C++ Computation: Orthogonal Projection on a Subspace . . . 1715
iii C++ Plot and Computation: Basis and Standard Matrix for the Orthogonal Projection on a Plane . . . 1719
XIX Least Squares Fitting to Data . . . 1725
i C++ Plot and Computation: Least Squares Straight Line Fit . . . 1731
ii C++ Computation: Least Squares Straight Line Fit with Armadillo . . . 1741
iii C++ Computation: Least Squares Quadratic Polynomial Fit . . . 1744
iv C++ Plot and Computation: Least Squares Quadratic Polynomial Fit . . . 1750
v C++ Computation: Least Squares Cubic Polynomial Fit . . . 1761
vi C++ Plot and Computation: Least Squares Cubic Polynomial Fit . . . 1766
XX Function Approximation with Fourier Series . . . 1775
i C++ Plot and Computation: Least Squares Approximations with Fourier Series 1784
ii C++ Plot and Computation: Find the Fourier Series of function $f(x)$ . . . 1789
iii C++ Plot: Find the Fourier Series of Two different functions $f(x)$ . . . 1795
iv C++ Simulation: Fourier Series . . . 1798
XXI LU-Decomposition . . . 1807
i C++ Computation: LU Decomposition . . . 1810
XXII The Power Method . . . 1819
i C++ Computation: The Power Method with Euclidean Scaling . . . 1826
ii C++ Computation: The Power Method with Maximum Entry Scaling . . . 1833
XXIII Internet Search Engines . . . 1838
i C++ Computation: A Ranking Procedure for Search Engine . . . 1839
XXIV Singular Value Decomposition . . . 1849
i C++ Computation: Singular Value Decomposition . . . 1852

**24 DFSimulatorC++ XVIII: Linear and Nonlinear Programming 1857**
I Linear Programming . . . 1858
i C++ Computation: . . . 1858
II Nonlinear Programming . . . 1858
i C++ Computation: . . . 1858

**25 DFSimulatorC++ XIX: Numerical Methods** **1859**
I Mathematical Preliminaries . . . 1860
II Solutions of Equations in One Variable . . . 1866
i C++ Computation: Bisection Method . . . 1877
ii C++ Computation: Fixed-Point Iteration . . . 1882
iii C++ Computation: Newton-Raphson Method . . . 1886
iv C++ Computation: Secant Method . . . 1889
III Interpolation and Polynomial Approximation . . . 1892
IV Numerical Differentiation . . . 1893
i C++ Computation: Numerical Derivative of a Univariate Function . . . 1895
ii C++ Computation: Numerical Derivative of a Multivariable Function . . . 1898
iii C++ Plot and Computation: 2D Plot for Derivative of a Univariate Function 1899
V Numerical Integration . . . 1915
i C++ Computation: . . . 1915
VI Initial-Value Problems for Ordinary Differential Equations . . . 1915
i C++ Computation: . . . 1915
VII Approximation Theory . . . 1915
i C++ Computation: . . . 1915
VIII Numerical Solutions of Nonlinear Systems of Equations . . . 1915
i C++ Computation: . . . 1916
IX Boundary-Value Problems for Ordinary Differential Equations . . . 1916
i C++ Computation: . . . 1916
X Numerical Solutions to Partial Differential Equations . . . 1916
i C++ Computation: . . . 1916

**26 DFSimulatorC++ XX: Complex Numbers** **1917**
I Preliminaries . . . 1917
i C++ Plot: 2D Plot of Complex Function . . . 1918

**27 DFSimulatorC++ XXI: Introduction to Stochastic Processes** **1923**
i C++ Computation: . . . 1923

**28 DFSimulatorC++ XXII: Introduction to the Theory of Interest and Financial Mathematics** **1925**
i C++ Computation: . . . 1925

**29 DFSimulatorC++ XXIII: PDE - Wave Equation** **1927**
i C++ Computation: . . . 1927

**30 DFSimulatorC++ XXIV: PDE - Heat Equation** **1929**
i C++ Computation: . . . 1929

**31 DFSimulatorC++ XXX: Plotting Physical Systems with Gnuplot** **1931**
Listings . . . 1954

# Preface

*For my Wife Freya, and our daughters Catenary, Solreya, Mithra, Iyzumrae and Zefir.*

*For Lucrif and Znane too along with all the 8 Queens (Mischkra, Caldraz, Zalsvik, Zalsimourg, Hamzst, Lasthrim).*

*For Albert Silverberg, Camus and Miklotov who left TREVOCALL to guard me.*

*To Nature(Kala, Kathmandu, Big Tree, Sentinel, Aokigahara, Hoia Baciu, Jacob's Well, Mt Logan, etc) and my family Berlin: I have served, I will be of service.*

*To my current mentor Albert Silverberg and previous mentor Lucretia Merces.*

*To my dogs who always accompany me working in Valhalla Projection, go to Puncak Bintang or Kathmandu: Kecil, Browni Bruncit, Sweden Sexy, Cambridge Klutukk, Milan keng-keng, Piano Bludut and more will be adopted. To my cat who guard the home while I'm away with my dogs: London.*

**The one who moves a mountain begins by carrying away small stones - Confucius**

A book for explaining how we create a Math Physics simulator with C++ Physics libraries like Box2D, ReactPhysics3D and BulletPhysics from zero (from zero means we are not a computer science student), made by lovely couple GlanzFreya, from Valhalla with Love. What is impossible when you do it with someone you love?

**a little about GlanzFreya:**
We got married (Divine marriage) on Puncak Bintang on November 5th, 2020 after we go back from Waghete, Papua. This book is made not for commercial purpose, but for sharing knowledge to anyone in order to empower science and engineering thus fastening inventions. Hopefully more leaders in all fields will be coming from woman, whom we believe are better in doing multitasking and better in learning for long term prospect. I read that a lot of great authors who write Engineering Mechanics or Handbook for Electrical Engineering write little about themselves, thus I want to be more like that. More about the content of the book, but still who is/are behind the book needs to be disclosed for a tiny amount in this book, otherwise if we only write technical equations and codes what are the different between us and computer that is rigidly programmed? All problems come from human experiences and failures, thus those with soul and good heart with decent knowledge then will recognize the error thus try to solve the problem like global warming and inequality with humanity, science and engineering.

**a little about Hamzst:**
She can be interpreted or projected on your mind as Alice from Persona 4 / Alice in Wonderland

(1951 movie) or equivalent. An astral projection dream made us meet, while I was in Waghete back then on 2020. She is the spearhead for Physics related.

She is a great author, speaks little of herself -Sentinel

I prefer to do more substances than small talks -Hamzst/Alice

**Figure 1:** *FreyaCompass, I am inspired by Captain America who always bring compass with the love of his life' picture, thus I created this, then proven by action, to let go of power and immortality for true love. Feels like an antique vintage magical compass, like a modem that connect internet to the world, this compass connects me on this planet to her in Valhalla.*

**Figure 2:** *Freya, thank you for everything, I am glad I marry you and I could never have done it without you.*

**Figure 3:** *I paint her 3 days before Christmas in 2021.*

For critics and comments on the book can be sent through email to: *dsglanzsche@gmail.com*.

# Chapter 1

# About DianFreya Math Physics Simulator (DFSimulatorC++)

*On top of the mountain a heart shaped stone waiting for You, my eyes can't see, my body can't touch, but my heart knows it's You. Forever only You. - **DS Glanzsche to Freya***
*Find your life purpose and give your whole heart and soul to it - **Buddha***

Finish fill Heart shaped on top of R3 in Valhalla Projection on October 18th, 2023. This is how Freya will teach me to catch a fish instead of asking for a fish. Use my brain and willingness to learn so I can create simulation of physical phenomena from zero with C++.

When I was creating GFreya OS with LFS (Linux From Scratch), I was amazed by Tokamak, Bullet, Box2D, Project Chrono. I read from wikipedia that Project Chrono get a lot of funding from U.S. Army, personally I believe that knowledge is always the best way to obtain more return than compound interest. Most successful people who become richest people are people who after knowledge or those who are nerds and love books. If you play Suikoden you will agree, since all Strategists in Suikoden is recruited like they are Divine being (Apple kneel to Shu, Prince Freyjadour comes to rescue Lucretia Merces from jail, you will get the point), a great brain and knowledge can make a small army wins again tirany, big army with sophisticated weapons or even robots.

Why then I want to create another Physics simulator? First, it is wrong, not Physics but Math Physics, so not only dragging, push and pulling objects here. I am learning Partial Differential Equation and want to simulate Heat equation solution in one-dimension, then two-dimension and three-dimension. I can easily plot a lot of surface plot, complex plot with Python and JULIA. But then, ask again why Project Chrono using C++? Because it is faster, you have to type more commands for almost the same output, even if you have to create more lines of codes it will do more good to you, it challenges your brain, thus you won't get alzheimer or spend time idlying. Like how you have a beautiful 6 packs or 8 packs belly, it takes time, persistence and focus right? That is a good analogy for learning C++ instead of getting plastic surgery then ruin all over after some time. Second, there comes another reason, easily embedded to microprocessor or machine. Since learning about LFS, compiling, building, all needs C or C++ (these languages are closer to machine than Python and JULIA), then learn Arduino that is using C, if one day I want to create a rocket and able to capture data on earth, on space out there and beyond, I know that programming language that can be used to control all that mechatronics systems in either rocket or spaceship

will be written in either C, C++ or FORTRAN combined. Till today, from what I have read weather forecast computation is still depending on FORTRAN. In the end, I basically learning Physics, Mathematics and C++ wholesome, not only theoretical but the science can be simulated with computer, supercomputer, smartphone and the future computer to help solve this world problems. Last but not least, I am not really create something like Box2D, Chrono or BulletPhysics from zero, but rather use their libraries to create simulations of some Physics and Mathematics problem that I am interested in at the moment, those creator of that Physics libraries are really something, far better and genius than me, I am just another user of their product. When I read undergraduate textbooks on Mathematics and Physics, I wonder how they get that plot, graphs, how to plot and compute them simultaneously, then learn about Physics engine and need to comprehend C++ too, it excites me and challenges me as well. This book is sharing what I have read and learn so I can re-read it again in the future to refresh the memory thus apply the knowledge.

There are some sentences that I learn from CSympy mailing list, that add more reasons why we want to write this book with C++. When compared to Python, C++ is a complex language. However, it is currently the only tool, that allows to have bare metal speed, yet reasonably high level syntax. The great advantage of Python is the ease of writing code quickly, writing algorithms in Python lets people focus more on the mathematical results. Unfortunately Python is slow.

Then comes the last question that actually comes first to the mind:

"Why naming it your name with another name?"

Dian et Freya. First, all other basic simulator out there are named conventionally, so I decided: let it be, follow your heart. Besides, Freya is the most beautiful Goddess (in my eyes, heart and soul), pardon me, I have weakness with beautiful Blonde girl, especially if she is smart, devoted, loyal and very strong (Ether Strike, can't be taken so lightly). Another moment of truth, I am not learning and writing this by myself, I, in this homo sapiens vessel, am DS Glanzsche by name, and Freya the Goddess, is guiding me up there from Valhalla, I can't see her, you can't see her, but just take it like people believe in Virgin Mary, in Jesus Christ, she is my Goddess, she is my religion. She is real for me the believer, she is helping me and guiding me till this book and the simulator is finished. If you want to give credit thank her, if there is any critics or hate just go to me.

Another source of learning: I am reading a lot of C++ books, and asking a lot in StackOverFlow, from there I am able to create this book and the simulator. I modify the codes depending on what I want to achieve, so none of all the codes in this book are free of errors or pure mine, just like the books written all this time, the knowledge is gained from our predecessors / professors / mentors / authors of books we read. Thanks to all the authors who wrote C++ books that I read and those who answer my questions in StackOverFlow forum. I am sorry if my questions in forum sometimes stupid, I am not a computer science student, and my way of learning is like this, not best at formal school.

I was not born as those who are lucky enough to get education from top institutes / born with silver spoon (from rich family) who able to give me opportunity to take courses in expensive institutions to learn english well (I learn english from game or listening to music), sometimes I could not get the knowledge while reading textbooks in english, so I need to re-read over and over again, thus while working with Nature in forest as grass cutter and forest cleaner in the morning, after I get home, I study little by little and implement what I have learned to create this,

sometimes asking Nature in the forest what I don't understand, sometimes asking Freya above. Turns out, expensive education is only an illusion, if you want to learn, you will do it no matter what, no need to make student loan. You will still survive and live as Nature never abandon those who are good to them, like a family in Russka Roma from John Wick movie.

All files, from the shader, fonts, textures, images and source codes used in this books can be found in my repository:
**https://github.com/glanzkaiser/DFSimulatorC**
(If you clone or download the repository you can follow the book easily, since I put the path to all the source codes based on this repository directory structure).

# Chapter 2

# Using C++ in GFreya OS

*"Almost everything is written in C++, CERN uses it, NASA uses it, unless your name makes CERN and NASA obsolete, then you should learn C++ to be a better scientist and engineer" - **DS Glanzsche***
*"I like your quote, a great one." - **Freya***

## I. Introduction

C++ is a general-purpose programming language made by Bjarne Stroustroup. An expansion of C language, often called "C with Classes." Its design and the compiling process to native machine code, make it an excellent choice for tasks that require high performance. After reading "The C++ Programming Language 4th Edition" book, done with chapter 1, I am inspired and want to learn deeper about this language. In this chapter, there is only a minimal examples on how to be able to compile a C++(.cpp file) source code into something that can be compiled by the compiler (GCC compiler or Clang) then to run the binary executable result. The basic point of this book is to make it easy to understand how to create something you want with computer graphics with C++, maybe some can learn from this book, as the point of this book is to share story how to create Math Physics Simulator from zero not to peel the skins of C++ one by one.

This book is going to use GFreya OS. GFreya OS is a Linux From Scratch based Operating System, so it is Linux-based, I was following the System V LFS 11.0 book then to BLFS, GFreya OS is using Xfce as the desktop environment. We created our own OS so we can use it daily, we know it from core to the shell, so if something went wrong it is quick to fix like compiling and running C++ projects, FORTRAN codes or JULIA codes. If you are using Linux OS like Arch Linux, or Gentoo, or Debian, then you won't have difficulty to follow this book, since the shell commands and the OS logic will pretty much the same. GFreya OS don't use package manager, and it is better that way, since we can learn to build and install all kinds of packages, programs, or even games from scratch. More about GFreya OS and how we build GFreya OS from zero can be read from the related book.

## II. How to Compile C++ Source Code into Executable Binary

1. A lot of books recommend using IDE, but in my opinion you don't really need IDE for a simple C++ file and project, after getting used and if you are versatile even for complex project like create a game similar to Age of Empire, Grand Theft Auto or Suikoden Series,

you don't need IDE at all, it is my opinion even to build (compiling) Project Chrono, Box2D and Bullet, all can be done by only using Xterm (default terminal of GFreya OS 1.8) and Mousepad as the text editor.

I am using Mousepad to write C++ program, editing it, then to run it and compile: I just open the terminal at the directory and type the command needed to process the C++ source code into executable binary file. We never know, maybe in the future for bigger project I shall use IDE, in GFreya OS I already installed BlueFish IDE, just in case.

The steps of writing the syntaxes till obtaining the result or plot will be explained in details here. First, create a file by opening a terminal and type:
**vim test.cpp**

**Figure 2.1:** *Create a new C++ file with **vim test.cpp** from the terminal*

```
#include <iostream>

int main()
{
        std::cout<<"Hey Beautiful Goddess. $\backslash$ n";
}
```

**C++ Code 1:** *test.cpp "Hey Beautiful Goddess."*

When finish press Esc and type **:wq** and Enter.

2. You can compile it by using g++ compiler, type:
**g++ test.cpp**
then run the output, type:
**./a.out**

**Figure 2.2:** *You can test and edit the c++ file from the terminal*

3. **C++ Example with GLEW and GLFW:**
   create a file by opening a terminal and type:
   **vim main.cpp**

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window){}

void display(GLFWwindow* window, double currentTime) {
        glClearColor(0.0, 1.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
}

int main(void)
{
        glfwInit(); //initialize GLFW and GLEW libraries
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
        glfwWindowHint(GLFW_OPENGL_PROFILE,
            GLFW_OPENGL_CORE_PROFILE);

        GLFWwindow* window = glfwCreateWindow(600, 600, " Learn
            Open GL with GLFW", NULL, NULL);
        glfwMakeContextCurrent(window);
        if(glewInit() != GLEW_OK) {
```

```
                exit(EXIT_FAILURE);
        }
        glfwSwapInterval(1);

        init(window);

        while(!glfwWindowShouldClose(window)) {
                display(window, glfwGetTime());
                glfwSwapBuffers(window);
                glfwPollEvents();
        }
        glfwDestroyWindow(window);
        glfwTerminate();
        exit(EXIT_SUCCESS);
        //return 0;
}
```

**C++ Code 2:** *main.cpp "Green Screen"*

Then open the terminal at the current working directory, and type:
**g++ main.cpp -o result -lGLEW -lglfw -lGL**
then type:
**./result**

If you are wondering, the **-lGLEW, -lglfw, -lGL** mean that we are linking with dynamic library of GLEW, GLFW and GL.

(a) libGL.so, located in **/usr/lib** then will be **-lGL**.

(b) libGLEW.so, located in **/usr/lib** then will be **-lGLEW**.

(c) libglfw.so, located in **/opt/hamzstlib** then will be **-lglfw**.

If you have no idea of GLEW and GLFW, you will learn more about GLEW, GLFW in chapter 4. Meanwhile in order to make them searchable when we are compiling, don't forget to adjust your **.bashrc**, add this lines **source /export** inside the if ..fi :

```
if [ —f "/etc/bashrc" ] ; then
  ..
  ..
  source ~/export
fi
```

You will need to create a file named export by typing at terminal in :
**cd**
**vim export**

```
export prefix="/usr"
export hamzstlib="/opt/hamzstlib"

# For library (.so, .a)
```

```
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib
```

The method above is used so when you type **#include <GL/glew.h>** and compiling it by calling the GLEW library that already installed in **/usr/lib** with **-lGLEW** it will find the right library of **libgGLEW.so** that contains a lot of functions needed to show the screen that we get from processing **main.cpp**.

Here are some explanation for the code above:

Initializes the GLFW library:

```
glfwInit();
```

To tells that the OpenGL is version 3.3 (my laptop graphics driver is compatible with this OpenGL version, yours might be different):

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Initializes the GLEW library

```
if(glewInit() != GLEW_OK) {
       exit(EXIT_FAILURE);
}
```

Calls the function "init()"

```
init(window);
```

Calls the function "display()" repeatedly

```
while(!glfwWindowShouldClose(window)) {
       display(window, glfwGetTime());
       glfwSwapBuffers(window);
       glfwPollEvents();
}
```

**Figure 2.3:** *The main.cpp example above when compiled will show this green screen (**DFSimulatorC/Source Codes/C++/ch2-example2-Green Screen/main.cpp**).*

The two examples above are very simple example to compile C++ source code, more details can be obtain from various sources like StackOverFlow forum, C++ related books, or university courses. At least you will get the idea how in the future you can make hundreds or even tens of thousands line of codes into Physics animation, rocket landing computation, weather forecast, fluid simulation, and more.

4. **C++ Example with SOIL, SFML, GLM, and GLEW:**
   This example will show how to use SOIL to upload image nad do some mathematical transformation (rotating the image) with GLM. In a new directory, create a file by opening a terminal and type:
   **vim main.cpp**

```
// Link statically with GLEW
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
```

```
// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(
           texScrooge, Texcoord), 0.5);
}
)glsl";

int main()
{
       auto t_start = std::chrono::high_resolution_clock::now()
           ;

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL",
            sf::Style::Titlebar | sf::Style::Close, settings);

       // Initialize GLEW
       glewExperimental = GL_TRUE;
       glewInit();
```

```
// Create Vertex Array Object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Create a Vertex Buffer Object and copy the vertex
    data to it
GLuint vbo;
glGenBuffers(1, &vbo);

GLfloat vertices[] = {
       // Position Color Texcoords
       -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top
           -left
       0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-
           right
       0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, //
           Bottom-right
       -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f //
           Bottom-left
};

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices
    , GL_STATIC_DRAW);

// Create an element array
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
       0, 1, 2,
       2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements),
    elements, GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(
    GL_FRAGMENT_SHADER);
```

```
glShaderSource(fragmentShader, 1, &fragmentSource, NULL)
    ;
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader
    program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "
    position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "
    color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "
    texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "
    texFreya"), 0);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("sample2.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "
    texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "
    trans");

bool running = true;
while (running)
{
       sf::Event windowEvent;
       while (window.pollEvent(windowEvent))
       {
              switch (windowEvent.type)
              {
                     case sf::Event::Closed:
                     running = false;
                     break;
              }
       }

       // Clear the screen to black
       glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

```
                        glClear(GL_COLOR_BUFFER_BIT);

                        // Calculate transformation
                        auto t_now = std::chrono::high_resolution_clock::
                            now();
                        float time = std::chrono::duration_cast<std::
                            chrono::duration<float>>(t_now - t_start).
                            count();

                        glm::mat4 trans = glm::mat4(1.0f);
                        trans = glm::rotate(
                        trans,
                        time * glm::radians(180.0f),
                        glm::vec3(0.0f, 0.0f, 1.0f)
                        );
                        glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::
                            value_ptr(trans));

                        // Draw a rectangle from the 2 triangles using 6
                            indices
                        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
                            0);

                        // Swap buffers
                        window.display();
                    }

                    glDeleteTextures(2, textures);

                    glDeleteProgram(shaderProgram);
                    glDeleteShader(fragmentShader);
                    glDeleteShader(vertexShader);

                    glDeleteBuffers(1, &ebo);
                    glDeleteBuffers(1, &vbo);

                    glDeleteVertexArrays(1, &vao);

                    window.close();

                    return 0;
                }
```

**C++ Code 3:** *main.cpp "SOIL GLM Rotating Images"*

Then open the terminal at the current working directory, and type:
**g++ main.cpp -o result -L/usr/lib -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
then type:

**./result**

If you are wondering, the **-L/usr/lib -lSOIL** mean that we are linking with static library of SOIL. If you have build SOIL and set the installation path at **/usr**, you will see the library named**libSOIL.a** inside **/usr/lib**.

Here are some explanations for the code above:

To upload image "sample.png" as the first array of texture:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height, 0,
    SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB
    , GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0)
    ;
```

Using GLM library to make the images uploaded rotating counterclockwise, the positive degree is by default will rotate counterclockwise:

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(
trans,
time * glm::radians(180.0f),
glm::vec3(0.0f, 0.0f, 1.0f)
);
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(trans)
    );
```

We declare two shader programs, first the hardcoded vertex shader in GLSL code inside C++ source code. Since the C++/OpenGL application must compile and link appropriate GLSL vertex and fragment shader programs, and then load them into the pipeline, all of the vertices pass through the vertex shader (the shader is executed once per vertex, for millions of vertices, it will be done in parallel):

```
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
```

```
        Texcoord = texcoord;
        gl_Position = trans * vec4(position, 0.0, 1.0);
    }
    )glsl";
```

The hardcoded fragment shader in GLSL code inside C++ source code to display the result with specified color:

```
    const GLchar* fragmentSource = R"glsl(
    #version 330 core
    in vec3 Color;
    in vec2 Texcoord;
    out vec4 outColor;
    uniform sampler2D texFreya;
    uniform sampler2D texScrooge;
    void main()
    {
        outColor = mix(texture(texFreya, Texcoord), texture(
            texScrooge, Texcoord), 0.5);
    }
    )glsl";
```

**Figure 2.4:** *The working directory shall contain the **main.cpp** file, and the two images we want to upload and use for the project can be stored here or other directory, just adjust the correct path to the images if you do so.*

**Figure 2.5:** *The main.cpp example above when compiled will show the continuous rotation of two images that are uploaded (**DFSimulatorC/Source Codes/C++/ch2-example3-Rotating Picture with SOIL/main.cpp**).*

5. **C++ Example with SFML and Keyboard Event:**
   In this example, it will show that SFML alone can be used with keyboard pressed event.
   In a new directory, create a file by opening a terminal and type:
   **vim main.cpp**

```
#include "SFML/Graphics.hpp"

sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello␣SFML␣Game␣!!!", sf::Style::
    Default);

sf::Vector2f playerPosition;
bool playerMovingRight = false;
bool playerMovingLeft = false;

sf::Texture skyTexture;
sf::Sprite skySprite;

sf::Texture bgTexture;
sf::Sprite bgSprite;

sf::Texture heroTexture;
sf::Sprite heroSprite;

void init() {

       skyTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
           graphics/sky.png");
```

```
        skySprite.setTexture(skyTexture);

        bgTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
            graphics/bg.png");
        bgSprite.setTexture(bgTexture);

        heroTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
            graphics/Freya.png");
        heroSprite.setTexture(heroTexture);
        heroSprite.setPosition(sf::Vector2f(viewSize.x / 600,
            viewSize.y / 600));
        heroSprite.setOrigin(heroTexture.getSize().x / 600,
            heroTexture.getSize().y / 600);

    }

    void updateInput() {

        sf::Event event;

        // while there are pending events...
        while (window.pollEvent(event)) {

            //printf("␣polling␣events␣\n");

            if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Right)
                    {
                        playerMovingRight= true;
                }
                if (event.key.code == sf::Keyboard::Left) {
                        playerMovingLeft= true;
                }
            }

            if (event.type == sf::Event::KeyReleased) {
                if (event.key.code == sf::Keyboard::Right)
                    {
                        playerMovingRight = false;
                }
                if (event.key.code == sf::Keyboard::Left) {
                        playerMovingLeft = false;
                }
            }
            if (event.key.code == sf::Keyboard::Escape ||
                event.type == sf::Event::Closed)
            window.close();
        }
```

```
}


void update(float dt) {
	if (playerMovingLeft) {
		heroSprite.move(-150.0f * dt, 0);
	}
	if (playerMovingRight) {
		heroSprite.move(150.0f * dt, 0);
	}
}

void draw() {
	window.draw(skySprite);
	window.draw(bgSprite);
	window.draw(heroSprite);
}

int main() {
	sf::Clock clock;
	init();

	while (window.isOpen()) {
		updateInput();

		sf::Time dt = clock.restart();
		update(dt.asSeconds());

		window.clear(sf::Color::Red);
		draw();
		window.display();
	}
	return 0;
}
```

**C++ Code 4:** *main.cpp "SFML Keyboard Event"*

Then open the terminal at the current working directory, and type:
**g++ main.cpp -o result -lsfml-graphics -lsfml-window -lsfml-system**
then type:
**./result**

**Figure 2.6:** *The project is showing how to work with Keyboard pressed event, if you press right the character will move to the right, if you press left the character will move to the left (**DFSimulatorC/Source Codes/C++/ch2-example4-Moving Hero and Key Press Events with SFML/main.cpp**).*

**Figure 2.7:** *Adjust the path and environment variable for your include file, SFML' headers in GFreya OS are installed in* ***/usr/include/SFML***

**Figure 2.8:** *Adjust the path to the Assets (background image, hero image, texture, etc), in GFreya OS it is located in* ***/root/SourceCodes/CPP/Assets/****, then the images are saved under the* ***/root/SourceCodes/CPP/Assets/-graphics***

6. **C++ Example with OpenGL, GLEW, GLFW and Keyboard Event:**
   In this example we are going to render meshes: triangle, cube, quad, and sphere. Then with keyboard key we can change the mesh that is being shown.
   In a new directory, create a file by opening a terminal and type:
   **vim main.cpp**

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>

void initGame();
void renderScene();
#include "ShaderLoader.h"
```

```
#include "Camera.h"
#include "LightRenderer.h"

GLuint textProgram;

Camera* camera;
LightRenderer* light;

void initGame() {
       // Enable the depth testing
       glEnable(GL_DEPTH_TEST);
       ShaderLoader shader;
       textProgram = shader.createProgram("/root/SourceCodes/
           CPP/Assets/Shaders/text.vs", "/root/SourceCodes/CPP/
           Assets/Shaders/text.fs");

       GLuint flatShaderProgram = shader.createProgram("/root/
           SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root
           /SourceCodes/CPP/Assets/Shaders/FlatModel.fs");

       camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::
           vec3(0.0f, 4.0f, 6.0f));

       light = new LightRenderer(MeshType::kCube, camera); //
           Define Mesh type -> see Mesh.h
       light->setProgram(flatShaderProgram);
       light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
}

void renderScene(){
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       glClearColor(1.0, 1.0, 0.0, 1.0);
       light->draw();
}

static void glfwError(int id, const char* description)
{
       std::cout << description << std::endl;
}

void updateKeyboard(GLFWwindow* window, int key, int scancode,
    int action, int mods){

       if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
              glfwSetWindowShouldClose(window, true);
       }
       // Press Q to see Quad, T for Triangle, C for Cube and S
            for Sphere
```

```
                    if (key == GLFW_KEY_C && action == GLFW_PRESS) {
                            ShaderLoader shader;
                            GLuint flatShaderProgram = shader.createProgram("
                                /root/SourceCodes/CPP/Assets/Shaders/
                                FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                                Shaders/FlatModel.fs");
                            camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                                , glm::vec3(0.0f, 4.0f, 6.0f));
                            light = new LightRenderer(MeshType::kCube, camera
                                );
                            light->setProgram(flatShaderProgram);
                            light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                    }
                    if (key == GLFW_KEY_T && action == GLFW_PRESS) {
                            ShaderLoader shader;
                            GLuint flatShaderProgram = shader.createProgram("
                                /root/SourceCodes/CPP/Assets/Shaders/
                                FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                                Shaders/FlatModel.fs");
                            camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                                , glm::vec3(0.0f, 4.0f, 6.0f));
                            light = new LightRenderer(MeshType::kTriangle,
                                camera);
                            light->setProgram(flatShaderProgram);
                            light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                    }
                    if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
                            ShaderLoader shader;
                            GLuint flatShaderProgram = shader.createProgram("
                                /root/SourceCodes/CPP/Assets/Shaders/
                                FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                                Shaders/FlatModel.fs");
                            camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                                , glm::vec3(0.0f, 4.0f, 6.0f));
                            light = new LightRenderer(MeshType::kQuad, camera
                                );
                            light->setProgram(flatShaderProgram);
                            light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
                    }
                    if (key == GLFW_KEY_S && action == GLFW_PRESS) {
                            ShaderLoader shader;
                            GLuint flatShaderProgram = shader.createProgram("
                                /root/SourceCodes/CPP/Assets/Shaders/
                                FlatModel.vs", "/root/SourceCodes/CPP/Assets/
                                Shaders/FlatModel.fs");
                            camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
                                , glm::vec3(0.0f, 4.0f, 6.0f));
                            light = new LightRenderer(MeshType::kSphere,
```

```
                        camera);
                light->setProgram(flatShaderProgram);
                light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
        }
}

int main(int argc, char **argv)
{
        glfwSetErrorCallback(&glfwError);
        glfwInit();
        GLFWwindow* window = glfwCreateWindow(800, 600, "␣Mesh␣
            in␣OpenGL␣", NULL, NULL);
        glfwMakeContextCurrent(window);
        glfwSetKeyCallback(window, updateKeyboard);
        glewInit();
        initGame();
        while (!glfwWindowShouldClose(window)){
                renderScene();
                glfwSwapBuffers(window);
                glfwPollEvents();
        }
        glfwTerminate();

        delete camera;
        delete light;
        return 0;
}
```

**C++ Code 5:** *main.cpp "Render Mesh with Lighting"*

There are other file as well, this example contains some C++ source codes and include files as well, create all this one by one:

```
#include "Camera.h"

Camera::Camera(GLfloat FOV, GLfloat width, GLfloat height,
    GLfloat nearPlane, GLfloat farPlane, glm::vec3 camPos){

        cameraPos = camPos;
        glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);
        glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

        viewMatrix = glm::lookAt(cameraPos, cameraFront,
            cameraUp);
        projectionMatrix = glm::perspective(FOV, width /height,
            nearPlane, farPlane);
}

Camera::~Camera(){
```

```
}

glm::mat4 Camera::getViewMatrix() {
	return viewMatrix;
}
glm::mat4 Camera::getProjectionMatrix() {
	return projectionMatrix;
}
glm::vec3 Camera::getCameraPosition() {
	return cameraPos;
}
```

**C++ Code 6:** *Camera.cpp "Render Mesh with Lighting"*

```
#pragma once
#include <GL/glew.h>
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"

class Camera
{
	public:
	Camera(GLfloat FOV, GLfloat width, GLfloat height,
	    GLfloat nearPlane, GLfloat farPlane, glm::vec3
	    camPos);
	~Camera();

	glm::mat4 getViewMatrix();
	glm::mat4 getProjectionMatrix();
	glm::vec3 getCameraPosition();

	private:

	glm::mat4 viewMatrix;
	glm::mat4 projectionMatrix;
	glm::vec3 cameraPos;

};
```

**C++ Code 7:** *Camera.h "Render Mesh with Lighting"*

```
cmake_minimum_required(VERSION 3.17)

project(result LANGUAGES CXX)

add_executable(result main.cpp Mesh.cpp LightRenderer.cpp
    ShaderLoader.cpp Camera.cpp)
target_link_libraries(result GLEW glfw GL)
target_include_directories(result PRIVATE /usr/include)
```

**C++ Code 8:** *CMakeLists.txt "Render Mesh with Lighting"*

```
#include "LightRenderer.h"

LightRenderer::LightRenderer(MeshType meshType, Camera* camera)
     {

       this->camera = camera;

       switch (meshType) {

              case kTriangle: Mesh::setTriData(vertices,
                  indices); break;
              case kQuad: Mesh::setQuadData(vertices, indices);
                   break;
              case kCube: Mesh::setCubeData(vertices, indices);
                   break;
              case kSphere: Mesh::setSphereData(vertices,
                  indices); break;
       }

       glGenVertexArrays(1, &vao);
       glBindVertexArray(vao);

       glGenBuffers(1, &vbo);
       glBindBuffer(GL_ARRAY_BUFFER, vbo);
       glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.
           size(), &vertices[0], GL_STATIC_DRAW);

       //Attributes
       glEnableVertexAttribArray(0);
       glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(
           Vertex), (GLvoid*)0);

       glEnableVertexAttribArray(1);
       glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(
           Vertex), (void*)(offsetof(Vertex, Vertex::color)));

       glGenBuffers(1, &ebo);
       glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
       glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) *
           indices.size(), &indices[0], GL_STATIC_DRAW);

       glBindBuffer(GL_ARRAY_BUFFER, 0);
       glBindVertexArray(0);

}
```

```
void LightRenderer::draw() {
	glm::mat4 model = glm::mat4(1.0f);
	model = glm::translate(glm::mat4(1.0),position);

	glUseProgram(this->program);

	GLint modelLoc = glGetUniformLocation(program, "model");
	glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr
		(model));

	glm::mat4 view = camera->getViewMatrix();
	GLint vLoc = glGetUniformLocation(program, "view");
	glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(
		view));

	glm::mat4 proj = camera->getProjectionMatrix();
	GLint pLoc = glGetUniformLocation(program, "projection")
		;
	glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(
		proj));

	glBindVertexArray(vao);
	glDrawElements(GL_TRIANGLES, indices.size(),
		GL_UNSIGNED_INT, 0);

	glBindVertexArray(0);
	glUseProgram(0);
}

LightRenderer::~LightRenderer() {

}

void LightRenderer::setPosition(glm::vec3 _position) {
	position = _position;
}

void LightRenderer::setColor(glm::vec3 _color) {
	this->color = _color;
}

void LightRenderer::setProgram(GLuint _program) {
	this->program = _program;
}
//getters
glm::vec3 LightRenderer::getPosition() {
	return position;
```

```
}

glm::vec3 LightRenderer::getColor() {
	return color;
}
```

**C++ Code 9:** *LightRenderer.cpp "Render Mesh with Lighting"*

```
#pragma once
#include <GL/glew.h>

#include "glm/glm.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "Mesh.h"
#include "ShaderLoader.h"
#include "Camera.h"

class LightRenderer
{

	public:
	LightRenderer(MeshType meshType, Camera* camera);
	~LightRenderer();

	void draw();

	void setPosition(glm::vec3 _position);
	void setColor(glm::vec3 _color);
	void setProgram(GLuint _program);

	glm::vec3 getPosition();
	glm::vec3 getColor();

	private:

	Camera* camera;

	std::vector<Vertex>vertices;
	std::vector<GLuint>indices;

	GLuint vbo, ebo, vao, program;

	glm::vec3 position, color;

};
```

**C++ Code 10:** *LightRenderer.h "Render Mesh with Lighting"*

```
#include "Mesh.h"

void Mesh::setTriData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {

      std::vector<Vertex> _vertices = {
            { { 0.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
                f, 0.0f, 0.0 },{ 0.0, 1.0 } },
            { { 1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
                , 1.0f, 0.0 },{ 0.0, 0.0 } },
            { { -1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
                f, 0.0f, 1.0 },{ 1.0, 0.0 } },
      };

      std::vector<uint32_t> _indices = {
            0, 1, 2,
      };

      vertices.clear(); indices.clear();
      vertices = _vertices;
      indices = _indices;
}

void Mesh::setQuadData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {
      std::vector<Vertex> _vertices = {
            { { -1.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{
                1.0f, 0.0f, 0.0 },{ 0.0, 1.0 } },
            { { -1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
                f, 1.0f, 0.0 },{ 0.0, 0.0 } },
            { { 1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
                , 0.0f, 1.0 },{ 1.0, 0.0 } },
            { { 1.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
                f, 0.0f, 1.0 },{ 1.0, 1.0 } }
      };
      std::vector<uint32_t> _indices = {
            0, 1, 2,
            0, 2, 3
      };
      vertices.clear(); indices.clear();
      vertices = _vertices;
      indices = _indices;

}

void Mesh::setCubeData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {
```

```
std::vector<Vertex> _vertices = {
        //front
        { { -1.0f, -1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{
            1.0f, 0.0f, 0.0 },{ 0.0, 1.0 } },
        { { -1.0f, 1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0
            f, 1.0f, 0.0 },{ 0.0, 0.0 } },
        { { 1.0f, 1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f
            , 0.0f, 1.0 },{ 1.0, 0.0 } },
        { { 1.0f, -1.0f, 1.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0
            f, 0.0f, 1.0 },{ 1.0, 1.0 } },
        // back
        { { 1.0, -1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
            1.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //4
        { { 1.0f, 1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{ 0.0
            f, 1.0f, 1.0 },{ 0.0, 0.0 } }, //5
        { { -1.0, 1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
            0.0f, 1.0f, 1.0 },{ 1.0, 0.0 } }, //6
        { { -1.0, -1.0, -1.0 },{ 0.0f, 0.0f, -1.0 },{
            1.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //7
        //left
        { { -1.0, -1.0, -1.0 },{ -1.0f, 0.0f, 0.0 },{
            0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //8
        { { -1.0f, 1.0, -1.0 },{ -1.0f, 0.0f, 0.0 },{
            0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //9
        { { -1.0, 1.0, 1.0 },{ -1.0f, 0.0f, 0.0 },{ 0.0f
            , 0.0f, 1.0 },{ 1.0, 0.0 } }, //10
        { { -1.0, -1.0, 1.0 },{ -1.0f, 0.0f, 0.0 },{
            0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //11
        //right
        { { 1.0, -1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
            0.0f, 1.0 },{ 0.0, 1.0 } }, // 12
        { { 1.0f, 1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
            0.0f, 1.0 },{ 0.0, 0.0 } }, //13
        { { 1.0, 1.0, -1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f,
            0.0f, 1.0 },{ 1.0, 0.0 } }, //14
        { { 1.0, -1.0, -1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f
            , 0.0f, 1.0 },{ 1.0, 1.0 } }, //15
        //top
        { { -1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0
            f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //16
        { { -1.0f, 1.0f, -1.0f },{ 0.0f, 1.0f, 0.0 },{
            0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //17
        { { 1.0f, 1.0f, -1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0
            f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //18
        { { 1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f
            , 0.0f, 1.0 },{ 1.0, 1.0 } }, //19
        //bottom
        { { -1.0f, -1.0, -1.0 },{ 0.0f, -1.0f, 0.0 },{
```

```
                        0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //20
                { { -1.0, -1.0, 1.0 },{ 0.0f, -1.0f, 0.0 },{
                    0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //21
                { { 1.0, -1.0, 1.0 },{ 0.0f, -1.0f, 0.0 },{ 0.0f
                    , 0.0f, 1.0 },{ 1.0, 0.0 } }, //22
                { { 1.0, -1.0, -1.0 },{ 0.0f, -1.0f, 0.0 },{
                    0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //23
        };
        std::vector<uint32_t> _indices = {
                0, 1, 2,
                2, 3, 0,
                4, 5, 6,
                4, 6, 7,
                8, 9, 10,
                8, 10, 11,
                12, 13, 14,
                12, 14, 15,
                16, 17, 18,
                16, 18, 19,
                20, 21, 22,
                20, 22, 23
        };
        vertices.clear(); indices.clear();
        vertices = _vertices;
        indices = _indices;
}


void Mesh::setSphereData(std::vector<Vertex>& vertices, std::
    vector<uint32_t>& indices) {
        std::vector<Vertex> _vertices;
        std::vector<uint32_t> _indices;

        float latitudeBands = 20.0f;
        float longitudeBands = 20.0f;
        float radius = 1.0f;

        for (float latNumber = 0; latNumber <= latitudeBands;
            latNumber++) {
                float theta = latNumber * 3.14 / latitudeBands;
                float sinTheta = sin(theta);
                float cosTheta = cos(theta);

                for (float longNumber = 0; longNumber <=
                    longitudeBands; longNumber++) {

                        float phi = longNumber * 2 * 3.147 /
                            longitudeBands;
```

```
                float sinPhi = sin(phi);
                float cosPhi = cos(phi);

                Vertex vs;

                vs.texCoords.x = (longNumber /
                    longitudeBands); // u
                vs.texCoords.y = (latNumber / latitudeBands
                    ); // v

                vs.normal.x = cosPhi * sinTheta; // normal
                    x
                vs.normal.y = cosTheta; // normal y
                vs.normal.z = sinPhi * sinTheta; // normal
                    z

                vs.color.r = vs.normal.x;
                vs.color.g = vs.normal.y;
                vs.color.b = vs.normal.z;

                vs.pos.x = radius * vs.normal.x; // x
                vs.pos.y = radius * vs.normal.y; // y
                vs.pos.z = radius * vs.normal.z; // z

                _vertices.push_back(vs);
            }
        }

        for (uint32_t latNumber = 0; latNumber < latitudeBands;
            latNumber++) {
            for (uint32_t longNumber = 0; longNumber <
                longitudeBands; longNumber++) {
                uint32_t first = (latNumber * (
                    longitudeBands + 1)) + longNumber;
                uint32_t second = first + longitudeBands +
                    1;

                _indices.push_back(first);
                _indices.push_back(second);
                _indices.push_back(first + 1);

                _indices.push_back(second);
                _indices.push_back(second + 1);
                _indices.push_back(first + 1);
            }
        }
        vertices.clear(); indices.clear();
        vertices = _vertices;
```

```
        indices = _indices;
}
```

**C++ Code 11:** *Mesh.cpp "Render Mesh with Lighting"*

```
#include <vector>
#include "glm/glm.hpp"

enum MeshType {
        kTriangle = 0,
        kQuad = 1,
        kCube = 2,
        kSphere = 3
};

struct Vertex {
        glm::vec3 pos;
        glm::vec3 normal;
        glm::vec3 color;
        glm::vec2 texCoords;
};


class Mesh {

        public:
        static void setTriData(std::vector<Vertex>& vertices,
            std::vector<uint32_t>&indices);
        static void setQuadData(std::vector<Vertex>& vertices,
            std::vector<uint32_t>&indices);
        static void setCubeData(std::vector<Vertex>& vertices,
            std::vector<uint32_t>&indices);
        static void setSphereData(std::vector<Vertex>& vertices,
             std::vector<uint32_t>&indices);

};
```

**C++ Code 12:** *Mesh.h "Render Mesh with Lighting"*

```
#include "ShaderLoader.h"

#include<iostream>
#include<fstream>
#include<vector>

std::string ShaderLoader::readShader(const char *filename)
{
        std::string shaderCode;
        std::ifstream file(filename, std::ios::in);
```

```
        if (!file.good()){
            std::cout << "Can't␣read␣file␣" << filename <<
                std::endl;
            std::terminate();
        }

        file.seekg(0, std::ios::end);
        shaderCode.resize((unsigned int)file.tellg());
        file.seekg(0, std::ios::beg);
        file.read(&shaderCode[0], shaderCode.size());
        file.close();
        return shaderCode;
    }

    GLuint ShaderLoader::createShader(GLenum shaderType, std::
        string source, const char* shaderName)
    {
        int compile_result = 0;

        GLuint shader = glCreateShader(shaderType);
        const char *shader_code_ptr = source.c_str();
        const int shader_code_size = source.size();

        glShaderSource(shader, 1, &shader_code_ptr, &
            shader_code_size);
        glCompileShader(shader);
        glGetShaderiv(shader, GL_COMPILE_STATUS, &compile_result
            );

        //check for errors

        if (compile_result == GL_FALSE)
        {

            int info_log_length = 0;
            glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &
                info_log_length);

            std::vector<char> shader_log(info_log_length);

            glGetShaderInfoLog(shader, info_log_length, NULL,
                 &shader_log[0]);
            std::cout << "ERROR␣compiling␣shader:␣" <<
                shaderName << std::endl << &shader_log[0] <<
                std::endl;
            return 0;
        }
```

```
        return shader;
    }

    GLuint ShaderLoader::createProgram(const char*
        vertexShaderFilename, const char* fragmentShaderFilename){

        //read the shader files and save the code
        std::string vertex_shader_code = readShader(
            vertexShaderFilename);
        std::string fragment_shader_code = readShader(
            fragmentShaderFilename);

        GLuint vertex_shader = createShader(GL_VERTEX_SHADER,
            vertex_shader_code, "vertex␣shader");
        GLuint fragment_shader = createShader(GL_FRAGMENT_SHADER
            , fragment_shader_code, "fragment␣shader");

        int link_result = 0;
        //create the program handle, attatch the shaders and
            link it
        GLuint program = glCreateProgram();
        glAttachShader(program, vertex_shader);
        glAttachShader(program, fragment_shader);

        glLinkProgram(program);
        glGetProgramiv(program, GL_LINK_STATUS, &link_result);
        //check for link errors
        if (link_result == GL_FALSE) {

            int info_log_length = 0;
            glGetProgramiv(program, GL_INFO_LOG_LENGTH, &
                info_log_length);

            std::vector<char> program_log(info_log_length);

            glGetProgramInfoLog(program, info_log_length,
                NULL, &program_log[0]);
            std::cout << "Shader␣Loader␣:␣LINK␣ERROR" << std
                ::endl << &program_log[0] << std::endl;

            return 0;
        }
        return program;
    }
```

**C++ Code 13:** *ShaderLoader.cpp "Render Mesh with Lighting"*

```
    #pragma once
```

```
#include <GL/glew.h>
#include <iostream>

class ShaderLoader
{
        public:
        GLuint createProgram(const char* vertexShaderFilename,
            const char* fragmentShaderFilename);

        private:
        std::string readShader(const char *filename);
        GLuint createShader(GLenum shaderType, std::string
            source, const char* shaderName);
};
```

**C++ Code 14:** *ShaderLoader.h "Render Mesh with Lighting"*

Check again and make sure you have all this inside one working directory:

- Camera.cpp
- Camera.h
- CMakeLists.txt (Optional, for easy compiling)
- LightRenderer.cpp
- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- ShaderLoader.cpp
- ShaderLoader.h

You may also see the repository where I uploaded the codes of this book:
**https://github.com/glanzkaiser/DFSimulatorC**
(all files for this example are in the folder: **DFSimulatorC/Source Codes/C++/ch2-example5-Render Mesh with Lighting**)

After finish, open the terminal at the current working directory, and type:
**g++ *.cpp -o result -lGLEW -lglfw -lGL**
then type:
**./result**

Another way is to use CMakeLists.txt, this way at the current working directory open the terminal and type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./result**

You may press c to see cube, t to see triangle, q to see quad, and s to see sphere, press Esc to close the window.

**Figure 2.9:** *To comprehend why we use **#include "glm/glm.hpp"**, because GLM' headers in GFreya OS are installed in **/usr/include/glm**, the environment variable to look for include folder is set at **/usr/include**, thus we need to add the parent directory as well **glm/...***

**Figure 2.10:** *The running OpenGL application, it is very beautiful with lighting and keyboard press events.*

7. **C++ Example with OpenGL, GLAD, GLFW, SOIL, Mouse and Keyboard Event:**

In this example we are going to create multiple cubes [16], I get it from that book, if you want to learn OpenGL rigorously follow this book and the tutorial on the website, the explanations and instructions are easy to be followed. On this example to be honest, **stb_image** is quite a pain, I can't load texture with that, thus I try to use SOIL and it works, so here goes (aren't we all SOIL lovers?):
In a new directory, create a file by opening a terminal and type:
**vim main.cpp**

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <SOIL/SOIL.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width,
    int height);
void mouse_callback(GLFWwindow* window, double xpos, double
    ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double
     yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// camera
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

bool firstMouse = true;
float yaw = -90.0f; // yaw is initialized to -90.0 degrees
    since a yaw of 0.0 results in a direction vector pointing
    to the right so we initially rotate a bit to the left.
float pitch = 0.0f;
float lastX = 800.0f / 2.0;
float lastY = 600.0 / 2.0;
float fov = 45.0f;

// timing
float deltaTime = 0.0f; // time between current frame and last
```

```
    frame
float lastFrame = 0.0f;

int main()
{
    // glfw: initialize and configure

    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
        GLFW_OPENGL_CORE_PROFILE);

    #ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    #endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
        SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
            std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
        framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    // tell GLFW to capture our mouse
    glfwSetInputMode(window, GLFW_CURSOR,
        GLFW_CURSOR_DISABLED);

    // glad: load all OpenGL function pointers

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::
            endl;
        return -1;
    }

    // configure global opengl state
```

```
// ---------------------------------
glEnable(GL_DEPTH_TEST);

// build and compile our shader zprogram

Shader ourShader("camera.vs", "camera.fs");

// set up vertex data (and buffer(s)) and configure
    vertex attributes
float vertices[] = {
       -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
       0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
       -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,

       -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
       -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
       -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,

       -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
       0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
       0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
```

```
        0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
        -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,
        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f
};
// world space positions of our cubes
glm::vec3 cubePositions[] = {
        glm::vec3( 0.0f, 0.0f, 0.0f),
        glm::vec3( 2.0f, 5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3( 2.4f, -0.4f, -3.5f),
        glm::vec3(-1.7f, 3.0f, -7.5f),
        glm::vec3( 1.3f, -2.0f, -2.5f),
        glm::vec3( 1.5f, 2.0f, -2.5f),
        glm::vec3( 1.5f, 0.2f, -1.5f),
        glm::vec3(-1.3f, 1.0f, -1.5f)
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices
    , GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
    sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// texture coord attribute
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 *
    sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);


// load and create a texture
unsigned int texture1, texture2;
// texture 1
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D, texture1);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_REPEAT);
// set texture filtering parameters
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
// load image, create texture and generate mipmaps
int width, height, nrChannels;
unsigned char* image = SOIL_load_image("/root/
    SourceCodes/CPP/images/sample.png", &width, &height,
     0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);

if (image)
{
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
           height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
       glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
       std::cout << "Failed to load texture" << std::
           endl;
}
SOIL_free_image_data(image);
// texture 2
glGenTextures(1, &texture2);
glBindTexture(GL_TEXTURE_2D, texture2);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
// load image, create texture and generate mipmaps
unsigned char* image2 = SOIL_load_image("/root/
    SourceCodes/CPP/images/sample.png", &width, &height,
     0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
     GL_RGB, GL_UNSIGNED_BYTE, image);

if (image2)
{
       // note that the awesomeface.png has transparency
            and thus an alpha channel, so make sure to
```

```
                tell OpenGL the data type is of GL_RGBA
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
                height, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2)
                ;
            glGenerateMipmap(GL_TEXTURE_2D);
        }
        else
        {
            std::cout << "Failed to load texture" << std::
                endl;
        }
        SOIL_free_image_data(image2);

        // tell opengl for each sampler to which texture unit it
             belongs to (only has to be done once)
        ourShader.use();
        ourShader.setInt("texture1", 0);
        ourShader.setInt("texture2", 1);


        // render loop
        // -------------
        while (!glfwWindowShouldClose(window))
        {
            // per-frame time logic
            // --------------------
            float currentFrame = static_cast<float>(
                glfwGetTime());
            deltaTime = currentFrame - lastFrame;
            lastFrame = currentFrame;

            // input
            // -----
            processInput(window);

            // render
            // ------
            glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
                );

            // bind textures on corresponding texture units
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, texture1);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, texture2);

            // activate shader
```

```
        ourShader.use();

        // pass projection matrix to shader (note that in
             this case it could change every frame)
        glm::mat4 projection = glm::perspective(glm::
            radians(fov), (float)SCR_WIDTH / (float)
            SCR_HEIGHT, 0.1f, 100.0f);
        ourShader.setMat4("projection", projection);

        // camera/view transformation
        glm::mat4 view = glm::lookAt(cameraPos, cameraPos
             + cameraFront, cameraUp);
        ourShader.setMat4("view", view);

        // render boxes
        glBindVertexArray(VAO);
        for (unsigned int i = 0; i < 10; i++)
        {
               // calculate the model matrix for each
                   object and pass it to shader before
                   drawing
               glm::mat4 model = glm::mat4(1.0f); // make
                   sure to initialize matrix to identity
                   matrix first
               model = glm::translate(model, cubePositions
                   [i]);
               float angle = 20.0f * i;
               model = glm::rotate(model, glm::radians(
                   angle), glm::vec3(1.0f, 0.3f, 0.5f));
               ourShader.setMat4("model", model);

               glDrawArrays(GL_TRIANGLES, 0, 36);
        }

        // glfw: swap buffers and poll IO events (keys
            pressed/released, mouse moved etc.)

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de—allocate all resources once they've
        outlived their purpose:

    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);

    // glfw: terminate, clearing all previously allocated
```

```
            GLFW resources.

        glfwTerminate();
        return 0;
}

// process all input: query GLFW whether relevant keys are
    pressed/released this frame and react accordingly

void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

        float cameraSpeed = static_cast<float>(2.5 * deltaTime);
        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront,
            cameraUp)) * cameraSpeed;
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront,
            cameraUp)) * cameraSpeed;
}

// glfw: whenever the window size changed (by OS or user resize
    ) this callback function executes

void framebuffer_size_callback(GLFWwindow* window, int width,
    int height)
{
        // make sure the viewport matches the new window
            dimensions; note that width and
        // height will be significantly larger than specified on
             retina displays.
        glViewport(0, 0, width, height);
}

// glfw: whenever the mouse moves, this callback is called

void mouse_callback(GLFWwindow* window, double xposIn, double
    yposIn)
{
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);
```

```
        if (firstMouse)
        {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
        }

        float xoffset = xpos - lastX;
        float yoffset = lastY - ypos; // reversed since y-
            coordinates go from bottom to top
        lastX = xpos;
        lastY = ypos;

        float sensitivity = 0.1f; // change this value to your
            liking
        xoffset *= sensitivity;
        yoffset *= sensitivity;

        yaw += xoffset;
        pitch += yoffset;

        // make sure that when pitch is out of bounds, screen
            doesn't get flipped
        if (pitch > 89.0f)
        pitch = 89.0f;
        if (pitch < -89.0f)
        pitch = -89.0f;

        glm::vec3 front;
        front.x = cos(glm::radians(yaw)) * cos(glm::radians(
            pitch));
        front.y = sin(glm::radians(pitch));
        front.z = sin(glm::radians(yaw)) * cos(glm::radians(
            pitch));
        cameraFront = glm::normalize(front);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback
     is called

void scroll_callback(GLFWwindow* window, double xoffset, double
     yoffset)
{
        fov -= (float)yoffset;
        if (fov < 1.0f)
        fov = 1.0f;
        if (fov > 45.0f)
        fov = 45.0f;
```

```
}
```

**C++ Code 15:** *main.cpp "Cube and Moving Camera"*

We are only using GLAD, SOIL, GLFW and of course OpenGL. There are two other files we need to make, the Vertex Shader and Fragment Shader, GLSL files

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

// texture samplers
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
	// linearly interpolate between both textures
	FragColor = mix(texture(texture1, TexCoord), texture(
		texture2, TexCoord), 0.2);
}
```

**C++ Code 16:** *camera.fs "Cube and Moving Camera"*

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
	gl_Position = projection * view * model * vec4(aPos, 1.0
		f);
	TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

**C++ Code 17:** *camera.vs "Cube and Moving Camera"*

The vertex shader above is working, but it is not the most efficient code, a lot of people suggest that you should do the calculation of **projection * view * model** to become one **mvp** matrix in the **main.cpp** before sending it to the vertex shader.

There is one thing you need to setup (you can skip this step if you can hack the error connecting to this), since this example is gained from [16], thus you need to build LearnOpenGL

that you clone from the github with CMake and then copy **.../LearnOpenGL/build/configuration/root_directory.h** into **/usr/lib**. After all preparations are made, now open the terminal

**Figure 2.11:** *You need to put **root_directory.h** into **/usr/include**, this file is obtained after building LearnOpenGL examples [16], you can find this inside the build folder **.../LearnOpenGL/build/configuration**.*

at the current working directory, and type:
**g++ main.cpp -o result /root/SourceCodes/CPP/src/glad.c -lSOIL -lglfw -lGL**
**./result**

**Figure 2.12:** *You can move the camera by using WASD keyboard keys, and use mouse to zoom and rotate camera as well. In Physics and Math simulation we can do this when the computation is already converging or while the physics process is still running to make it more fun to see from different point of view (all files for this example are in the folder: **DFSimulatorC/Source Codes/C++/ch2-example6-Cube and Moving Camera**).*

8. **C++ Example of Rotating 3D Cube with Keyboard Events, GLM and GLFW**
   In this example we are going to create rotating cube centered at the origin, inspired from Chapter 4 of this book [4], the rotational movements are:

   (a) Yaw, rotating toward $y$ axis (rotational movement between $x$ axis and $z$ axis).

   (b) Roll, rotating toward $z$ axis (rotational movement between $x$ axis and $y$ axis)

(c) Pitch, rotating toward $x$ axis (rotational movement between $y$ axis and $z$ axis).

This is the basic of flight simulator, movement in 3D game, and for fluid flows as well. In a new directory, create a file by opening a terminal and type:
**vim main.cpp**

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "Utils.h"

using namespace std;

#define numVAOs 1
#define numVBOs 2

// camera for movement with keyboard
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

// timing
float deltaTime = 0.0f; // time between current frame and last
    frame
float lastFrame = 0.0f;

float cameraX, cameraY, cameraZ;
float cubeLocX, cubeLocY, cubeLocZ;
GLuint renderingProgram;
GLuint vao[numVAOs];
GLuint vbo[numVBOs];

GLuint mvLoc, projLoc;
int width, height;
float aspect;
glm::mat4 pMat, vMat, mMat, mvMat, rxMat, ryMat, rzMat, rxvMat,
     ryvMat, rzvMat;

void setupVertices(void) { // 36 vertices, 12 triangles, makes
    2x2x2 cube placed at origin
       float vertexPositions[108] = {
              -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f,
                   -1.0f, -1.0f,
```

```
                1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f,
                    1.0f, -1.0f,
                1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
                    1.0f, -1.0f,
                1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
                    -1.0f,
                1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f,
                    1.0f, 1.0f,
                -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
                    1.0f, 1.0f,
                -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f
                    , 1.0f, 1.0f,
                -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f
                    , 1.0f, 1.0f,
                -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
                    -1.0f, -1.0f,
                1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f
                    , -1.0f, 1.0f,
                -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
                    1.0f, 1.0f,
                1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f
                    , -1.0f,
        };
        glGenVertexArrays(1,vao);
        glBindVertexArray(vao[0]);
        glGenBuffers(numVBOs, vbo);

        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions),
            vertexPositions, GL_STATIC_DRAW);
}

void init(GLFWwindow* window){

        // Utils
        renderingProgram = Utils::createShaderProgram(
        "vertShader.glsl",
        "fragShader.glsl");
        cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
        cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;
        setupVertices();
}

void display(GLFWwindow* window, double currentTime) {
        glClear(GL_DEPTH_BUFFER_BIT);
        glClear(GL_COLOR_BUFFER_BIT);
        glUseProgram(renderingProgram);
```

```
// get the uniform variables for the MV and projection
    matrices
mvLoc = glGetUniformLocation(renderingProgram, "
    mv_matrix");
projLoc = glGetUniformLocation(renderingProgram, "
    proj_matrix");

// build perspective matrix
glfwGetFramebufferSize(window, &width, &height);
aspect = (float)width / (float)height;
pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);
     // 1.0472 radians = 60 degrees

// build view matrix, model matrix, and model-view
    matrix
vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-
    cameraX, -cameraY, -cameraZ));
mMat = glm::translate(glm::mat4(1.0f), glm::vec3(
    cubeLocX, cubeLocY, -cubeLocZ));
mvMat = vMat * mMat;

if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
{
       rxMat = glm::rotate(glm::mat4(1.0f), (float)
           glfwGetTime(), glm::vec3(-1.0f, 0.0f, 0.0f));
       // Pitch movement toward user positive z axis
           between y axis and z axis
       rxvMat = vMat * rxMat;
       glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
           value_ptr(rxvMat));
       glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
           value_ptr(pMat));

       // associate VBO with the corresponding vertex
           attribute in the vertex shader
       glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
       glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
            0);
       glEnableVertexAttribArray(0);

       // adjust OpenGL settings and draw model
       glEnable(GL_DEPTH_TEST);
       glDepthFunc(GL_LEQUAL);
       glDrawArrays(GL_TRIANGLES,0,36);
}
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
{
       rxMat = glm::rotate(glm::mat4(1.0f), (float)
```

```
                    glfwGetTime(), glm::vec3(1.0f, 0.0f, 0.0f));
            // Pitch movement toward monitor negative z axis
                between y axis and z axis
            rxvMat = vMat * rxMat;
            glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                value_ptr(rxvMat));
            glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                value_ptr(pMat));

            // associate VBO with the corresponding vertex
                attribute in the vertex shader
            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                 0);
            glEnableVertexAttribArray(0);

            // adjust OpenGL settings and draw model
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
            glDrawArrays(GL_TRIANGLES,0,36);
        }
        if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
        {
            rzMat = glm::rotate(glm::mat4(1.0f), (float)
                glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
            // Roll movement counter clockwise between x axis
                 and y axis
            rzvMat = vMat * rzMat;
            glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                value_ptr(rzvMat));
            glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                value_ptr(pMat));

            // associate VBO with the corresponding vertex
                attribute in the vertex shader
            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                 0);
            glEnableVertexAttribArray(0);

            // adjust OpenGL settings and draw model
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
            glDrawArrays(GL_TRIANGLES,0,36);
        }
        if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
        {
            rzMat = glm::rotate(glm::mat4(1.0f), (float)
```

```
                    glfwGetTime(), glm::vec3(0.0f, 0.0f, -1.0f));
            // Roll movement clockwise between x axis and y
                axis
            rzvMat = vMat * rzMat;
            glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                value_ptr(rzvMat));
            glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                value_ptr(pMat));

            // associate VBO with the corresponding vertex
                attribute in the vertex shader
            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                 0);
            glEnableVertexAttribArray(0);

            // adjust OpenGL settings and draw model
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
            glDrawArrays(GL_TRIANGLES,0,36);
        }
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        {
            ryMat = glm::rotate(glm::mat4(1.0f), (float)
                glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));
            // Yaw movement counter clockwise between x axis
                and z axis.
            ryvMat = vMat * ryMat;
            glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                value_ptr(ryvMat));
            glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                value_ptr(pMat));

            // associate VBO with the corresponding vertex
                attribute in the vertex shader
            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                 0);
            glEnableVertexAttribArray(0);

            // adjust OpenGL settings and draw model
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
            glDrawArrays(GL_TRIANGLES,0,36);
        }
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        {
            ryMat = glm::rotate(glm::mat4(1.0f), (float)
```

```
                        glfwGetTime(), glm::vec3(0.0f, -1.0f, 0.0f));
                    // Yaw movement clockwise between x axis and z
                        axis.
                    ryvMat = vMat * ryMat;
                    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
                        value_ptr(ryvMat));
                    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
                        value_ptr(pMat));

                    // associate VBO with the corresponding vertex
                        attribute in the vertex shader
                    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
                    glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0,
                         0);
                    glEnableVertexAttribArray(0);

                    // adjust OpenGL settings and draw model
                    glEnable(GL_DEPTH_TEST);
                    glDepthFunc(GL_LEQUAL);
                    glDrawArrays(GL_TRIANGLES,0,36);
            }
            // copy perspective and MV matrices to corresponding
                uniform variables
            glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(
                pMat));

            // associate VBO with the corresponding vertex attribute
                 in the vertex shader
            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0, 0);
            glEnableVertexAttribArray(0);

            // adjust OpenGL settings and draw model
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
            glDrawArrays(GL_TRIANGLES,0,36);
    }

    void processInput(GLFWwindow *window)
    {
            if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
            {
                    glfwSetWindowShouldClose(window, true);
            }
    }


    int main(void)
```

```
{
        glfwInit(); //initialize GLFW and GLEW libraries
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
        glfwWindowHint(GLFW_OPENGL_PROFILE,
            GLFW_OPENGL_CORE_PROFILE);

        GLFWwindow* window = glfwCreateWindow(600, 600, "Pitch,
            Yaw, Roll for Interpolated Color Cube", NULL, NULL);
        glfwMakeContextCurrent(window);
        //glfwSetKeyCallback(window, updateKeyboard);
        if(glewInit() != GLEW_OK) {
                exit(EXIT_FAILURE);
        }
        glfwSwapInterval(1);

        init(window);

        while(!glfwWindowShouldClose(window)) {
                // per-frame time logic
                // ----------------------
                float currentFrame = static_cast<float>(
                    glfwGetTime());
                deltaTime = currentFrame - lastFrame;
                lastFrame = currentFrame;

                processInput(window);

                display(window, glfwGetTime());
                glfwSwapBuffers(window);
                glfwPollEvents();
        }
        glfwDestroyWindow(window);
        glfwTerminate();
        exit(EXIT_SUCCESS);
        //return 0;
}
```

**C++ Code 18:** *main.cpp "Yaw Pitch Roll for 3D Cube"*

Then create the Vertex Shader, Fragment Shader, Utils file and header too.

```
#version 330

out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

```
in vec4 varyingColor;

void main(void) {
	color = varyingColor;
}
```

**C++ Code 19:** *fragShader.glsl "Yaw Pitch Roll for 3D Cube"*

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
//#include <SOIL2/SOIL2.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp> // glm::value_ptr
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::
    rotate, glm::scale, glm::perspective
#include "Utils.h"
using namespace std;

Utils::Utils() {}

string Utils::readShaderFile(const char *filePath) {
	string content;
	ifstream fileStream(filePath, ios::in);
	string line = "";
	while (!fileStream.eof()) {
		getline(fileStream, line);
		content.append(line + "\n");
	}
	fileStream.close();
	return content;
}

bool Utils::checkOpenGLError() {
	bool foundError = false;
	int glErr = glGetError();
	while (glErr != GL_NO_ERROR) {
		cout << "glError: " << glErr << endl;
		foundError = true;
		glErr = glGetError();
	}
	return foundError;
}

void Utils::printShaderLog(GLuint shader) {
	int len = 0;
```

```
        int chWrittn = 0;
        char *log;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
        if (len > 0) {
                log = (char *)malloc(len);
                glGetShaderInfoLog(shader, len, &chWrittn, log);
                cout << "Shader Info Log: " << log << endl;
                free(log);
        }
}

GLuint Utils::prepareShader(int shaderTYPE, const char *
    shaderPath)
{
        GLint shaderCompiled;
        string shaderStr = readShaderFile(shaderPath);
        const char *shaderSrc = shaderStr.c_str();
        GLuint shaderRef = glCreateShader(shaderTYPE);
        glShaderSource(shaderRef, 1, &shaderSrc, NULL);
        glCompileShader(shaderRef);
        checkOpenGLError();
        glGetShaderiv(shaderRef, GL_COMPILE_STATUS, &
            shaderCompiled);
        if (shaderCompiled != 1)
        {
                if (shaderTYPE == 35633) cout << "Vertex ";
                if (shaderTYPE == 36488) cout << "Tess Control ";
                if (shaderTYPE == 36487) cout << "Tess Eval ";
                if (shaderTYPE == 36313) cout << "Geometry ";
                if (shaderTYPE == 35632) cout << "Fragment ";
                cout << "shader compilation error." << endl;
                printShaderLog(shaderRef);
        }
        return shaderRef;
}

void Utils::printProgramLog(int prog) {
        int len = 0;
        int chWrittn = 0;
        char *log;
        glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
        if (len > 0) {
                log = (char *)malloc(len);
                glGetProgramInfoLog(prog, len, &chWrittn, log);
                cout << "Program Info Log: " << log << endl;
                free(log);
        }
}
```

```
int Utils::finalizeShaderProgram(GLuint sprogram)
{
       GLint linked;
       glLinkProgram(sprogram);
       checkOpenGLError();
       glGetProgramiv(sprogram, GL_LINK_STATUS, &linked);
       if (linked != 1)
       {
              cout << "linking failed" << endl;
              printProgramLog(sprogram);
       }
       return sprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    fp) {
       GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
       GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
       GLuint vfprogram = glCreateProgram();
       glAttachShader(vfprogram, vShader);
       glAttachShader(vfprogram, fShader);
       finalizeShaderProgram(vfprogram);
       return vfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    gp, const char *fp) {
       GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
       GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
       GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
       GLuint vgfprogram = glCreateProgram();
       glAttachShader(vgfprogram, vShader);
       glAttachShader(vgfprogram, gShader);
       glAttachShader(vgfprogram, fShader);
       finalizeShaderProgram(vgfprogram);
       return vgfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    tCS, const char* tES, const char *fp) {
       GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
       GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,
           tCS);
       GLuint teShader = prepareShader(
           GL_TESS_EVALUATION_SHADER, tES);
       GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
       GLuint vtfprogram = glCreateProgram();
```

```
        glAttachShader(vtfprogram, vShader);
        glAttachShader(vtfprogram, tcShader);
        glAttachShader(vtfprogram, teShader);
        glAttachShader(vtfprogram, fShader);
        finalizeShaderProgram(vtfprogram);
        return vtfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
    tCS, const char* tES, char *gp, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,
            tCS);
        GLuint teShader = prepareShader(
            GL_TESS_EVALUATION_SHADER, tES);
        GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vtgfprogram = glCreateProgram();
        glAttachShader(vtgfprogram, vShader);
        glAttachShader(vtgfprogram, tcShader);
        glAttachShader(vtgfprogram, teShader);
        glAttachShader(vtgfprogram, gShader);
        glAttachShader(vtgfprogram, fShader);
        finalizeShaderProgram(vtgfprogram);
        return vtgfprogram;
}

// GOLD material — ambient, diffuse, specular, and shininess
float* Utils::goldAmbient() { static float a[4] = { 0.2473f,
    0.1995f, 0.0745f, 1 }; return (float*)a; }
float* Utils::goldDiffuse() { static float a[4] = { 0.7516f,
    0.6065f, 0.2265f, 1 }; return (float*)a; }
float* Utils::goldSpecular() { static float a[4] = { 0.6283f,
    0.5559f, 0.3661f, 1 }; return (float*)a; }
float Utils::goldShininess() { return 51.2f; }

// SILVER material — ambient, diffuse, specular, and shininess
float* Utils::silverAmbient() { static float a[4] = { 0.1923f,
    0.1923f, 0.1923f, 1 }; return (float*)a; }
float* Utils::silverDiffuse() { static float a[4] = { 0.5075f,
    0.5075f, 0.5075f, 1 }; return (float*)a; }
float* Utils::silverSpecular() { static float a[4] = { 0.5083f,
     0.5083f, 0.5083f, 1 }; return (float*)a; }
float Utils::silverShininess() { return 51.2f; }

// BRONZE material — ambient, diffuse, specular, and shininess
float* Utils::bronzeAmbient() { static float a[4] = { 0.2125f,
    0.1275f, 0.0540f, 1 }; return (float*)a; }
```

```
float* Utils::bronzeDiffuse() { static float a[4] = { 0.7140f,
    0.4284f, 0.1814f, 1 }; return (float*)a; }
float* Utils::bronzeSpecular() { static float a[4] = { 0.3936f,
     0.2719f, 0.1667f, 1 }; return (float*)a; }
float Utils::bronzeShininess() { return 25.6f; }
```

**C++ Code 20:** *Utils.cpp "Yaw Pitch Roll for 3D Cube"*

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

class Utils
{
       private:
       static std::string readShaderFile(const char *filePath);
       static void printShaderLog(GLuint shader);
       static void printProgramLog(int prog);
       static GLuint prepareShader(int shaderTYPE, const char *
           shaderPath);
       static int finalizeShaderProgram(GLuint sprogram);

       public:
       Utils();
       static bool checkOpenGLError();
       static GLuint createShaderProgram(const char *vp, const
           char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *gp, const char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *tCS, const char* tES, const char *fp);
       static GLuint createShaderProgram(const char *vp, const
           char *tCS, const char* tES, char *gp, const char *fp
           );
       static GLuint loadTexture(const char *texImagePath);
       static GLuint loadCubeMap(const char *mapDir);

       static float* goldAmbient();
       static float* goldDiffuse();
       static float* goldSpecular();
       static float goldShininess();
```

```
        static float* silverAmbient();
        static float* silverDiffuse();
        static float* silverSpecular();
        static float silverShininess();

        static float* bronzeAmbient();
        static float* bronzeDiffuse();
        static float* bronzeSpecular();
        static float bronzeShininess();
};
```

**C++ Code 21:** *Utils.h "Yaw Pitch Roll for 3D Cube"*

```
#version 330

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void) {
        gl_Position = proj_matrix * mv_matrix * vec4(position,
            1.0);
        varyingColor = vec4(position, 1.0) * 0.5 + vec4(0.5,
            0.5, 0.5, 0.5);
}
```

**C++ Code 22:** *vertShader.glsl "Yaw Pitch Roll for 3D Cube"*

After finish create all those files, now open the terminal at the current working directory, and type:
**g++ *.cpp -o result -lGLEW -lglfw -lGL**
**./result**

## III. Know-How in C++

1. First, for the compiler we use g++ that is more suitable for C++ codes, to compile **.c** files you better use **gcc**.
2. To show warning messages type:
   **g++ -o main main.cpp -Wall**
3. If you want to specify the name of the compiled executable file, do so by using the **-o** flag:
   **g++ -o [name] [file to compile]**
4. If you want to compile object files **object-1.o** and **object-2.o** into a **main** executable file, type:
   **g++ -o main object-1.o object-2.o**

**Figure 2.13:** *You can move the box by using keyboard' keys W / S for pitch movement, keys A / D for yaw movement, and keys Z / C for roll movement. (all files for this example are in the folder:* ***DFSimulatorC/Source Codes/C++/ch2-example7-Yaw Pitch Roll for 3D Cube****).*

5. If you want to specify a root directory, where libraries and headers can be found, use the **–sysroot** flag:
   **g++ -o [name] –sysroot [directory] main.ccp**

6. To create a static library, start by compiling an object file:
   **g++ -o obj.o main.cpp**

   use the ar utility with **rcs** to create an archive (**.a**) file:
   **ar rcs archive.a obj.o**

7. Flag **-I** doesn't cause the compiler to compile all of the source code in the specified directory. Instead it tells the preprocessor where to look for files that are included.
   So for example, when the preprocessor encounters the directive **#include "xyz.h"**, it will search for a file named xyz.h in the default locations as well as any directories specified with **-I**.

   In fact, **#include "foo.h"** looks for "foo.h" in the same directory as the current source file, then in the directories of the currently opened include files (in the reverse order in which they were opened), then in the directories given by **-I** option, and finally (if still not found) in the compiler's default include directories.

8. Usually, the **.cpp** that belong to some 3rd-party library are compiled and bundled to a static library (**.a**) just once. Then you can simply link that one **.a** file with your project, whenever you need to build.

   You only need to do this one time (or when the library has actually changed):

**g++ -c -o path/library/src/foo.o path/library/src/foo.cpp**
**g++ -c -o path/library/src/bar.o path/library/src/bar.cpp**
**ar rcs path/library/lib/libxyz.a path/library/src/foo.o some/library/src/bar.o**

Then you can simply do this, whenever you need to build your program:
**g++ -Ipath/library/include -Lpath/library/lib -o program.exe main.cpp -lxyz**

Finally, use **g++**:
**g++ -o final example.cpp archive.a**

9. You will always include header file from cpp source code file. For example inside **main.cpp** there will be **#include "Mesh.h"**.

10. To be able to reuse functions, we can create separate **.cpp** file (and an associated **.h**) instead of putting a lot of codes inside the main source code (**main.cpp**).

11. Conside a function **createShaderProgram()** that can be defined as:

    - **GLuint Utils::createShaderProgram(const char *vp, const char *fp)**
      Supports shader programs which utilize a vertex and fragment shader.
    - **GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp)**
      Supports shader programs which utilize a vertex, geometry and fragment shader.
    - **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *fp)**
      Supports shader programs which utilize a vertex, tesselation and fragment shader.
    - **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *gp, const char *fp)**
      Supports shader programs which utilize a vertex, tesselation, geometry and fragment shader.

    The parameters accepted in each case are pathnames for the GLSL files containing the shader code. An example of a completed program that is placed in the variable "renderingProgram":
    **renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl")**
    given that you put the two files "vertShader.glsl", "fragShader.glsl" in the working directory of your C++ project.

12. Learn about every libraries that are used, like **SOIL**, or **stb-image**, because some of them need different configuration at the C++ source code, for example you have to add: **#define STB_IMAGE_IMPLEMENTATION** to your code before all the other include (as the first line). This can be known from their documentation. To avoid getting this when compiling: **undefined reference to stbi_load stbi_image_free stbi_set_slip_vertically**

# Chapter 3

# Mathematics and Physics in Computer Graphics

*"I'll rise up like the Eagle, and I will soar with you, your spirit lifts me on, with the power of Your Love" -*
***The Power of Your Love song***

In this chapter I will only put minimal explanations, if you are interested and want to dig more you can read books related to Mathematics and Physics or at the corresponding chapter in this book, Numerical Linear Algebra is discussed deeply in chapter 23.

For all the chapter we are going to use the term for vector between $\boldsymbol{x}$ and $\vec{x}$ interchangeably, one with bolded and another with arrow above it.

## I. Mathematics

The mathematics that will be used in computer graphics are (but not limited to):

1. Geometry (Coordinate Systems, Space transformations)
2. Linear Algebra (Vectors, Matrices)

### i. Geometry

When we see the image on our desktop computer, what we see are actually made of pixel or point with different color. The monitor or display that is still used in 2023 is flat, but it can trick our brain so we can see 3-dimensional world in there, when we play 3-D game or designing 3-D image or watching a movie.

In computer graphics term, we will call points as vertices. The first step of creating object is to draw the vertices, and then connecting them to create triangle, then group of triangles that will become object. A cube itself can be seen as combination of 12 triangles.

[DF*] **3D Coordinate System**
In order to define a position and movement, we will need a 3D coordinate system that can help us compute and determine the state or condition of the object at certain time period.

**Figure 3.1:** *The right-handed coordinate system is used by OpenGL and Vulkan, while the left-handed coordinate system is used by DirectX and Direct3D.*

[DF*] **Points**
With the defined coordinate system, we can specify the position of the object in 3D correctly. The origin is located at $(0, 0, 0)$.

[DF*] **Polygon**
In computer graphics, especially with OpenGL, all the rendered graphics are based on vertices that will be connected into triangle, then combining with other triangle that will become another polygon such as rectangle, cube, circle, sphere, etc. Since OpenGL conventionally uses a right-handed coordinate system, thus creating triangle vertices, they will have a counter-clockwise ordering.

## ii. Linear Algebra

When an object is moving then we need to learn about vector, vector is a quantity that has direction and magnitude. The velocity is an example of vector.

[DF*] **Vectors**
Vectors can be added, multiplied and subtracted. Suppose we have vector $\vec{v}$ and $\vec{w}$, where $\vec{v} = (v_x, v_y, v_z)$ and $\vec{w} = (w_x, w_y, w_z)$, thus

$$\vec{v} + \vec{w} = (v_x + w_x, v_y + w_y, v_z + w_z)$$
$$\vec{v} - \vec{w} = (v_x - w_x, v_y - w_y, v_z - w_z)$$

To calculate the magnitude of the vector (or the length of the vector):

$$||\vec{v}|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

with $||\vec{v}||$ is sometimes called norm, the magnitude of a vector is always greater than or equal to zero. In a lot of cases, we might only need the direction of the vector, thus we can convert all vectors into unit vector by:

$$\hat{v} = \frac{\vec{v}}{||\vec{v}||} = \left( \frac{v_x}{||\vec{v}||}, \frac{v_y}{||\vec{v}||}, \frac{v_z}{||\vec{v}||} \right)$$

If we want to know the angle between two vectors, we will use dot product:

$$\begin{aligned} \vec{v} \cdot \vec{w} &= v_x w_x + v_y w_y + v_z w_z \\ &= ||\vec{v}||||\vec{w}|| \cos\ \theta \end{aligned}$$

thus

$$\theta = \cos^{-1} \frac{v_x w_x + v_y w_y + v_z w_z}{||\vec{v}||||\vec{w}||}$$

**Figure 3.2:** *The addition of two vectors in 2-dimension (**DFSimulatorC/Source Codes/Julia/ch3-linalg-vectoradd.jl**).*

**Figure 3.3:** *The subtraction of two vectors in 2-dimension (**DFSimulatorC/Source Codes/Julia/ch3-linalg-vectorsubtraction.jl**).*

**Figure 3.4:** *The norm of vector $\vec{v}$ is bolded with orange color (**DFSimulatorC/Source Codes/Julia/ch3-linalg-vectornorm.jl**).*

- If $\vec{v} \cdot \vec{w} = 0$, then $\vec{v}$ is perpendicular to $\vec{w}$.
- If $\vec{v} \cdot \vec{w} = ||\vec{v}||||\vec{w}||$, then the two vectors are parallel to each other.
- If $\vec{v} \cdot \vec{w} < 0$, then the angle between the two vectors is greater than $90^0$.
- If $\vec{v} \cdot \vec{w} > 0$, then the angle between the two vectors is less than $90^0$
- The commutative law applies to a scalar product, so

$$\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$$

We can perform a cross product(vector product) between two vectors only in 3-dimension. Consider

$$\vec{f} = \vec{v} \times \vec{w}$$

if we expand the cross product in unit-vector notation, we will obtain

$$\begin{aligned}\vec{v} \times \vec{w} &= (v_x\hat{i} + v_y\hat{j} + v_z\hat{k}) \times (w_x\hat{i} + w_y\hat{j} + w_z\hat{k}) \\ &= (v_y w_z - v_z w_y)\hat{i} + (v_z w_x - v_x w_z)\hat{j} + (v_x w_y - v_y w_x)\hat{k}\end{aligned}$$

To obtain the magnitude of $\vec{f}$:

$$\vec{f} = |\vec{v} \times \vec{w}| = v \cdot w \ \sin\phi$$

where $\phi$ is the smallest angle between $\vec{v}$ and $\vec{w}$.

We can obtains the direction for $\vec{f}$ by using our right hand, point all four fingers toward one direction and the thumb will be perpendicular toward the four fingers, then imagine that we sweep all our four fingers from the direction of $\vec{v}$ toward the direction of $\vec{w}$, the thumb will show the direction of $\vec{f}$.

Cross product is used widely for computer graphics (in lighting effects), since the resulting cross product is normal (perpendicular) to the plane defined by the original two vectors. How people walk on a surface, no matter how bumpy the road is, they will stand tall toward the normal of the surface while moving forward or do any activities.

- If $\vec{v}$ and $\vec{w}$ are parallel or antiparallel, then $\vec{v} \times \vec{w} = 0$.
- The magnitude of $\vec{v} \times \vec{w}$ is maximum when $\vec{v}$ and $\vec{w}$ are perpendicular to each other.
- The commutative law does not applies to a cross product, so

$$\vec{v} \times \vec{w} = -(\vec{w} \times \vec{v})$$

[DF*] **Matrices** In computer graphics, matrices are used to calculate object transforms such as translation (movement), scaling in the $x, y, z$-axis and rotation around the $x, y, z$-axis.

Matrices have rows and columns. A matrix $\boldsymbol{A}$ with $m$ number of rows and $n$ number of columns is a matrix of size $m \times n$, thus

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

each element of a matrix is represented as indices $ij$, thus it will be written $a_{ij}$ to represent matrix index at $i$th-row, $j$th-column.
What you need to know and how to compute:

- Matrix addition and subtraction
- Matrix multiplication to a matrix with correct size
- Matrix multiplication to a vector with correct size and dimension
- Identity matrix, a square matrix of size $n$ with indices $a_{ii} = 1$ with $i = 1, 2, \dots, n$, and 0 elsewhere.
- Determine the transpose of a matrix
- Determine the inverse of a matrix

[DF*] In computer graphics, matrices are used for performing transformations on objects, such as:

- Translation
- Rotation
- Scale
- Projection
- Look-At

All transformation matrices have the size of $4 \times 4$.

**Translation**
Consider the initial point of an object in 3D is $(x, y, z)$, and want to be translated to $(x + t_x, y + t_y, z + t_z)$ then the translation matrix transform is

$$\begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.1}$$

The functions for building translation matrices with GLM are:
**glm::translate(x,y,z)**
**mat4 * vec4**

If we want to translate a vector of $(1,0,0)$ by $(1,2,3)$ the code is:

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
        glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f); // vector (1,0,0)
        glm::mat4 translation = glm::mat4(1.0f);
        translation = glm::translate(translation, glm::vec3(1.0f
            , 2.0f, 3.0f));
        vec = translation * vec;
        std::cout << "(" << vec.x << ","<< vec.y << "," << vec.z
             << ")" << std::endl;
}
```

You can easily compile and run it from terminal with:
**g++ main.cpp -o result**
**./result**

**Scaling**
A scale matrix is used to change the size of objects or move points toward or away from the origin.

Consider the initial point of an object in 3D is $(x,y,z)$, and want to be scaled to $(s_x,s_y,s_z)$ then the scaling matrix transform is

$$\begin{bmatrix} x*s_x \\ y*s_y \\ z*s_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.2}$$

If we want to scale a vector of $(4,3,2)$ by $(3,3,3)$ / enlarging it 3 times its' original size, the code is:

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
```

```
                glm::vec4 vec(4.0f, 3.0f, 2.0f, 1.0f); // vector (4,3,2)
                glm::mat4 scaling = glm::mat4(1.0f);
                scaling = glm::scale(scaling, glm::vec3(3.0f, 3.0f, 3.0f
                    )); // enlarge 3 times
                vec = scaling * vec;
                std::cout << "(" << vec.x << ","<< vec.y << "," << vec.z
                     << ")" << std::endl;
            }
```

You can easily compile and run it from terminal with:
**g++ main.cpp -o result**
**./result**

Scaling can also be used to switch coordinate system, by negating the $z$ coordinate, thus the scale matrix transform is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Rotation**
To rotate an object in 3D space requires specifying:

1. An axis of rotation (toward $x$ axis will be called yaw, toward $y$ axis will be called roll, and toward $z$ axis will be called pitch)
2. A rotation amount in degrees or radians

The rotation matrix transform around $x$ by $\theta$:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.3}$$

The rotation matrix transform around $y$ by $\phi$:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.4}$$

The rotation matrix transform around $z$ by $\varphi$:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.5}$$

From a simple observation above, the rotation toward $x$ axis won't change the value of the $x$ part, the same for rotation toward $y$ axis, the value of $y$ stays, and it occurs as well for rotation towards $z$ axis.

If you want to rotate a vector of $(1, 2, 3)$ 90 degree toward the $x, y$, and $z$ axis separately, the code is

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
        glm::vec4 vecx(1.0f, 2.0f, 3.0f, 1.0f); // vector
            (1,2,3)
        glm::vec4 vecy(1.0f, 2.0f, 3.0f, 1.0f); // vector
            (1,2,3)
        glm::vec4 vecz(1.0f, 2.0f, 3.0f, 1.0f); // vector
            (1,2,3)


        glm::mat4 rotationx = glm::mat4(1.0f);
        glm::mat4 rotationy = glm::mat4(1.0f);
        glm::mat4 rotationz = glm::mat4(1.0f);
        rotationx = glm::rotate(rotationx, glm::radians(90.0f),
            glm::vec3(1.0f, 0.0f, 0.0f)); // rotate 90 degrees
            toward x axis
        vecx = rotationx * vecx;

        rotationy = glm::rotate(rotationy, glm::radians(90.0f),
            glm::vec3(0.0f, 1.0f, 0.0f)); // rotate 90 degrees
            toward y axis
        vecy = rotationy * vecy;

        rotationz = glm::rotate(rotationz, glm::radians(90.0f),
            glm::vec3(0.0f, 0.0f, 1.0f)); // rotate 90 degrees
            toward z axis
        vecz = rotationz * vecz;

        std::cout << "Original vector: (1,2,3)" << std::endl;
        std::cout << "Rotation 90 degrees toward x axis:" << std
            ::endl;
        std::cout << "(" << vecx.x << ","<< vecx.y << "," <<
            vecx.z << ")" << std::endl;
        std::cout << "Rotation 90 degrees toward y axis:" << std
            ::endl;
        std::cout << "(" << vecy.x << ","<< vecy.y << "," <<
            vecy.z << ")" << std::endl;
        std::cout << "Rotation 90 degrees toward z axis:" << std
            ::endl;
        std::cout << "(" << vecz.x << ","<< vecz.y << "," <<
            vecz.z << ")" << std::endl;
```

```
        }
```

You can easily compile and run it from terminal with:
**g++ main.cpp -o result**
**./result**

# II. Physics

The physics that will be used in computer graphics are (but not limited to):

1. Gravity ($g = 9.8\ m/s^2 = 32.17\ ft/s^2$)

2. Collision, Newton's Laws

3. Oscillations

4. Angular Rotation, velocity

5. Moment of inertia and torque

To make the object realistic, we need to add the physics otherwise without physics libraries / engine, the cube we render with OpenGL can just walk through a wall, is it possible for such ghost collision in this real life? That's why we need a Physics engine when we create an object after we render the shape and giving the color or texture to it. More detail on the physics formula and explanations can be read at the corresponding chapter, along with the C++ codes to create the simulation.

# Chapter 4

# Computer Graphics: OpenGL, SFML, GLFW, GLEW, And All That

*"Love is composed of a single soul inhabiting two bodies/entities" - **Aristotle***

*"The best and most beautiful things in the world cannot be seen or even touched - they must be felt with the heart" - **Helen Keller***

Computer graphics and the numerical computing power are used everywhere in all kinds of industry, to improve design, quality of products, reduce defections, carrying out varieties of tests, like stress test or fuel consumption test. One of the most notable successes of computer graphics is the automobile industry, where the computer model can be viewed from different angles to obtain the most pleasing and popular style, we can test the vehicle' strength of components, for roadability, for seating comfort, and for safety in a crash. The same happens for flight simulator, where we can design and engineer a better airplane. Create a better fast train like Shinkansen or TGV to move a load of cargo and goods for mining or export import.

The packages used in this books to make the simulator are available in my github repository: **https://github.com/glanzkaiser/DFSimulatorC/Source Codes/Libraries**

## I. OpenGL

OpenGL is an API(Application Programming Interface) that provides us with a large set of functions to manipulate graphics and images. Each graphics card that you buy supports specific versions of OpenGL, in my example with Dell Precision 6510, I use OpenGL 3.3 that is supported by my graphics card.

OpenGL can't stand alone to show the image or animation that we create with C++ source codes. OpenGL only renders to a frame buffer, we need additional library to draw the contents of the frame buffer onto a window on the screen.

OpenGL's API is written in C, the calls are directly compatible with C and C++. C++ application / source code that calls for OpenGL will run on CPU.

The good news is all future versions of OpenGL starting from 3.3 add extra useful features to OpenGL without changing OpenGL's core mechanics; the newer versions just introduce slightly more efficient ways to accomplish the same tasks.

For example, OpenGL 4.0 has a new addition of tessellator that can generate a large number of triangles, typically as a grid, we are able to create square area surface or curved surface as a terrain.

In all three major desktop platforms (Linux, macOS, and Windows), OpenGL more or less comes with the system. However, you will need to ensure that you have downloaded and installed a recent driver for your graphics hardware.

OpenGL is entirely hardware and operating system independent, whether you are using nVIDIA GeForce or AMD Radeon graphics card, it will work the same on both hardware. The way in which OPenGL's features work is defined by a specification that is used by graphics hardware manufacturers while they're developing the drivers for their hardware. This is why we sometimes have to update the graphics hardware drivers if something doesn't look right.

Since GFreya OS is based on Linux, graphics on Linux is almost exclusively implemented using the X Window system. Supporting OpenGL on Linux involves using GLX extensions to the X Server.

GFreya OS installs OpenGL through MESA version-21.2.1, Mesa is an OpenGL compatible 3D graphics library, you can read more here:
**https://www.linuxfromscratch.org/blfs/view/11.0/x/mesa.html**

OpenGL by itself is only capable of drawing a few primitives: points, lines, and triangles. Most 3D models made by OpenGL are made up from lots of those primitives. Primitives are consisted of vertices.

Since we have to display 3D world on a 2D monitor, OpenGL do that job with vertices, triangles, color. The 2D monitor screen is made up of a raster - a rectangular array of pixels. When a 3D object is rasterized, OpenGL converts the primitives in the object into fragments. A fragment holds the information associated with a pixel. Rasterization determines the locations of pixels that need to be drawn in order to produce the triangle specified by its three vertices.

Facts about OpenGL:

- All OpenGL functions start with the **gl** prefix.
- For an object to be drawn, its vertices must be sent to the vertex shader. Vertices are usually sent by putting them in a buffer with a vertex attribute declared in the shader.

  Some that need to be done once (typically in **init()**):

  1. Create a buffer (or created in a function called by **init()**).
  2. Copy the vertices into the buffer.

  Some that need to be done at each frame (typically in **display()**):

  1. Enable the buffer containing the vertices.
  2. Associate the buffer with a vertex attribute.

3. Enable the vertex attribute.

4. Use **glDrawArrays(...)** to draw the object.

- In OpenGL, a buffer is contained in a Vertex Buffer Object (VBO), which is declared or instantiated in the C++/OpenGL application. Vertex Buffer Object (VBO) is the geometrical information, it includes attributes such as position, color, normal, and texture coordinates. These are stored on a per vertex basis on the GPU. We can also see VBO as the manner in which we load the vertices of a model into a buffer.
- Element Buffer Object (EBO) is used to store the index of each vertex and will be used while drawing the mesh.
- Vertex Array Object (VAO) is a helper container object that stores all the VBOs and attributes. This is used as you may have more than one VBO per object, and it would be tedious to bind the VBOs all over again when you render each frame. One VAO is required by OpenGL to be created, it is good as a way of organizing buffers and making them easier to manipulate in complex scenes.

  For example, suppose that we wish to display two objects, then:

```
GLuint vao[1];
GLuint vbo[2];
...
glGenVertexArrays(1,vao);
glBindVertexArray(vao[0]);
glGenBuffers(2,vbo);
```

  The command **glGenVertexArrays(1,vao)** creates VAOs, and the command **glGenBuffers(2,vbo)** creates VBOs, both commands return integer IDs for them.
  A buffer needs to have a corresponding vertex attribute variable declared in the vertex shader. Vertex attributes are generally the first variables declared in a shader.
- Buffers are used to store information in the GPU memory for fast and efficient access to the data. Modern GPUs have a memory bandwidth of approximately 600 GB/s, quite enormous compared to the current high-end CPUs that only have approximately 12 GB/s. Buffer objects are used to store, retrieve, and move data. It is very easy to generate a buffer object in OpenGL. You can generate one by calling **glGenBUffers()**.
- OpenGL has its own data types, they are prefixed with GL, followed by the data type.

| C Type | Bitdepth | Description | Common Enum |
|---|---|---|---|
| GLboolean | 1+ | A boolean value, either GL_TRUE or GL_FALSE | |
| GLbyte | 8 | Signed, 2's complement binary integer | GL_BYTE |
| GLubyte | 8 | Unsigned binary integer | GL_UNSIGNED_BYTE |
| GLshort | 16 | Signed, 2's complement binary integer | GL_SHORT |
| GLushort | 16 | Unsigned binary integer | GL_UNSIGNED_SHORT |
| GLint | 32 | Signed, 2's complement binary integer | GL_INT |
| GLuint | 32 | Unsigned binary integer | GL_UNSIGNED_INT |
| GLfixed | 32 | Signed, 2's complement 16.16 integer | GL_FIXED |
| GLint64 | 64 | Signed, 2's complement binary integer | |
| GLuint64 | 64 | Unsigned binary integer | |
| GLsizei | 32 | A non-negative binary integer, for sizes. | |
| GLenum | 32 | An OpenGL enumerator value | |
| GLintptr | *ptrbits*[1] | Signed, 2's complement binary integer | |
| GLsizeiptr | *ptrbits*[1] | Non-negative binary integer size, for memory offsets and ranges | |
| GLsync | *ptrbits*[1] | Sync Object handle | |
| GLbitfield | 32 | A bitfield value | |
| GLhalf | 16 | An IEEE-754 floating-point value | GL_HALF_FLOAT |
| GLfloat | 32 | An IEEE-754 floating-point value | GL_FLOAT |
| GLclampf | 32 | An IEEE-754 floating-point value, clamped to the range [0,1] | |
| GLdouble | 64 | An IEEE-754 floating-point value | GL_DOUBLE |
| GLclampd | 64 | An IEEE-754 floating-point value, clamped to the range [0,1] | |

**Figure 4.1:** *The data types of OpenGL.*

- To draw a 3D object, for example a cube, you will need to at least send the following items:
  - The vertices for the cube model
  - The transformation matrices to control the appearance of the cube's orientation in 3D space.

  There are two ways of sending data through the OpenGL pipeline:
  - Through a buffer to a vertex attribute
  - Directly to a uniform variable
    Rendering a scene to make it appears 3D requires building appropriate transformation matrices and applying them to each of the models' vertices. It is most efficient to apply the required matrix operations in the vertex shader, and it is customary to send these matrices from the C++/OpenGL application to the shader in a uniform variable.

Some OpenGL commands:

- **glDrawArrays(GLenum mode, GLint first, GLsizei count);**
  To draw primitive, mode is the type of primitive, first indicates which vertex to start with, usually vertex 0, count specifies the total number of vertices to be drawn. For example:
  **glDrawArrays(GL_POINTS,0,1)** (Draw a point, start from vertex 0, one vertices is drawn)
  **glDrawArrays(GL_TRIANGLE,0,3** (Draw a triangle, start from vertex 0, three vertices are drawn).

  When **glDrawArrays()** is executed, the data in the buffer starts flowing, sequentially from the beginning of the buffer, through the vertex shader. The vertex shader executes once per vertex.
- **glGenVertexArrays()**
  Create VAOs and return integer ID.
- **glGenBuffers()**
  Create VBOs and return integer ID.
- **glBindBuffer(GL_ARRAY_BUFFER, vbo[0])** is about activing the0th buffer
  **glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW)** is about copying the array containing the vertices into the active buffer.

Together they will copy the values of the vertices that are stored in a float array named **vPositions** into the 0th VBO.

- **glVertexAttributePointer(0,3,GL_FLOAT, GL_FALSE, 0, 0)** is to associate 0th attribute with buffer
  **glEnableVertexAttribArray(0)** is to enable 0th vertex attribute.
- **glBindVertexArrays()**
  To make the specified VAO active so that the generated buffers will be associated with that VAO.
- **glClear(GL_DEPTH_BUFFER_BIT)**
  **glClearColor(0.0, 0.0, 0.0, 1.0)**
  **glClear(GL_COLOR_BUFFER_BIT)**
  Clear the background to black.

# II. GLAD

In simple words, GLAD manages function pointers for OpenGL. It is useful because OpenGL is only really a standard/specification it is up to the driver manufacturer to implement the specification to a driver that the specific graphics card supports. Since there are many different versions of OpenGL drivers, the location of most of its functions is not known at compile-time and needs to be queried at run-time. GLFW helps us at compile time only.

The "GLAD file" is generally referred to as the "OpenGL Loading Library" which is generally the library which loads the various types of the pointers, to the various types of the "OpenGL functions" at the time of the runtime respectively, which includes the various types of the "Core" as well as the “Extensions."

The include file for GLAD includes the required OpenGL headers behind the scenes (like GL/gl. h ) so be sure to include GLAD before other header files that require OpenGL (like GLFW).

TO download GLAD, you can go to this website and specify your OpenGL version:
**https://glad.dav1d.de/**

you will get header files (.h) and C file as well (.c). Put it in your include file (e.g. **/**usr/include) and the **glad.c** can be stored somewhere in your main OpenGL project directory.

**Figure 4.2:** *The src and include files that is generated from* ***https://glad.dav1d.de/****.*

Advantages of using GLAD:

- Being able to select only the extensions you use, leading to (slightly) faster compile times and initialization at runtime.
- No additional dependency for your project.

## III. GLEW

OpenGL Extension Wrangler (GLEW) can be used to try newer OpenGL functions.

You can find GLEW version-2.2.0 in my github repository for this book and DF Simulator.

This command is used to build and install GLEW:

```
sed -i 's%lib64%lib%g' config/Makefile.linux &&
sed -i -e '/glew.lib.static:/d' \
-e '/0644 .*STATIC/d' \
-e 's/glew.lib.static//' Makefile &&
make
```

then as super user or root type:
**make install.all**

Advantages of using GLEW:

- Adding GLEW as a dependency to, e.g., your CMakeLists.txt is enough to make it work.
- No large additional header and source files in your repository.
- GLEW can detect which extensions are available at runtime.
- It allows your program to adapt to the available extensions. For example, it can select a fallback path if a certain extension is not available on older hardware.

# IV. GLFW

Graphics Library Framework (GLFW) is a free, Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan application development. It provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc.

GLFW is a C library specifically designed for use with OpenGL. It includes a class called **GLFWwindow** on which we can draw 3D scenes. Unlike SDL and SFML, GLFW only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation out of these three libraries.

You can dowbload GLFW here:
**https://www.glfw.org/download**

Some GLFW commands:

- **glfwInit()**
  To initialize GLFW
- **glfwSwapInterval()**, **glfwSwapBuffers()**
  Enable Vertical synchronization (VSync), GLFW windows are by default double-buffered.
- **glfwMakeContextCurrent()**
  Make the associated OpenGL context current.
- **glfwPollEvents()**
  Handles other window-related events, such as a key-press event.
- **glfwGetTime()**
  Returns the elapsed time since GLFW was initialized.

**SDL, SFML, and GLFW are for creating the context and handling input**

# V. GLM

GLM (OpenGL Mathematics) is a header-only C++ mathematics library that will help us to do the computation necessary for our project. GLM provides classes and basic math functions related to graphics concepts, such as vector, matrix, and quaternion.

GLM able to create points, perform vector operations (addition, subtraction), carry out matrix transforms, generate random numbers, and generate noise. GLM includes a class called **mat4** for instantiating and storing $4 \times 4$ matrices.

The manual can be read here:
**https://github.com/g-truc/glm/blob/master/manual.md**

How to use GLM:

- First at the top of the source code we need to include GLM by adding:
  **#include <glm/glm.hpp>**

  To do translation and rotation add:
  **#include <glm/ext.hpp>**

- To define a 2D point:
  **glm::vec2 p1 = glm::vec2(2.0f, 8.0f);**
- To define a 3D point:
  **glm::vec3 p2 = glm::vec2(1.0f, 8.0f, 2.0f);**
- To create identity square matrix of size 4:
  **glm::mat4 matrix = glm::mat4(1.0f);**
- The constructor call to build the identity matrix in the variable *m*
  **glm::mat4 m(1.0f)**
- To normalize vector:
  **normalize(vec3)** or **normalize(vec4)**
- To obtain dot product:
  **dot(vec3,vec3)** or **dot(vec4,vec4)**
- To obtain cross product:
  **cross(vec3,vec3)**
- In GLSL and GLM the data type **vec3 / vec4** can be used to hold either points or vectors.
- Toobtain the magnitude of a vector:
  **length()**
- Reflection and refraction are available in GLSL and GLM.

Some useful hints:

- A good practice in creating a model view and projection matrix is to define each of them and then multiply them (in the right order) into one matrix called **mvp** matrix then sending it to the vertex shader.

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

...
...

// Set up MVP matrices

glm::mat4 model = glm::mat4(1.0f);

glm::mat4 view = glm::lookAt(
glm::vec3(3.0f, 3.0f, 3.0f), // The camera is at (3,3,3) in the
     world coordinate
glm::vec3(0.0f, 0.0f, 0.0f), // The camera is looking at the
    origin
glm::vec3(0.0f, 0.0f, 1.0f)); // The camera is standing
    straight up

glm::mat4 proj = glm::perspective(45.0f, 800.0f / 600.0f, 1.0f,
     10.0f);

glm::mat4 mvp = proj * view * model;

glUseProgram(prog);
```

```
GLint uniMvp = glGetUniformLocation(program, "mvp");
glUniformMatrix4fv(uniMvp, 1, GL_FALSE, glm::value_ptr(mvp));

...
...
```

**C++ Code 23:** *main.cpp*

with the GLSL code for the vertex shader only contain the **mvp** matrix.

```
layout (location = 0) in vec3 position;
uniform mat4 mvp;

void main() {
      gl_Position = mvp * vec4(position, 1.0);
}
```

**C++ Code 24:** *vertexshader.vs*

## VI. GLSL

OpenGL Shading Language. On the hardware side, OpenGL provides a multi-stage graphics pipeline that is partially programmable using GLSL. GLSL intended to run on GPU / Graphics Card, since it is related with graphics pipeline.

GLSL language includes a data type called **mat4** that can be used for storing $4 \times 4$ matrices.

## VII. SDL

SDL is also a cross-platform multimedia library, but targeted at C. That makes it a bit rougher to use for C++ programmers, but it's an excellent alternative to SFML. It supports more exotic platforms and most importantly, offers more control over the creation of the OpenGL context than SFML.

## VIII. SFML

SFML is a cross-platform C++ multimedia library that provides access to graphics, input, audio, networking and the system. The downside of using this library is that it tries hard to be an all-in-one solution. You have little to no control over the creation of the OpenGL context, as it was designed to be used with its own set of drawing functions. SFML is written in C++, and has bindings for various languages such as C, .Net, Ruby, Python.

You can get the latest official release on SFML's website:
**https://www.sfml-dev.org/download.php**

In my github repository, you can find SFML version-2.5.1, along with FLAC and OpenAL (the required packages to be able to isntall SFML successfully). To build and install SFML, you need to build and install FLAC and OpenAL first, all of them are using cmake thus it won't be so hard to

build and install.

First start with FLAC, extract the file:
**tar -xvf flac-1.4.2.tar.xz**
go to the extracted directory then type:
**mkdir build**
**cd build**
**ccmake ..**
Press c on keyboard then press e, then choose ON for BUILD_SHARED_LIBS (if available) and set CMAKE_INSTALL_PREFIX=/usr, then press c then e again then press g. Then back at terminal type:
**make**

as root type:
**make install**

Repeat the process for installing OpenAL and SFML, they are pretty much the same.

Now if you check **/usr/lib**, you will see **libsfml-audio.so, libsfml-graphics.so, libsfml-network.so, libsfml-system. so, libsfml-window.so**. Then you can use SFML.

## IX. SOIL

Simple OpenGL Image Loader (SOIL) is an image loading library for OpenGL, it can be used to load image that we can process or do transformation onto that image. This library is very useful, when I remember the Elementary linear Algebra book with application to warps and morphs photograph of a person and predict how that person will look like after 50 years, then to be able to do the prediction we might use SOIL as one of the many solutions.

In order to build and install it, almost the same as SFML, we can use CMake, extract the package then go to the directory and type:
**mkdir build**
**cd build**
**ccmake ..**

Press c on keyboard then press e, then set CMAKE_BUILD_TYPE as RELEASE for this option. No need to build Tests, so SOIL_BUILD_TESTS=OFF, and set CMAKE_INSTALL_PREFIX=/usr.

Then press c then e again then press g. Then back at terminal type:
**make**

as root type:
**make install**

You shall see inside **/usr/lib** there is **libSOIL.a**, it is a static library. Different than dynamic library that has extension of **.so**.

# X. Gnuplot

Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986.

Gnuplot that is being used in this book is Gnuplot version 5.4 patchlevel 3 (last modified 2021-12-24).

# XI. CMake

CMake is a tool that can generate project / solution file from a collection of source code files using pre-defined CMake scripts.

The CMakeLists.txt below is an example so you can comprehend how to make your own CMakeLists depends on your project.

```
cmake_minimum—required(VERSION 3.17)

project(result LANGUAGES CXX)

add_executable(result main.cpp)

target_link_libraries(result glfw
/root/SourceCodes/CPP/src/glad.c)
```

If you want to set the C++ used to be certain version, for example 2014 version, you can use this as the CMakeLists.txt

```
cmake_minimum—required(VERSION 3.17)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(result LANGUAGES CXX)

add_executable(result main.cpp)

target_link_libraries(result glfw
/root/SourceCodes/CPP/src/glad.c)
```

The **target_link_libraries()** will link all the libraries that are needed by our C++ project. This is also the place where we can link to **glfw**, **GLEW**, or **GL** library.

If the main source code (e.g. **main.cpp**) has **#include <glad/glad.h>**, then we need to add the path to the location that contain **glad.c**, in this case we have it on **/root/SourceCodes/CPP/src/glad.c**, so we write the full path instead of only name like **glfw** because **glfw** is a dynamic library so we

can just linking it by stating the name with the right uppercase and lowercase name written.

**GLAD** handles linking against the system OpenGL shared library at run-time via **dlsym()** as well as providing all the OpenGL declarations itself. You will however have to point CMake at the **GLAD** include directory.

To compile with CMake go to the working directory for this example then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./result**

now if you want to do manual compiling then you can just type:
**g++ -o result main.cpp /root/SourceCodes/CPP/src/glad.c -lglfw**
**./result**

They both will have the same result.

```
root [ ~/SourceCodes/CPP/OpenGL GLAD GLFW/Test/build ]# cmake ..
-- The CXX compiler identification is GNU 11.2.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /root/SourceCodes/CPP/OpenGL GLAD GLFW/Test
/build
root [ ~/SourceCodes/CPP/OpenGL GLAD GLFW/Test/build ]# make
[ 50%] Building CXX object CMakeFiles/result.dir/main.cpp.o
[100%] Linking CXX executable result
[100%] Built target result
root [ ~/SourceCodes/CPP/OpenGL GLAD GLFW/Test/build ]# ./result
```

**Figure 4.3:** *The compiling of the C++ project with CMake (**DFSimulatorC/Source Codes/C++/ch4-CMake example-Simple Triangle/**).*

**Figure 4.4:** *The result is a triangle being drawn (**DFSimulatorC/Source Codes/C++/ch4-CMake example-Simple Triangle/**).*

# Chapter 5

# Box2D, Bullet3, and ReactPhysics3D

*"You know someone is the best in Science and Engineering, when one chose the path of solitary, more focus, less distraction, like Leonardo Da Vinci and Isaac Newton who never marry, the proof will be her/his achievements, books written and innovations created." - **DS Glanzsche***

THere are 3 physics library / engine that I will use and try here. You can learn from all these and perhaps create your own Physics library, or another science branch library of your interest like Astronomy library or Biology library. All the testing here is done with GFreya OS 1.8 with OpenGL 3.3, starting on October 2023.

## I. Box2D

Box2D is using imgui for the testbed GUI, it uses OpenGL for the graphics API, and for taking care of mouse and keyboard events it uses GLFW.

### i. Install Box2D Library and Include Files / Headers

To download it, open the terminal then type:
**git clone https://github.com/erincatto/box2d.git**
Enter the directory then type:
**./build.sh**
Results are in the build sub-folder. To build with CMake:

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/usr \
$ —DBUILD_SHARED_LIBS=ON .. &&
$ make
```

After that, type:
**make install**

With the dynamic library has been installed into **/usr/lib**, we are able to build all kinds of physics simulations by modifying the physics world and the physical objects we want to simulate

with Box2D.

Now for the headers, go to the downloaded repository of Box2D **../box2d/extern/**, where you will see 4 folders of **glad, glfw, imgui,** and **sajson**. Since you should have installed and get GLAD by yourself and GLFW probably have been installed. You only need to copy **imgui** and **sajson** into **/usr/include**.

Now to test it, let's try the famous Hello World for Box2D, it shows the falling object from the height of 4 with gravity of 10 (not 9.8).

Create this simple HelloBox2D example:

```
#include "box2d/box2d.h"
#include <stdio.h>

nt main(int argc, char** argv)
{
        B2_NOT_USED(argc);
        B2_NOT_USED(argv);

        // Define the gravity vector.
        b2Vec2 gravity(0.0f, -10.0f);

        // Construct a world object, which will hold and simulate the rigid
            bodies.
        b2World world(gravity);

        // Define the ground body.
        b2BodyDef groundBodyDef;
        groundBodyDef.position.Set(0.0f, -10.0f);

        // Call the body factory which allocates memory for the ground body
        // from a pool and creates the ground box shape (also from a pool).
        // The body is also added to the world.
        b2Body* groundBody = world.CreateBody(&groundBodyDef);

        // Define the ground box shape.
        b2PolygonShape groundBox;

        // The extents are the half-widths of the box.
        groundBox.SetAsBox(50.0f, 10.0f);

        // Add the ground fixture to the ground body.
        groundBody->CreateFixture(&groundBox, 0.0f);

        // Define the dynamic body. We set its position and call the body
            factory.
        b2BodyDef bodyDef;
        bodyDef.type = b2_dynamicBody;
```

```
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);

// Define another box shape for our dynamic body.
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);

// Define the dynamic body fixture.
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;

// Set the box density to be non-zero, so it will be dynamic.
fixtureDef.density = 1.0f;

// Override the default friction.
fixtureDef.friction = 0.3f;

// Add the shape to the body.
body->CreateFixture(&fixtureDef);

// Prepare for simulation. Typically we use a time step of 1/60 of a
// second (60Hz) and 10 iterations. This provides a high quality
    simulation
// in most game scenarios.
float timeStep = 1.0f / 60.0f;
int32 velocityIterations = 6;
int32 positionIterations = 2;

// This is our little game loop.
for (int32 i = 0; i < 60; ++i)
{
       // Instruct the world to perform a single step of simulation.
       // It is generally best to keep the time step and iterations
           fixed.
       world.Step(timeStep, velocityIterations, positionIterations);

       // Now print the position and angle of the body.
       b2Vec2 position = body->GetPosition();
       float angle = body->GetAngle();

       printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
}

// When the world destructor is called, all bodies and joints are
    freed. This can
// create orphaned pointers, so be careful about your world
    management.
```

```
        return 0;
    }
```

**C++ Code 25:** *main.cpp "Hello Box2D"*

To compile it type:
**g++ main.cpp -o result -lbox2d**
**./result**
(-lbox2d is linking to shared library of Box2D' libbox2d.so)

**Figure 5.1:** *The running HelloBox2D shows decreasing height from 4 to 1. The computation is the core of Box2D, to shows the visualization like at the testbed, by default Box2D uses imgui, but you can also use your own GUI besides imgui (file for this example in folder:* ***DFSimulatorC/Source Codes/C++/ch5-box2d-helloworld****).*

## ii. Build Box2D Testbed

For simplicity instead of putting all the codes for Box2D testbed, I advise you to copy from my repository **https://github.com/glanzkaiser/DFSimulatorC/Source Codes/C++/ch5-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed** You can configure the **CMakeLists.txt** so you can only generate the example you are concern about, less time compiling. The C++ source codes for every examples are located in **../tests/**, how to modify them, you need to comprehend C++ around medium-level at least, and you can read the manual here:
**https://box2d.org/documentation/**

**Figure 5.2:** *The testbed of Box2D, one of the beauty of imgui is that it can offers parameters changing and a lot of modification on the scene (files for this example in folder:* ***DFSimulatorC/Source Codes/C++/ch5-box2d-testbed****).*

## iii. Know-How in Box2D

- There are 3 body types in Box2D: Static, Kinematic, and Dynamic
- Box2D support motion and collisions with
  1. Body class provides the motion.
  2. Fixture and Shape classes are for collisions.
- Properties in class Body: Position, Linear Velocity, Angular Velocity, Body Type. In the header file (**../box2d/include/box2d/b2_body.h**) you can see all the body definitions under **struct B2_API b2BodyDef...**, you can edit this and recompile to recreate the library to adjust to your modification.

```
struct B2_API b2BodyDef
{
        /// This constructor sets the body definition default
            values.
        b2BodyDef()
        {
                position.Set(0.0f, 0.0f);
                angle = 0.0f;
                linearVelocity.Set(0.0f, 0.0f);
                angularVelocity = 0.0f;
                linearDamping = 0.0f;
                angularDamping = 0.0f;
                allowSleep = true;
                awake = true;
                fixedRotation = false;
                bullet = false;
                type = b2_staticBody;
                enabled = true;
                gravityScale = 1.0f;
        }
        ...
}
```

- In Box2D we have impulse as force times 1 second.
- Fixtures are used to describe the size, shape, and material properties of an object in the physics scene. When two bodies collide, their fixtures are used to decide how they will react. Every fixture has a shape which is used to check for collisions with other fixtures as it moves around the scene. Shape can be a circle or a polygon.
- Motion depends on Force, current velocity, and mass

$$\begin{aligned}\triangle s &= v\triangle t \\ &= v_0\triangle t + \frac{1}{2}a\left(\triangle t\right)^2 \\ &= v_0\triangle t + \frac{1}{2}\left(\frac{F}{m}\right)\left(\triangle t\right)^2\end{aligned}$$

The mass comes from the Fixture class. Fixture gives volume to the body. The density of a fixture multiplied by it's area becomes the mass of that fixture. To set a density for a fixture here is one example:

```
b2FixtureDef myFixtureDef;
...
myFixtureDef.density = 1;
```

- Four ways to move a Dynamic body:
  1. Forces: **applyForce** (linear), **applyTorque** (angular).
     For joints, complex shapes. Hard to control.
  2. Impulses: **applyLinearImpulse** (linear), **applyAngularImpulse** (angular).
     Great for joints, complex shapes. Extremely hard to control.
  3. Velocity: **setLinearVelocity** (linear), **setAngularVelocity** (angular)
     Very easy to control, not for joints, complex shapes.
  4. Translation: **setTransform**
- Shape stores the object geometry (either boxes, circles or polygons, it must be convex). Shape also stores object density, which mass is area times density. The higher the density, the higher the mass.

```
bodydef = newBodyDef();
bodydef.type = type;
bodydef.position.set(position);
bodydef.angle = angle;

body1 = world.createBody(bodydef);

bodydef.position.set(position2);
body2 = world,createBody(bodydef);

shape1 = new PolygonShape();
shape2 = new PolygonShape();
shape1.set(verts1);
shape2.set(verts2);

fixdef = new FixtureDef();
fixdef.density = density;

fixdef.shape = shape1;
fixture1 = body1.createFixture(fixdef);
fixdef.shape = shape2;
fixture2 = body1.createFixture(fixdef);
```

- Size in Box2D:
  1. 1 Box2D unit = 1 meter
  2. 1 density = 1 $kg/m^2$
- Fixture has only one body, while bodies have many fixtures.
- Properties in Box2D:
  1. Friction: stickiness
  2. Restitution: bounciness

Both value should be within 0 to 1.

- For custom collisions you can use **ContactListeners**, with two primary methods in interface:vvvvvv
  1. **beginContact**: When objects first collide
  2. **endContact**: When objects no longer collide

  It can be used for color changing in Box2D.
- Collision filtering can be used to define what can collide with certain fixture.
- Joints connect bodies, and they are affected by fixtures. Joints must use force or impulse, manual velocity might violate constraints.
- Box2D tends to allocate a large number of small objects (around 50-300 bytes). Using the system heap through malloc or new for small objects is inefficient and can cause fragmentation. Box2D's solution is to use a small object allocator (SOA). Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. However, you do have to allocate a **b2World** on your own.

## II. Bullet

Bullet Physics is a professional open source collision detection, rigid body and soft body dynamics library. The library is free for commercial use under the ZLib license.

You can read more about Bullet here: **bulletphysics.org**

### i. Install Bullet Library and Include Files / Headers

To get the library, open the terminal and type:
**git clone https://github.com/bulletphysics/bullet3.git**

Enter the directory, then type:
**mkdir build**
**cd build**
**ccmake ..**
**make -j4** (build with 4 cores, faster than just **make**)

If you want to use manual typing, from the bullet directory, type:

```
mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/opt/hamzstlib/Physics/bulletinstall \
$ —DBUILD_SHARED_LIBS=ON .. &&
$ make
```

after finish compiling then type:
**make install**
It will install Bullet' libraries and include files at the install prefix place.

To get more information on the installation directory, where Bullet installs the libraries and search for the include files go to **../lib/cmake/bullet/** and read the **BulletConfig.cmake**, make sure it is not clashing with our own **LIBRARY_PATH**. You can see below on create an example for

Bullet, how I set the path the same as the **BulletConfig.cmake** to find the libraries and include files for Bullet.

## ii. Know-How in Bullet

- Bullet can do discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives.
- The main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform for all the objects.
- The most important data structures and computation stages for rigid body physics: The entire

**Figure 5.3:** *The pipeline for Bullet physics is executed from left to right, starting by applying gravity, and ending by position integration, updating the world transform.*

  physics pipeline computation and its data structures are represented in Bullet by a dynamics world. When performing **stepSimulation** on the dynamics world, all the above stages are executed. The default dynamics world implementation is the **btDiscreteDynamicsWorld**.
- How to create a **btDiscreteDynamicsWorld, btCollisionShape, btMotionState**, and **btRigidBody**. Each frame call the **stepSimulation** on the dynamics world, and synchronize the world transform for your graphics object, the requirements:
  1. Put **#include "btBulletDynamicsCommon.h"** in your source file.
  2. Required include path: **Bullet/src** folder.
  3. Required libraries: **BulletDynamics, BulletCollision, LinearMath**.
- The derivations from **btDynamicsWorld** are **btDiscreteDynamicsWorld** and **btSoftRigidDynamicsWorld** will provide a high level interface that manages your physics objects and constraints. It also implements the update of all objects each frame.
- To construct a **btRigidBody** or **btCollisionObject** you need to provide:
  1. Mass, positive for dynamics moving objects and 0 for static objects
  2. Collision shape (Box, Sphere, Cone, Convex Hull, Triangle Mesh)
  3. Material properties (friction and restitution)

  Then update each frame with **stepSimulation**, call it on the dynamics world. The **btDiscreteDynamicsWorld** automatically takes into account variable timestep by performing interpolation instead of simulation for small timesteps. It uses an internal fixed timestep of 60 Hertz. The **stepSimulation** will perform collision detection and physics simulation. It updates the world transform for active objects by calling **btMotionState**'s **setWorldTransform**.

### iii. Create an Example with Bullet

First, in order to link the library correctly and find the include files for Bullet, you will need to modify the export file by typing at terminal in :
**cd**
**vim export**

```
export prefix="/usr"
export hamzstlib="/opt/hamzstlib"
export physics="$hamzstlib/Physics"

# For library (.so, .a)
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib:$physics/bulletinstall/lib"

export PKG_CONFIG_PATH="$physics/bulletinstall/lib/pkgconfig:$hamzstlib/lib
    /pkgconfig:"
```

The **bulletinstall** is the folder containing the **include, lib, output** directories after compiling Bullet.

Thus, it will be able to find the Bullet' libraries that have been built / compiled in GFreya OS. We are going to use the libraries **-lBulletDynamics -lBulletCollision -lLinearMath**.

Now, in a working directory (assumed still empty) create a file as usual, type:
**vim main.cpp**

**C++ Code 26:** *main.cpp "Bullet Example"*

## III. ReactPhysics3D

ReactPhysics3D is an open source C++ physics engine library that can be used in 3D simulations and games. The library is released under the Zlib license. ReactPhysics3D is using OpenGL.

### i. Install ReactPhysics3D Library and Include Files / Headers

To get the library, open the terminal and type:
**git clone https://github.com/DanielChappuis/reactphysics3d.git**

Now to build it, enter the downloaded repository, then type:
**mkdir build**
**cd build**
**ccmake ..**

Choose to build the library, then install it at the **/usr/lib**, so it can be used later on.

## ii. Know-How in ReactPhysics3D

- The first thing you need to do when you want to use ReactPhysics3D is to instantiate the **PhysicsCommon** class. This class is used as a factory to instantiate one or multiple physics worlds and other objects. It is also responsible for logging and memory management. To create a physics world:
  **PhysicsWorld* world = physicsCommon.createPhysicsWorld();**
  This method will return a pointer to the physics world that has been created. How to create the world settings:

```
PhysicsWorld::WorldSettings settings;
settings.defaultVelocitySolverNbIterations = 20;
settings.isSleepEnabled = false;
settings.gravity = Vector(0,-9.81,0)

PhysicsWorld* world = physicsCommon.createPhysicsWorld();
```

  this will create the physics world with your own settings.
- You probably only need one physics world with multiple objects.
- Simulating many bodies is cost expensive, thus sleeping technique is used to deactivate resting bodies. A body is put to sleep when its' linear and angular velocity stay under a given velocity threshold for certain amount of time.
- The base memory allocations in ReactPhysics3D are done using the **std::malloc()** and **std::free()** methods.
- Some methods that you can use:
  1. **testOverlap()**
     This group of methods can be used to test whether the colliders of two bodies overlap or not
  2. **testCollision()**
     This group of methods will give you the collision information (contact points, normals, etc) for colliding bodies.
  3. **testPointInside()**
     This method will tell you if a 3D point is inside a given CollisionBody, RigidBody, or Collider.
- Types of a Rigid Body:
  1. **Static body**
     A static body has infinite mass, zero velocity but its position can be changed manually. A static body does not collide with other static or kinematic bodies. You can use this as a floor or ground.
  2. **Kinematic body**
     A kinematic body has infinite mass, its' velocity can be changed manually and its' position computed by the physics engine. A kinematic body does not collide with other static or kinematic bodies.
  3. **Dynamic body**
     A dynamic body has non-zero mass, with velocity determined by forces and its' position is determined by the physics engine. A dynamic body can collide with other dynamic, static or kinematic bodies. You can use this as a rock or apple or any kind of object in the physics world.

You can create a rigid body then change the type of RigidBody with:

```
Vector3 position(0.0, 3.0, 0.0);
Quaternion orientation = Quaternion::identity();
Transform transform(position, orientation);

RigidBody* body = world -> createRigidBody(transform);
body->setType(BodyType::KINEMATIC);
```

- Damping is the effect of reducing the velocity of the rigid body during the simulation effects like air friction for instance. By default, no damping is applied. Without angular damping a pendulum will never comes to rest. You can choose to damp the linear or/and the angular velocity of a rigid body. You need to use:
  **RigidBody::setLinearDamping()**
  **RigidBody::setAngularDamping()**
  the damping value has to be positive. The value of 0 means no damping at all.

# Chapter 6

# DFSimulatorC++ 0: C+++ = C++ + Eigen + Gnuplot + SymbolicC++ and All That

*"Bring that book, the one that is written your name on it, replacing the real author name (on College Geometry book)" - **Sentinel***

I put this chapter before the real physics simulation, because after we do simulation and want to analyze more, we need to be able to plot the data, clean the data, manipulate the data if there are lots of NaN or outliers, I think Gnuplot that can be called directly from C++ offers a great promise in the future of scientific computing.

In this chapter, we also want to introduce a library for C++, SymbolicC++ [7] that uses C++ and object-oriented programming to develop a computer algebra system. SymbolicC++ is a general purpose computer algebra system written in the programming language C++. It is free software released under the terms of the GNU General Public License. SymbolicC++ is used by including a C++ header file or by linking against a library.

Object-oriented programming is an approach to software design that is based on classes rather than procedures. This approach maximizes modularity and information hiding. Object-oriented design provides many advantages. For example, it combines both the data and the functions that operate on that data into a single unit. Such a unit (abstract data type) is called a class. This library can be used for symbolic computations in mathematics, like determining a derivative or integral of a function. If there is SymPy for Python, then SymbolicC++ is for C++.

In the first version of SymbolicC++ the main data type for symbolic computation was the Sum class. The list of available classes included:

- Verylong : An unbounded integer implementation
- Rational : A template class for rational numbers
- Quaternion : A template class for quaternions
- Derive : A template class for automatic differentiation
- Vector : A template class for vectors (see vector space)
- Matrix : A template class for matrices (see matrix (mathematics))

- Sum : A template class for symbolic expressions

The second version of SymbolicC++ featured new classes such as the Polynomial class and initial support for simple integration. The third version features a complete rewrite of SymbolicC++ and was released in 2008. This version encapsulates all symbolic expressions in the Symbolic class.

The greatness of SymbolicC++ combined with Gnuplot and C++ is that all that I have done with Julia to create Riemann sum plot, plot integral of a function, plot derivative of a function, can also be done in C++, more codes but more rewarding, since it is very fast, and not taking too many storage, I compare this with my Julia installation, I add a lot of packages while in Qemu for GFreya OS, do Rsync to move the downloaded packages into GFreya OS in dual boot (that can't be connected to internet so people using GFreya OS can focus more on either coding or writing books, not finding something to buy in internet), turns out the **./julia** folder is 14 GB with packages that I have added including: **SymPy, PrettyTables, OrdinaryDiffEq, Optim, CairoMakie, Makie,** and more. Now let see **gnuplot_i.hpp**, a header file that is needed to plot with Gnuplot with C++ code is only 58 kb, and all the SymbolicC++ headers are 294 kb only.

Besides, SymbolicC++, there are other libraries that can fill the gap from SymbolicC++, like **Ginac** and **Armadillo** both are linear algebra library that can be used interchangeably depends on our problems.

**Figure 6.1:** *The size of **.julia** in GFreyaOS is 14 GB as of November 16, 2023.*

All the codes related to this chapter can be obtained in the repository:
**https://github.com/glanzkaiser/DFSimulatorC**
(Find them under this directory **../DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++**)

I have put the libraries source codes in the repository as well, but you can also find it yourself in internet:

- To obtain **gnuplot_i.hpp**
  You need to download the interfacing code from here: **https://code.google.com/archive/p/gnuplot-cpp/**.

- To obtain **gnuplot-iostream**
  You can get it here: **https://github.com/dstahlke/gnuplot-iostream**.

  For more advanced usage of Gnuplot you need to install Boost library. Always remember **gnuplot-iostream** relies on the Boost library, we will always link to Boost library when using this header. Boost is a very common library, but it doesn't come together with the C++ compiler, so make sure it is properly installed. Boost package compilation and installation is covered in BLFS book, see Linux From Scratch book. Fortunately, Boost is installed in GFreya OS used in this book, so we are all covered.
- To obtain **SymbolicC++** headers
  You can find it at the official website: **http://issc.uj.ac.za** and to download it directly you can

go here: **https://symboliccpp.sourceforge.net/**.

(Users of SymbolicC++ with GCC on 64-bit may need to use the -fno-elide-constructors flag.)

- To obtain **Eigen** you can download it from this link:
  **https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz**

- To obtain **Ginac** you can download it from this link:
  **https://www.ginac.de/ginac.git/**
- To obtain **Armadillo** you can download it from this link:
  **https://sourceforge.net/projects/arma/files/armadillo-12.6.7.tar.xz/download**

# I. Gnuplot

We start with Gnuplot waltz with C++, since it only takes a simple header to be included in the working directory to plot any kind of function even in 3 dimension.

## i. Plot a Slope with Gnuplot from C++

First, we will learn how to plot a slope $y = x$, first create the file name **main.cpp**

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-
    communication with Gnuplot

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined
    (__TOS_WIN__)
#include <conio.h> //for getch(), needed in wait_for_key()
#include <windows.h> //for Sleep()
void sleep(int i) { Sleep(i*1000); }
#endif

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{

       cout << "*** example of gnuplot control through C++ ***" <<
           endl << endl;

       //
       // Using the GnuplotException class
       //
```

```
        try
        {
                Gnuplot g1("lines");

                //
                // Slopes
                //
                cout << "*** plotting slopes" << endl;
                g1.set_title("Slope Plotting");

                cout << "y = x" << endl;
                g1.plot_slope(1.0,0.0,"y=x");
                g1.showonscreen(); // window output

                wait_for_key();

        }
        catch (GnuplotException ge)
        {
                cout << ge.what() << endl;
        }


        cout << endl << "*** end of gnuplot example" << endl;

        return 0;
}



void wait_for_key ()
{
        #if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) ||
             defined(__TOS_WIN__) // every keypress registered, also
            arrow keys
        cout << endl << "Press any key to continue..." << endl;

        FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
        _getch();
        #elif defined(unix) || defined(__unix) || defined(__unix__) ||
            defined(__APPLE__)
        cout << endl << "Press ENTER to continue..." << endl;

        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();
        #endif
        return;
```

```
        }
```

**C++ Code 27:** *main.cpp "Slope plotting Gnuplot C++"*

Now copy **gnuplot_i.hpp** from the repository under the include folder, you will find it here: **../Source Codes/include/**

```
#ifndef _GNUPLOT_PIPES_H_
#define _GNUPLOT_PIPES_H_


#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <sstream> // for std::ostringstream
#include <stdexcept>
#include <cstdio>
#include <cstdlib> // for getenv()
#include <list> // for std::list


#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(
    __TOS_WIN__)
//defined for 32 and 64-bit environments
#include <io.h> // for _access(), _mktemp()
#define GP_MAX_TMP_FILES 27 // 27 temporary files it's Microsoft restriction
#elif defined(unix) || defined(__unix) || defined(__unix__) || defined(
    __APPLE__)
//all UNIX-like OSs (Linux, *BSD, MacOSX, Solaris, ...)
#include <unistd.h> // for access(), mkstemp()
#define GP_MAX_TMP_FILES 64
#else
#error unsupported or unknown operating system
#endif

//declare classes in global namespace


class GnuplotException : public std::runtime_error
{
      public:
      GnuplotException(const std::string &msg) : std::runtime_error(msg){}
};

...
...

void Gnuplot::remove_tmpfiles(){
      if ((tmpfile_list).size() > 0)
```

```
        {
                for (unsigned int i = 0; i < tmpfile_list.size(); i++)
                remove( tmpfile_list[i].c_str() );

                Gnuplot::tmpfile_num —= tmpfile_list.size();
        }
}
#endif
```

**C++ Code 28:** *gnuplot_i.hpp*

I only put the head and tail of the source code since it took 50 pages of this book just to show the full codes of **gnuplot_i.hpp**, those who wrote the source codes is amazing.

Put **gnuplot_i.hpp** and **main.cpp** under one same folder / working directory then type:
**g++ -o main main.cpp**
**./main**

**Figure 6.2:** *The plot of a slope $y = x$ with Gnuplot inside C++ you can move the view from the GUI with arrow key press on your keyboard (the plot code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/Slope Plotting/main.cpp****).*

I get two warnings but can still run the example. If you want to use Makefile create a Makefile in the current working directory:

```
CFLAGS = —ggdb
DEFINES = —DDEBUGGA
INCLUDES =
LIBS = —lstdc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) —c $(CFLAGS) $(DEFINES) $(INCLUDES) $<

all:: main
```

```
gnuplot_i.o: gnuplot_i.hpp
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

**C++ Code 29:** *Makefile "Gnuplot Slope Makefile"*

now you can type:
**make**
**./make**

Another example of the Makefile that set the C++ version being used to be version 2011 is like this

```
CFLAGS = -ggdb
CXXFLAGS= -g -std=c++11 -Wall -pedantic
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<

all:: main

gnuplot_i.o: gnuplot_i.hpp
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

**C++ Code 30:** *Makefile "Gnuplot Slope Makefile"*

In 2024, C++23 (C++ version 2023) features are available since GCC 11 (the compiler). To enable C++23 support, add the command-line parameter **-std=c++2b** to your **g++** command line. To enable GNU extensions in addition to C++23 features, add **-std=gnu++2b**.

That, is another way to compile besides **g++ -o main main.cpp**, sometimes for big projects Makefile is more preferred.

## ii. Plot a 2D Function with Gnuplot from C++

We will be able to learn how to plot function, not only trigonometric, but also transcendental functions (exponential and logarithm).First create the file name **main.cpp** in a new working directory

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
    with Gnuplot

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(
    __TOS_WIN__)
#include <conio.h> //for getch(), needed in wait_for_key()
#include <windows.h> //for Sleep()
void sleep(int i) { Sleep(i*1000); }
#endif

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{

       cout << "*** example of gnuplot control through C++ ***" << endl <<
           endl;

       try
       {
              Gnuplot g1("lines");

              cout << "*** plotting slopes" << endl;
              g1.set_title("Function Plotting");

              //cout << "y = sin(x)" << endl;
              //g1.plot_equation("sin(x)","sine");

              //cout << "y = log(x)" << endl;
              //g1.plot_equation("log(x)","logarithm");

              //cout << "y = exp(x) * tan(2*x)" << endl;
              //g1.plot_equation("exp(x)*tan(2*x)","exponential and
                  trigonometric product");

              cout << "y = sin(x) * cos(2*x)" << endl;
```

```
                g1.plot_equation("sin(x)*cos(2*x)","sine product");

                g1.showonscreen(); // window output

                wait_for_key();

        }
        catch (GnuplotException ge)
        {
                cout << ge.what() << endl;
        }

        cout << endl << "*** end of gnuplot example" << endl;

        return 0;
}

void wait_for_key ()
{
        #if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined
            (__TOS_WIN__) // every keypress registered, also arrow keys
        cout << endl << "Press any key to continue..." << endl;

        FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
        _getch();
        #elif defined(unix) || defined(__unix) || defined(__unix__) || defined(
            __APPLE__)
        cout << endl << "Press ENTER to continue..." << endl;

        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();
        #endif
        return;
}
```

**C++ Code 31:** *main.cpp "Function plotting Gnuplot C++"*

Put **gnuplot_i.hpp** and **main.cpp** under one same folder / working directory then type:
**g++ -o main main.cpp**
**./main**

There are four choices here, and you can uncomment three of them and only plot 1 out of the three that you want to plot:

```
//cout << "y = sin(x)" << endl;
//g1.plot_equation("sin(x)","sine");

//cout << "y = log(x)" << endl;
//g1.plot_equation("log(x)","logarithm");
```

**Figure 6.3:** *The plot of a function* $\sin(x)\cos(2x)$ *with Gnuplot inside C++ (the plot code can be located in:* ***DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/Function Plotting/main.cpp****).*

```
//cout << "y = exp(x) * tan(2*x)" << endl;
//g1.plot_equation("exp(x)*tan(2*x)","exponential and trigonometric product
    ");
cout << "y = sin(x) * cos(2*x)" << endl;
g1.plot_equation("sin(x)*cos(2*x)","sine product");
```

The makefile from previous section is also working on this code.

## iii. Plot a 3D Function with Gnuplot from C++

After some warming up, now how about plotting a surface plot? Building Information Modeling like AutoDesk and AutoCad are all in 3 dimension, so the user able to simulate a building or highway and to apply a lot of tests for the standard quality determination. This is the very first step before one day maybe we can create something better than AutoDesk and AutoCad for all engineering and science branches.

From this section there are two important things you need to know:

1. Whenever you want to call for Gnuplot, always remember that you need to have **gnuplot_i.hpp** in your working directory, along with the C++ source codes.

2. the Makefile from the last two sections can be used in this section and below related to calling Gnuplot from C++.

Now, in a new working directory, create a **main.cpp**.

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
    with Gnuplot

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array
```

```
void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{

       cout << "*** example of gnuplot control through C++ ***" << endl <<
           endl;

       try
       {
              Gnuplot g1("lines");
              cout << "window: splot with hidden3d" << endl;
              g1.set_isosamples(25).set_hidden3d();
              g1.plot_equation3d("x**2+y*y");

              wait_for_key();

       }
       catch (GnuplotException ge)
       {
              cout << ge.what() << endl;
       }


       cout << endl << "*** end of gnuplot example" << endl;

       return 0;
}

void wait_for_key ()
{
       cout << endl << "Press ENTER to continue..." << endl;

       std::cin.clear();
       std::cin.ignore(std::cin.rdbuf()->in_avail());
       std::cin.get();
       return;
}
```

**C++ Code 32:** *main.cpp "3D Surface plotting Gnuplot C++"*

From the working directory type:
**g++ -o main main.cpp**
**./main**

Compared to the last two source codes, since this book is using GFreya OS, thus I delete the

**Figure 6.4:** *The 3-dimensional plot of a surface $f(x,y) = x^2y^2$ with Gnuplot inside C++ (the plot code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/3D Surface Plotting/main.cpp****).*

commands that are used for Windows OS to save more space, thus if you are using Microsoft Windows, see the last two source codes and copy what I have deleted into your source codes.

## iv. Plot a 3D Curve / Line from User Defined Points with Gnuplot from C++

Now, this is more fun, since in this section, we learn if we have an input points of $x, y, z$, then we can plot a line or curve out of them.

In a new working directory, create **main.cpp** file.

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
    with Gnuplot

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{

        cout << "*** example of gnuplot control through C++ ***" << endl <<
            endl;

        try
        {
                std::vector<double> x, y, z;
```

```
                    for (int i = 0; i < NPOINTS; i++) // fill double arrays x, y,
                         z
                    {
                           x.push_back((double)i); // x[i] = i
                           y.push_back((double)i * (double)i); // y[i] = i^2
                           z.push_back( x[i]*y[i] ); // z[i] = x[i]*y[i] = i^3
                    }
                    Gnuplot g1("lines");
                    cout << endl << endl << "*** user-defined lists of points (x
                        ,y,z)" << endl;
                    g1.unset_grid();
                    g1.plot_xyz(x,y,z,"user-defined points 3d");

                    wait_for_key();

             }
             catch (GnuplotException ge)
             {
                    cout << ge.what() << endl;
             }

             cout << endl << "*** end of gnuplot example" << endl;

             return 0;
      }

      void wait_for_key ()
      {
             cout << endl << "Press ENTER to continue..." << endl;

             std::cin.clear();
             std::cin.ignore(std::cin.rdbuf()->in_avail());
             std::cin.get();
             return;
      }
```

**C++ Code 33:** *main.cpp "3D Curve plotting Gnuplot C++"*

From the working directory type:
**g++ -o main main.cpp**
**./main**

The **NPOINTS** defined at the beginning is the number of points that are going to be plotted or the length of array, the higher the number, the smoother the curve.

If you want to plot the points only instead of curve connecting the curve, you can replace

```
      cout << endl << endl << "*** user-defined lists of points (x,y,z)" << endl
          ;
      g1.unset_grid();
```

**Figure 6.5:** *The 3-dimensional plot of a line connecting 50 points defined by user with Gnuplot inside C++ (the plot code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/3D Line from Points/main.cpp****).*

```
g1.plot_xyz(x,y,z,"user-defined points 3d");
```

with

```
g1.set_style("points").plot_xy(x,y,"user-defined points 2d");
cout << endl << endl << "*** user-defined lists of points (x,y,z)" << endl
    ;
g1.unset_grid();
g1.plot_xyz(x,y,z,"user-defined points 3d");
```

**Figure 6.6:** *The 3-dimensional plot of 50 points defined by user with Gnuplot inside C++ (the plot code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/3D Plot xyz Points/main.cpp****).*

There are a lot of other Gnuplot examples that can be called from C++ source codes in the repository in **../DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/**, you can check it out yourselves.

## v. Plot Data From Textfile in 2D with Gnuplot

Now, let the fun begins. When we play with Box2D and we can print the output of the velocity of the object, or the kinetic energy, or other parameter, it comes in textfile (.txt), there are two ways to plot the result from txt:

1. By calling gnuplot directly from terminal at the working directory containing the textfile
2. By using C++ code to plot with Gnuplot through C++

We are going to examine the second option, it is longer, but compared to the possibilities it opens, it has tremendous applications. First, we can combine with SymbolicC++ and thus interpolate the graph with Fourier Series or Cubic Spline after plotting the data, we can also find the derivative of the interpolated function. Option 1 is the easiest, it only takes 1 line to plot the velocity of an object, but then to do deeper analysis on the variable/s related, that will comes to differential equation or partial differential equation, like solving double pendulum problem, option 2 is way to go for that.

First, in a new working directory, create a textfile **matrix.txt**

```
1.1
2.4
3.5
1.4
5.5
6.7
3.2
4.7
7.5
-9.7
-20
```

**C++ Code 34:** *matrix.txt "2D Plot from Textfile with Gnuplot through C++"*

It is only an example, you can print your own output from Box2D simulation or another Physics Engine / simulation.

Then, create the C++ code **main.cpp**.

```
// Use fstream to read from a file, and ofstream to write to a file.
// g++ main.cpp

#include <iostream>
#include <fstream>
#include <string>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
    with Gnuplot

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 11 // length of array

const int numRows=11;
```

```
const int numCols=1;

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;
using namespace std;

int main(int argc, char* argv[])
{
       cout << "*** Example of Gnuplot plot data from txt through C++ ***"
           << endl << endl;
       std::fstream in("matrix.txt");
       float tiles[numRows][numCols];
       std::vector<double> y;
       for (int i = 0; i < numRows; i++)
       {
              for (int j = 0; j < numCols; j++)
              {
                     in >>tiles[i][j]; // get data row i column j from the
                         textfile
                     cout <<tiles[i][j] << ' '<< endl;
              }
              y.push_back((float)tiles[i][0]); // Thanks for telling me
                  this trick Freya.. I owe you a lot.
       }

       try
       {
              std::vector<double> x;
              for (int i = 0; i < NPOINTS; i++) // fill double array x
              {
                     x.push_back((double)i); // x[i] = i
              }
              Gnuplot g1("lines");
              cout << endl << endl << "*** user-defined lists of points (x
                  ,y)" << endl;
              g1.set_grid();
              g1.set_style("linespoints").plot_xy(x,y,"(x,y)");

              wait_for_key();

       }
       catch (GnuplotException ge)
       {
              cout << ge.what() << endl;
       }
```

```
        cout << endl << "*** end of gnuplot example" << endl;

        return 0;
}

void wait_for_key ()
{
        cout << endl << "Press ENTER to continue..." << endl;

        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();
        return;
}
```

**C++ Code 35:** *main.cpp "2D Plot from Textfile with Gnuplot through C++"*

Compile it by typing from terminal:
**g++ -o result main.cpp**
**./result**

You can also compile by using **Makefile**, create one in the current working directory.

```
CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<

all:: main

gnuplot_i.o: gnuplot_i.hpp
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

**C++ Code 36:** *Makefile "2D Plot from Textfile with Gnuplot through C++"*

Compile with Makefile by typing:
**make**
**./main**

**Figure 6.7:** *The plot from "matrix.txt" with Gnuplot through C++ (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/2D Plot from Textfile****).*

## vi. Plot Data and Fitting Curve From Textfile in 2D with "gnuplot-iostream.h"

In the previous sections, all examples are using header of "gnuplot_i.hpp", now we are going to use "gnuplot-iostream.h" to plot data from textfile input and then plot a fitting curve (interpolation method) of the form

$$f(x) = a + bx + cx^2$$

with $a, b$, and $c$ are constants that can be determined by the fitting formula, it is built in from Gnuplot.

First, in a new working directory put "gnuplot-iostream.h" there. Then create the main C++ source code, **main.cpp**.

```
#include <fstream>
#include <vector>
#include <cmath>

#include "gnuplot-iostream.h"

const int numRows=11;
const int numCols=1;

using std::cout;
using std::endl;
using namespace std;

int main() {
       std::fstream in("matrix.txt");
       float tiles[numRows][numCols];
       std::vector<std::pair<double, double>> xy_pts;
```

```
        for (int i = 0; i < numRows; i++)
        {
            for (int j = 0; j < numCols; j++)
            {
                in >>tiles[i][j]; // get data row i column j from the
                    textfile
                cout <<tiles[i][j] << ' '<< endl;
            }
            xy_pts.emplace_back(i, tiles[i][0]); // Thanks a lot
                Beautiful Goddess, Freya!!!
        }
        Gnuplot gp;

        gp << "set xrange [0:12]\n";
        gp << "f(x) = a + b*x + c*x*x\n";
        gp << "fit f(x) '-' via a,b,c\n";
        gp.send1d(xy_pts);
        gp << "plot '-' with points title 'input', f(x) with lines title '
            fit'\n";
        gp.send1d(xy_pts);
    }
```

**C++ Code 37:** *main.cpp "2D Plot and Fit Curve from Textfile with Gnuplot through C++"*

Now, you can create again the **matrix.txt** that containing 11 points of data (11 $\times$ 1 matrix or column vector with size of 11), read previous section for the corresponding textfile.

You can compile it by typing:
**g++ -o main main.cpp -lboost_iostreams -lboost_system -lboost_filesystem**
**./main**

Another alternative is by creating Makefile, so you can just type:
**make**
**./main**

Clean the object file with **make clean**, but you have to manually remove the log by typing:
**rm -rf fit.log**

```
CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES = gnuplot-iostream.h
LIBS = -lstdc++ -lboost_iostreams -lboost_system -lboost_filesystem
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<
```

```
all:: main

gnuplot_i.o:
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

**C++ Code 38:** *Makefile "2D Plot and Fit Curve from Textfile with Gnuplot through C++"*

**Figure 6.8:** *The plot and fitting curve from "matrix.txt" with Gnuplot through C++ with "gnuplot-iostream.h" and Boost (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/2D Plot and Fit Curve from Textfile****).*

```
*******************************************************************************
Fri Nov 17 14:01:50 2023


FIT:    data read from '-'
        format = z
        x range restricted to [0.00000 : 12.0000]
        #datapoints = 11
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x) = a + b*x + c*x*x
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a           b           c
   0 3.6888989987e+04   0.00e+00  2.79e+01    1.000000e+00   1.000000e+00   1.000000e+00
   5 1.8786273982e+02  -7.62e-06  2.79e-04   -2.267133e+00   4.812028e+00  -6.062937e-01

After 5 iterations the fit converged.
final sum of squares of residuals : 187.863
rel. change during last iteration : -7.61745e-11

degrees of freedom    (FIT_NDF)                        : 8
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 4.84591
variance of residuals (reduced chisquare) = WSSR/ndf    : 23.4828

Final set of parameters            Asymptotic Standard Error
=======================            =========================
a               = -2.26713         +/- 3.692       (162.8%)
b               = 4.81203          +/- 1.718       (35.7%)
c               = -0.606294        +/- 0.1654      (27.29%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.816  1.000
c               0.672 -0.963  1.000
```

**Figure 6.9:** *The fit.log file containing the details for the function and parameters used for curve fitting.*

# II. SymbolicC++ Library

## i. Compile and Install SymbolicC++ Library

Download SymbolicC++ that can be used to be compiled into library (this file name is **SymbolicC++3-3.35-ac.tar.gz**, you can type from terminal:
**wget http://sourceforge.net/projects/symboliccpp/files/SymbolicC%2B%2B%203.35/SymbolicC%2B%2B3-3.35-ac.tar.gz/download**
(to create the library)

or if you only want the headers file, type:
**wget http://sourceforge.net/projects/symboliccpp/files/SymbolicC%2B%2B%203.35/SymbolicC%2B%2B3-3.35.tar.gz/download**

all the files are also available in my repository under the directory **../DFSimulatorC/Source Codes/Libraries/**, we are going to proceed by creating the library, thus extract it by typing:
**tar -xvf SymbolicC++3-3.35-ac.tar.gz**
**cd SymbolicC++3-3.35**
**./configure –prefix=/usr**
**make**
**make install**

The include files installed from the method above produce error while compiling. The solution is (if you are using GFreya OS) make sure in **/usr/include/symbolic** you replace all of them from the include in my repository **DFSimulatorC/Source Codes/include/symbolic**, this include in my repository is obtained from the downloaded SymbolicC++ that you can find here:
**DFSimulatorC/Source Codes/Libraries/SymbolicC++3-3.35.zip**.
or from:
**http://sourceforge.net/projects/symboliccpp/files/SymbolicC%2B%2B%203.35/SymbolicC%2B%2B3-3.35.tar.gz/download**

```
xterm
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,-rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
/bin/mkdir -p /usr/share/doc
/bin/install -c -m 644 README /usr/share/doc/README.SymbolicC++
/bin/install -c -m 644 doc/introsymb.pdf /usr/share/doc/SymbolicC++.pdf
test -z "/usr/include" || /bin/mkdir -p "/usr/include"
 /bin/install -c -m 644 include/array.h include/cloning.h include/derive.h inclu
de/identity.h include/matnorm.h include/matrix.h include/multinomial.h include/p
olynomial.h include/quatern.h include/rational.h include/symbolicc++.h include/v
ecnorm.h include/vector.h include/verylong.h '/usr/include'
test -z "/usr/include/symbolic" || /bin/mkdir -p "/usr/include/symbolic"
 /bin/install -c -m 644 include/symbolic/constants.h include/symbolic/equation.h
 include/symbolic/functions.h include/symbolic/integrate.h include/symbolic/numb
er.h include/symbolic/product.h include/symbolic/solve.h include/symbolic/sum.h
include/symbolic/symbol.h include/symbolic/symbolic.h include/symbolic/symbolicc
++.h include/symbolic/symerror.h include/symbolic/symmatrix.h '/usr/include/symb
olic'
make[1]: Leaving directory '/mnt/samsung/home/browni/Téléchargements/SymbolicC++
3-3.35'
root [ /mnt/samsung/home/browni/Téléchargements/SymbolicC++3-3.35 ]#
```

**Figure 6.10:** *If you are successful building/compiling then install SymbolicC++, it will look like this.*

(this zip/tar only contains lisp, examples and headers file for SymbolicC++ and can't be compiled to library)

**Figure 6.11:** *The libraries will be installed in* ***/usr/lib****.*

## ii. Symbolic Derivation and Integration with SymbolicC++ Library

Now, for the real challenge we want to derivate and integrate a simple function $f(x) = x^3 + x^2$ with respect to $x$, it is a function of one variable, so it won't be too hard.

Create a file name **derivation.cpp**.

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
        Symbolic x("x");
        Symbolic y, dy;
        y = (x*x*x)+ (x*x);
        cout << "y = " << y << endl;
        dy = df(y, x);

        cout << "derivative of y = " << dy << endl;
        return 0;
}
```

**C++ Code 39:** *derivation.cpp "Symbolic Derivation for Function in One Variable C++"*

In the working directory, open terminal and type:
**g++ -o result derivation.cpp -lsymbolicc++**
**./result**

Afterwards, create a file name **integral.cpp**.

**Figure 6.12:** *The result of differentiating $f(x) = x^3 + x^2$ with respect to $x$ (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/SymbolicC++/derivation.cpp****).*

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
        Symbolic x("x"), y;
        y = (x*x*x)+ (x*x);
        cout << "y = " << y << endl;
        y = integrate(y,x);
        cout << "integral of y = " << y << endl;

        return 0;
}
```

**C++ Code 40:** *integral.cpp "Symbolic Integration for Function in One Variable C++"*

Now in the working directory, open terminal and type:
**g++ -o integralresult integral.cpp -lsymbolicc++**
**./integralresult**

**Figure 6.13:** *The result of integrating $f(x) = x^3 + x^2$ with respect to $x$ (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/SymbolicC++/integral.cpp****).*

You can create a Makefile for this example, rename the cpp file into **main.cpp**, then create the Makefile.

```
CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++ -lsymbolicc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<
```

```
all:: main

main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

**C++ Code 41:** *Makefile "SymbolicC++ Example"*

Now in the working directory, open terminal and type:
**make**
**./main**

To clean all files created after compiling (.o files and the binary executable file) type:
**make clean**

## III. Eigen Library

Besides SymbolicC++, we are going to use Eigen library to help us for numerical linear algebra computation. Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Eigen doesn't have any dependencies other than the C++ standard library. You can read more on Eigen here:
**https://eigen.tuxfamily.org/index.php?title=Main_Page**

Eigen in used in various fields, to get a clear vision of what it is capable of, let see the usage of Eigen by a lot of projects all around the world in science:

1. GINESTRA, a semiconductor device simulator with a focus on advanced dielectric materials and interfaces.
2. G+Smo, an open-source library for geometric design and numerical simulation with isogeometric analyis.
3. FlexibleSUSY, a spectrum generator which calculates the masses of elementary particles.
4. The ATLAS experiment at the LHC (Large Hadron Collider) at CERN is using Eigen, as reported in this article, noting "Eigen was chosen since it offered the largest performance improvements for ATLAS use cases of the options investigated."
5. The Large Survey Synoptic Telescope (website; trac) is a project to build a 3.2Mpixel camera on an 8.4m telescope and survey the entire visible sky every three days.
6. Avogadro, an opensource advanced molecular editor.
7. The 3D astronomical visualization application Celestia is now using Eigen for all orbital and geometric calculation.
8. ENigMA is a multiphysics numerical library which uses Eigen.

9. iMSTK is an open source software toolkit written in C++ that aids rapid prototyping of interactive multi-modal surgical simulations.
10. Quantum++ is a modern C++ general purpose quantum computing library, composed solely of template header files.
11. Quit, a collection of programs for processing quantitative MRI data.
12. Spin-Scenario, a flexible scripting environment for realistic magnetic resonance (NMR/MRI) simulations.
13. mbsolve is an open-source solver tool for the Maxwell-Bloch equations, which are used to model light-matter interaction in nonlinear optics.

In computer graphics:

1. Computational Geometry Algorithms Library (CGAL), a collaborative effort to develop a robust, easy to use, and efficient C++ software library of geometric data structures and algorithms.
2. VcgLib, an opensource C++ template library for the manipulation and processing of triangle and tetrahedral meshes. (switched from home made math classes)
3. MeshLab, an opensource software for the processing and editing of unstructured 3D triangular meshes and point cloud. (switched from vcglib's math classes)
4. Theia, an opensource C++ structure from motion library tailored for researchers, BSD licensed.
5. openMVG, a simple library for multiple view geometry.
6. libigl is a simple C++ geometry processing library with wide functionality.
7. Madplotlib makes it easier to plot 2D charts on Qt from data created by Eigen::ArrayXf.
8. ApproxMVBB is a small library to compute fast approximate oriented bounding boxes of 3D point clouds.

in Robotics and engineering:

1. The Yujin Robot company uses Eigen for the navigation and arm control of their next gen robots. (switched from blitz, ublas and tvmet)
2. The Robotic Operating System (ROS) developed by Willow Garage.
3. The Darmstadt Dribblers autonomous Humanoid Robot Soccer Team and Darmstadt Rescue Robot Team use Eigen for navigation and world modeling.
4. The Mobile Robot Programming Toolkit (MRPT), a set of libraries for SLAM, localization and computer vision, moved to Eigen (switched from home made math classes).
5. RBDL: a C++ library for rigid body dynamics.
6. RL a self-contained C++ library for robot kinematics, motion planning and control.

7. BTK is a Biomechanical ToolKit, licensed under BSD whose primary goal is to propose a set of tools for the analysis of the human body motion which is independent of any acquisition system. It proposes bindings for Matlab/Octave and Python, and a GUI software called Mokka to visualize/analyze 3D/2D motion capture data.
8. libpointmatcher is a "Iterative Closest Point" library for 3D mapping in robotics.
9. RobOptim is a modern, Open-Source, C++ library for numerical optimization applied to robotics.
10. towr is a light-weight and extensible C++ library for trajectory optimization for legged robots.
11. Pinocchio: a fast and efficient Rigid Body Dynamics library

In numerical computations:

1. Google's TensorFlow is an Open Source Software Library for Machine Intelligence
2. Google's Ceres solver is a portable C++ library that allows for modeling and solving large complicated nonlinear least squares problems.
3. The Manifold ToolKit MTK provides easy mechanisms to enable arbitrary algorithms to operate on manifolds. It also provides a Sparse Least Squares Solver (SLoM) and an Unscented Kalman Filter (UKFoM).
4. IFOPT is a modern, light-weight, Eigen-based C++ interface to Nonlinear Programming solvers, such as Ipopt and Snopt.
5. CppNumericalSolvers is a lightweight header-only library for non-linear optimization including various solvers: CG, L-BGFS-B, CMAes, Nelder-Mead.
6. GTSAM is a library implementing smoothing and mapping (SAM) in robotics and vision, using factor graphs and Bayes networks.
7. g2o is an open-source C++ framework for optimizing graph-based nonlinear least-square problems.
8. redsvd is a RandomizED Singular Value Decomposition library for sparse or very large dense matrices.
9. Shogun: a large scale machine learning toolbox.
10. Stan: a statistical package based on Eigen that includes a reverse-mode automatic differentiation implementation.
11. StOpt, the STochastic OPTimization library aims at providing tools in C++ for solving some stochastic optimization problems encountered in finance or in the industry.
12. Nelson an open computing environment for engineering and scientific applications using modern C/C++ libraries (Boost, Eigen, FFTW, . . . ) and others state of art numerical libraries. (GPL2)
13. trustOptim is a trust-region based non linear solver supporting sparse Hessians (C++ implementation with R binding).

## i. Compile and Install Eigen3 Library

To download it, open xterm / terminal and type:
**wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz**
Extract it, then enter the directory.

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/opt/hamzstlib \
$ —DCMAKE_INSTALL_LIBDIR=/opt/hamzstlib \
$ —Wno—dev .. &&
$ make
```

**C++ Code 42:** *Compile Eigen*

After that, type:
**make install**

The installed eigen is located under **/opt/hamzstlib/include/eigen3**

**Figure 6.14:** *If you are successful building/compiling then install Eigen, the installed directory will look like this.*

# IV. GiNaC Library

GiNaC is a C++ library for symbolic computations. The GiNaC open framework for symbolic computation within the C++ programming language does not try to define a language of its own as conventional CAS do. Instead, it extends the capabilities of C++ by symbolic manipulations.

You can read more on GiNaC here:
**https://www.ginac.de/tutorial/**

## i. Compile and Install GiNaC Library

First you need to have CLN with version above or equal 1.2.2, download it from terminal by typing:
**wget https://www.ginac.de/CLN/cln-1.3.6.tar.bz2**

Extract it, then enter the directory

```
$ ./configure --enable-shared --prefix=/opt/hamzstlib &&
$ make
```

**C++ Code 43:** *Compile CLN*

After that, type:
**make install**

Now, download the GiNaC source code from here:
**https://www.ginac.de/ginac.git/**
Extract it, then enter the directory.

```
$ export CXXFLAGS="-Wall␣-O2"
$ ./configure --enable-shared --prefix=/opt/hamzstlib
```

**C++ Code 44:** *Compile GiNaC*

After proper configuration you can now build the library by typing
**make**

Just to make sure GiNaC works properly you may run a collection of regression tests by typing
**make check**

After that, to install GiNaC on your system, simply type
**make install**

The explanation why we need CLN is because GiNaC uses Bruno Haible's library CLN, and we need this library to store numerical things. CLN stands for Class Library for Numbers or alternatively Common Lisp Numbers. CLN by itself is build on top of another library, the GNU Multiple Precision library GMP, which is an extremely fast library for arbitrary long integers and rationals as well as arbitrary precision floating point numbers. It is very commonly used by several popular cryptographic applications. CLN extends GMP by several useful things: First, it introduces the complex number fields over either reals or rationals. Second, it automatically

converts rationals to integers if the denominator is unity and complex numbers to real numbers if the imaginary part vanishes and also correctly treats algebraic functions. Third, it provides good implementations of state-of-the-art algorithms for all trigonometric and hyperbolic functions as well as for calculation of some useful constants.

## V. Armadillo Library

Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use.

Features:

1. It provides efficient classes for vectors, matrices and cubes; dense and sparse matrices are supported
2. Various matrix decompositions (eigen, SVD, QR, etc) are provided through integration with LAPACK, or one of its high performance drop-in replacements (eg. MKL or OpenBLAS)
3. Can automatically use OpenMP multi-threading (parallelisation) to speed up computationally expensive operations
4. Can be used for machine learning, pattern recognition, computer vision, signal processing, bioinformatics, statistics, finance, etc.

Armadillo extensively uses template meta-programming, so it is recommended to enable optimisation when compiling programs (eg. use the **-O2** or **-O3** options for GCC or clang).

You can read more on Armadillo here:
**https://arma.sourceforge.net/docs.html**

Caveats:

1. The matrix power is generally not the same as applyging the **pow()** function to each element
2. To find the inverse of a matrix, use **inv()**
3. To solve a system of linear equations, use **solve()**, more particularly to find approximate solutions to under-/over-determined or rank deficient systems of linear equations, **solve()** can be considerably faster and/or more accurate.
4. To find the matrix square root, use **sqrmat()**

### i. Compile and Install Armadillo Library

Before installing Armadillo, first install OpenBLAS and LAPACK, along with the corresponding development/header files, to download OpenBLAS go to this link:
**https://github.com/OpenMathLib/OpenBLAS/releases/tag/v0.3.26**

Extract it, then enter the directory.

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/opt/hamzstlib &&
$ make
```

**C++ Code 45:** *Compile OpenBLAS*

After that, type:
**make install**

```
Install the project...
-- Install configuration: ""
-- Installing: /opt/hamzstlib/lib/libopenblas.a
-- Installing: /opt/hamzstlib/include/openblas/openblas_config.h
-- Installing: /opt/hamzstlib/include/openblas/f77blas.h
-- Installing: /opt/hamzstlib/include/openblas/cblas.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke_example_aux.h
-- Installing: /opt/hamzstlib/include/openblas/lapack.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke_config.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke_mangling.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke_utils.h
-- Installing: /opt/hamzstlib/include/openblas/lapacke_mangling.h
-- Installing: /opt/hamzstlib/lib/pkgconfig/openblas.pc
-- Installing: /opt/hamzstlib/lib/cmake/OpenBLAS/OpenBLASConfig.cmake
-- Installing: /opt/hamzstlib/lib/cmake/OpenBLAS/OpenBLASConfigVersion.cmake
-- Installing: /opt/hamzstlib/lib/cmake/OpenBLAS/OpenBLASTargets.cmake
-- Installing: /opt/hamzstlib/lib/cmake/OpenBLAS/OpenBLASTargets-noconfig.cmake
root [ /mnt/samsung/home/browni/Téléchargements/OpenBLAS-0.3.26/build ]#
```

**Figure 6.15:** *If you are successful installing OpenBLAS, it will look like this.*

After finish with OpenBLAS, now we go to download LAPACK from here:
**https://github.com/Reference-LAPACK/lapack/archive/refs/tags/v3.10.1.tar.gz**
After you extract it, enter the folder then compile it.

```
$ mkdir build &&
$ cd build &&
$ ccmake .. && make
```

**C++ Code 46:** *Compile LAPACK*

(You can choose Lapacke, Lapack++, CBLAS, anything you want)

After finish click (c) to configure then (g) to generate, then type:
**make -j8**

After finish compiling, install it, type:
**make install**

**BLAS** and **LAPACK** are the bottom of food chain for numerical and linear algebra computation till today. For example if you want to solve a linear system $A\boldsymbol{x} = \boldsymbol{b}$, where $A$ is the coefficient matrix, $\boldsymbol{b}$ is the solution vector, and $\boldsymbol{x}$ are the unknowns. We can implement Gaussian elimination with this **LAPACK** library, the routine that we will use is **dgesv**.

Now download Armadillo from:
**https://sourceforge.net/projects/arma/files/armadillo-12.6.7.tar.xz/download**
Extract it, then enter the directory.

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/opt/hamzstlib &&
$ make
```

**C++ Code 47:** *Compile Armadillo*

**How to use Armadillo:**

- Don't forget to use **using namespace arma;** at the beginning of the C++ code, because most of the functions need to include arma, it is almost the same as **std::cout**, so you can just call to construct a matrix name $A$ like this with Armadillo:
  **mat A;**

  if you forget the **using namespace arma;**, then it will prompt an error on compiling.

## VI. GSL Library

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.
The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

Unlike the licenses of proprietary numerical libraries the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

### i. Compile and Install GSL Library

To download it, type:
**wget https://mirror.ibcp.fr/pub/gnu/gsl/gsl-latest.tar.gz**
Extract it, then enter the directory.

```
$ ./autogen.sh \
$ ./configure ——prefix=/opt/hamzstlib \
$ ——enable—maintainer—mode &&
$ make
```

**C++ Code 48:** *Compile GSL*

After that, type:
**make install**

**Function areas in GSL**

The complete range of subject areas covered by the library includes

| | | |
|---|---|---|
| Complex Numbers | Quasi-Random Sequences | Discrete Hankel Transforms |
| Roots of Polynomials | Random Distributions | Root-Finding |
| Special Functions | Statistics | Minimization |
| Vectors and Matrices | Histograms | Least-Squares Fitting |
| Permutations | N-Tuples | Physical Constants |
| Sorting | Monte Carlo Integration | IEEE Floating-Point |
| BLAS Support | Simulated Annealing | Discrete Wavelet Transforms |
| Linear Algebra | Differential Equations | Basis splines |
| Eigensystems | Interpolation | Running Statistics |
| Fast Fourier Transforms | Numerical Differentiation | Sparse Matrices and Linear Algebra |
| Quadrature | Chebyshev Approximation | |
| Random Numbers | Series Acceleration | |

**Figure 6.16:** *Scientific problems that are covered by GSL library.*

## VII. Tomsolver Library (Nonlinear System Solver)

Tomsolver is a C++ library for solving simple nonlinear system of equations.

Another library you can use for solving nonlinear equations are **Ceres, PetSC, dlib**, they all can be used from C++.

Why we need a separate nonlinear solver? Computing for nonlinear equation solution isn't easy. It's not the same as solving linear equations. You aren't always guaranteed to get a solution. Your choice of initial condition and incrementation strategy can have a profound effect on the solution you will get.

You can read more on Tomsolver here:
**https://github.com/tomwillow/tomsolver/**

### i. Compile and Install Tomsolver Library

First, clone the repository by typing:
**git clone https://github.com/tomwillow/tomsolver**

Then enter the directory, now since GFreya OS is immune / not using internet and Tomsolver need an internet to make the compiling works, thus in order to make it works we can just copy **../tomsolver/single/include/tomsolver/tomsolver.hpp** to **/usr/include/tomsolver/**.

Now you can just use tomsolver' features to solve simple nonlinear system.

**(Warning! This is for advanced user only)**
But if you still want to compile it, we have to use GFreya OS that is in Qemu inside CAELinux so we can access internet from there, from inside CAELinux OS start GFreya OS in Qemu and then type

```
$ cd /sources
$ git clone https://github.com/tomwillow/tomsolver
$ cd tomsolver
$ mkdir build
$ cd build
$ cmake —DCMAKE_INSTALL_PREFIX=/usr .. &&
$ make
```

**C++ Code 49:** *Compile Tomsolver in GFreya OS in Qemu*

After that, you can do **rsync** to the GFreya OS in the dual boot from GFreya OS in Qemu (warning: this is for advanced ) from CAELinux' terminal type:
**sudo -i**
**modprobe nbd max_part=8**
**qemu-nbd –connect=/dev/nbd0 /var/lib/libvirt/images/GFreya.qcow2**
**cd /media/browni/ee99dc21-0a85-4f5d-b2f5-c3351f0da878/sources**
**rsync -rah –stats –info=progress2 'tomsolver' '/FreyaSSD/sources/'**
(this will move the compiled Tomsolver into GFreya OS main OS' sources directory, GFreya OS main OS is installed in the second SSD of my laptop and the path to it is: **/FreyaSSD**)

Now shutdown / reboot and enter GFreya OS as the main OS from the dual boot choice in GRUB, and type from GFreya OS' terminal:
**cd /sources/tomsolver/build/**
**make install**
(Tomsolver has been compiled with internet from GFreya OS in Qemu inside CAELinux OS, then we do rsync to the GFreya OS main OS, finally we install it in the GFreya OS main OS)

It happens that I am using CAELinux for my other OS in my laptop, if your OS is other than CAELinux then you have to adjust the syntax.

**Figure 6.17:** *The location for* ***tomsolver.hpp*** *inside the cloned repository.*

```
xterm
root [ /sources/tomsolver/build ]# make install
Consolidate compiler generated dependencies of target TomSolver
[ 33%] Built target TomSolver
Consolidate compiler generated dependencies of target gtest
[ 38%] Built target gtest
Consolidate compiler generated dependencies of target gtest_main
[ 42%] Built target gtest_main
Consolidate compiler generated dependencies of target TomSolverTest
[ 80%] Built target TomSolverTest
Consolidate compiler generated dependencies of target TomSolverSingleTest
[ 85%] Built target TomSolverSingleTest
Consolidate compiler generated dependencies of target Example_Solve
[ 90%] Built target Example_Solve
Consolidate compiler generated dependencies of target Example_Solve2
[ 95%] Built target Example_Solve2
Consolidate compiler generated dependencies of target DiffMachine
[100%] Built target DiffMachine
Install the project...
-- Install configuration: ""
-- Old export file "/usr/lib/cmake/GTest/GTestTargets.cmake" will be replaced.
Removing files [/usr/lib/cmake/GTest/GTestTargets-noconfig.cmake].
-- Installing: /usr/lib/cmake/GTest/GTestTargets.cmake
-- Installing: /usr/lib/cmake/GTest/GTestTargets-noconfig.cmake
-- Installing: /usr/lib/cmake/GTest/GTestConfigVersion.cmake
-- Installing: /usr/lib/cmake/GTest/GTestConfig.cmake
-- Up-to-date: /usr/include
-- Up-to-date: /usr/include/gtest
-- Installing: /usr/include/gtest/gtest-spi.h
-- Installing: /usr/include/gtest/gtest-test-part.h
-- Installing: /usr/include/gtest/gtest-printers.h
-- Installing: /usr/include/gtest/gtest_prod.h
-- Installing: /usr/include/gtest/gtest-death-test.h
-- Up-to-date: /usr/include/gtest/internal
-- Installing: /usr/include/gtest/internal/gtest-death-test-internal.h
-- Installing: /usr/include/gtest/internal/gtest-port-arch.h
-- Installing: /usr/include/gtest/internal/gtest-param-util.h
-- Up-to-date: /usr/include/gtest/internal/custom
-- Installing: /usr/include/gtest/internal/custom/README.md
-- Installing: /usr/include/gtest/internal/custom/gtest-printers.h
-- Installing: /usr/include/gtest/internal/custom/gtest-port.h
-- Installing: /usr/include/gtest/internal/custom/gtest.h
-- Installing: /usr/include/gtest/internal/gtest-port.h
-- Installing: /usr/include/gtest/internal/gtest-internal.h
-- Installing: /usr/include/gtest/internal/gtest-type-util.h
-- Installing: /usr/include/gtest/internal/gtest-string.h
-- Installing: /usr/include/gtest/internal/gtest-filepath.h
-- Installing: /usr/include/gtest/gtest_pred_impl.h
-- Installing: /usr/include/gtest/gtest-param-test.h
-- Installing: /usr/include/gtest/gtest-typed-test.h
-- Installing: /usr/include/gtest/gtest.h
-- Installing: /usr/include/gtest/gtest-message.h
-- Installing: /usr/include/gtest/gtest-matchers.h
```

**Figure 6.18:** *The installation of Tomsolver from inside the GFreya OS main OS.*

Now you can try it, compiling the whole Tomsolver library or not, the example below can still work.

```
#include <tomsolver/tomsolver.hpp>

using namespace tomsolver;

int main() {
	std::setlocale(LC_ALL, ".UTF8");

	try
	{
		// create equations as a symbolic vector
		SymVec f
		{
			Parse("(x-a)^2 + (y-b)^2") - Var("d1"),
			Parse("(x-c)^2 + (y-d)^2") - Var("d2"),
		};

		// replace known values
		f.Subs(VarsTable
		{
			{"a", 1.0},
			{"b", 2.0},
			{"c", 3.0},
			{"d", 4.0},
			{"d1", 100.0},
			{"d2", 100.0},
		});

		cout << f << endl;

		// solve
		auto ans = Solve(f);

		cout << "x = " << ans["x"] << endl;
		cout << "y = " << ans["y"] << endl;
	}
	catch (const std::exception &err)
	{
		std::cerr << err.what() << std::endl;
		return -1;
	}

	return 0;
}
```

**C++ Code 50:** *main.cpp*

To compile it, type:
**g++ -o result main.cpp**
**./result**

or with CMake, type:
**mkdir build**
**cd build**
**cmake .. make**
**./result**

The CMakeLists.txt is available in the repository' folder for this example.

```
.root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot S
ymbolicC++/ch23-Numerical Linear Algebra/Simple Nonlinear System with Tomsolver/
build ]# ./result
[(x-1)^2+(y-2)^2-100
 (x-3)^2+(y-4)^2-100]
x = 9
y = -4
```

**Figure 6.19:** *The computation to obtain solution for a simple nonlinear system of equations (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Simple Nonlinear System with Tomsolver/**).*

## VIII. Assimp (Open Asset Import Library)

Open Asset Import Library is a library to load various 3D file formats into a shared, in-memory format. It supports more than 40 file formats for import and a growing selection of file formats for export.

Open Asset Import Library is implemented in C++. The directory structure looks like this:

1. /code : Source code
2. /contrib : Third-party libraries
3. /doc : Documentation (doxysource and pre-compiled docs)
4. /fuzz : Contains the test code for the Google Fuzzer project
5. /include : Public header C and C++ header files
6. /scripts : Scripts are used to generate the loading code for some formats
7. /port : Ports to other languages and scripts to maintain those.
8. /test : Unit- and regression tests, test suite of models
9. /tools : Tools (old assimp viewer, command line 'assimp')
10. /samples : A small number of samples to illustrate possible use cases for Assimp

The usefulness of Assimp is located on the ability to import 3D object, in Mathematics and Physics if we can manipulate / do transformation on 3D object we can basically do anything, like do a progressive transformation on a wanted criminal' photo so we can know how he/she looks like after 10 years or 20 years. Assimp works really well with GLFW and OpenGL, there is a chance with only these minimal number of libraries we can create a lighter version of BIM (Building Information Modeling) with necessary Mathematics and Physics formula to model a highway, building, toll road, or a city in highest detail.

You can read more on Assimp here:
**https://assimp-docs.readthedocs.io/en/latest/**

## i. Compile and Install Assimp

To download Assimp you can clone the repository by opening terminal and type this:
**git clone https://github.com/assimp/assimp.git**

After you clone the repository, then enter the directory.

```
$ mkdir build &&
$ cd build &&
$ cmake —DCMAKE_INSTALL_PREFIX=/usr .. &&
$ make
```

**C++ Code 51:** *Compile Assimp*

After that, type:
**make install**

It will by default install the dynamic / shared library. Inside the directory **/usr/lib** you will see the installed library **libassimp.so**.

**Figure 6.20:** *The shared library of Assimp.*

# Chapter 7

# DFSimulatorC++ I: Motion in Two Dimensions

*"There's no need to rush things, there's much to explain." - **Maria Traydor (Star Ocean 3)***

## I. Position, Displacement, Velocity, and Acceleration

The general way of locating a particle is starting with a position vector, that determined where the particle is located at certain time. If we denote $\hat{r}$ as the position vector, it can be written in the unit-vector notation:

$$\hat{r} = x\hat{i} + y\hat{j} + z\hat{k}$$

where $x\hat{i}$, $y\hat{j}$, $z\hat{k}$ are the vector components of $\hat{r}$, and the coefficients $x, y$, and $z$ are its scalar components.

If a particle undergoes a displacement $\triangle\vec{r}$ in time interval $\triangle t$, its average velocity $\vec{v}_{avg}$ for that time interval is

$$\vec{v}_{avg} = \frac{\triangle\vec{r}}{\triangle t}$$

As $\triangle t$ goes to 0, $\vec{v}_{avg}$ reaches a limit called either the velocity or the instantaneous velocity $\vec{v}$:

$$\vec{v} = \frac{d\vec{r}}{dt}$$

which can be written in unit-vector notation as

$$\vec{v} = v_x\hat{i} + v_y\hat{j} + v_z\hat{k}$$

where the scalar components of $\vec{v}$ are

$$v_x = \frac{dx}{dt}, \quad v_y = \frac{dy}{dt}, \quad v_z = \frac{dz}{dt}$$

If a particle's velocity changes from $\vec{v}_1$ to $\vec{v}_2$ in time interval $\triangle t$, its average acceleration during $\triangle t$ is

$$\vec{a}_{avg} = \frac{\vec{v}_2 - \vec{v}_1}{\triangle t}$$

As $\triangle t$ goes to 0, $\vec{a}_{avg}$ reaches a limiting value called the instantaneous acceleration $\vec{a}$:

$$\vec{a} = \frac{d\vec{v}}{dt}$$

which can be written in unit-vector notation as

$$\vec{a} = a_x\hat{i} + a_y\hat{j} + a_z\hat{k}$$

where the scalar components of $\vec{a}$ are

$$a_x = \frac{dv_x}{dt}, \quad a_y = \frac{dv_y}{dt}, \quad v_z = \frac{dv_z}{dt}$$

The basic equations for motion with constant acceleration are

$$v = v_0 + at \tag{7.1}$$

$$x - x_0 = v_0 t + \frac{1}{2}at^2 \tag{7.2}$$

Another equations that can be used to solve any constant acceleration problem:

$$v^2 = v_0^2 + 2a(x - x_0) \tag{7.3}$$

$$x - x_0 = \frac{1}{2}(v_0 + v)t \tag{7.4}$$

$$x - x_0 = vt - \frac{1}{2}at^2 \tag{7.5}$$

To prove equation **(7.1)**, we can write the indefinite integral from the definition of acceleration

$$\begin{aligned} dv &= a\ dt \\ \int dv &= \int a\ dt \\ \int dv &= a\int dt \\ v &= at + C \end{aligned}$$

To evaluate the constant $C$, we let $t = 0$, at which $v(0) = v_0$ (the initial value), thus

$$\begin{aligned} v_0 &= a(0) + C \\ v_0 &= C \end{aligned}$$

hence we will obtain $v = v_0 + at$.
Now let's take the indefinite integral for the definition of velocity with respect to time

$$\begin{aligned} dx &= v\ dt \\ \int dx &= \int v\ dt \\ \int dx &= \int (v_0 + at)\ dt \\ \int dx &= v_0\int\ dt + a\int t\ dt \\ x &= v_0 t + \frac{1}{2}at^2 + K \end{aligned}$$

To find $K$, we set the initial value at $t = 0$ with $x(0) = x_0$, thus

$$x_0 = v_0(0) + \frac{1}{2}a(0)^2 + K$$
$$x_0 = K$$

we will obtain $x - x_0 = v_0 t + \frac{1}{2}at^2$, this proves equation **(7.2)**.

## II. Projectile Motion

We consider a special case in two-dimensional motion: when a particle moves in a vertical plane with some initial velocity $\vec{v}_0$ but its acceleration is always the freefall acceleration $\vec{g}$, which is downward. That particle is called projectile, mean that it is being projected or launched, and its motion is called projectile motion, like throwing a baseball into its ring. Ballistic, missile, a lot of sports like golf, tennis involve the study of projectile motion.

In projectile motion, a particle is launched into the air with a speed of $\vec{v}_0$ and at an angle $\theta_0$

**Figure 7.1:** *The illustration of projectile motion, an object is launched into the air at the origin of the coordinate system with launch velocity $\vec{v}_0$ at angle $\theta_0$.*

(as measured from a horizontal $x$ axis). Its horizontal acceleration during flight is zero, while its vertical acceleration is $-g$. Thus, the equation of motion for the particle can be written as

$$x - x_0 = (v_0 \ \cos \ \theta_0)t \tag{7.6}$$

$$y - y_0 = (v_0 \ \sin \ \theta_0)t - \frac{1}{2}gt^2 \tag{7.7}$$

$$v_y = v_0 \ \sin \ \theta_0 - gt \tag{7.8}$$

$$v_y^2 = (v_0 \ \sin \ \theta_0)^2 - 2g(y - y_0) \tag{7.9}$$

## III. Simulation for Projectile Motion with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:

**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose, e.g. **Motion in 2D/Projectile Motion**.

```
#include "test.h"
#include <iostream>

class ProjectileMotion: public Test
{
        public:

        ProjectileMotion()
        {
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                b2Timer timer;
                // Perimeter Ground body
                {
                        b2BodyDef bd;
                        b2Body* ground = m_world->CreateBody(&bd);

                        b2EdgeShape shapeGround;
                        shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                            (30.0f, 0.0f));
                        ground->CreateFixture(&shapeGround, 0.0f);

                        b2EdgeShape shapeTop;
                        shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
                            (30.0f, 30.0f));
                        ground->CreateFixture(&shapeTop, 0.0f);

                        b2EdgeShape shapeLeft;
                        shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                            (-40.0f, 30.0f));
                        ground->CreateFixture(&shapeLeft, 0.0f);

                        b2EdgeShape shapeRight;
                        shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
                            (30.0f, 30.0f));
                        ground->CreateFixture(&shapeRight, 0.0f);
                }

                // Create the ball
                b2CircleShape ballShape;
                ballShape.m_p.SetZero();
```

```
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.15f; // the bounciness
        ballFixtureDef.density = 7.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(-25.0f, 0.5f);
        // ballBodyDef.angularDamping = 0.2f;

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);

        m_createTime = timer.GetMilliseconds();
    }
    b2Body* bodylbox;
    b2Body* m_ball;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_S:
            int theta = 45;
            float v0 = 20.0f;
            m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), v0*
                sin(theta)));
            break;
        }
    }

    void Step(Settings& settings) override
    {
        b2MassData massData = m_ball->GetMassData();
        g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
            massData.mass);
        m_textLine += m_textIncrement;

        b2Vec2 position = m_ball->GetPosition();
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
            f", position.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
```

```
                    f", position.y);
                m_textLine += m_textIncrement;

                b2Vec2 velocity = m_ball->GetLinearVelocity();
                g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x = %.6
                    f", velocity.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y = %.6
                    f", velocity.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
                    ",
                m_createTime);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f \n", velocity.x, velocity.y);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new ProjectileMotion;
        }

        float m_createTime;
};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Motion",
    ProjectileMotion::Create);
```

**C++ Code 52:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- Create a ball that is staying still on the ground.

```
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.15f; // the bounciness
            ballFixtureDef.density = 7.3f; // this will affect the ball
                mass
            ballFixtureDef.friction = 0.1f;
            ballFixtureDef.shape = &ballShape;

            b2BodyDef ballBodyDef;
```

```
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-25.0f, 0.5f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
```

You can change the restitution, density, friction for different kind of ball for your own research.

- Create keyboard event, when pressed 'S' the ball is launched with initial velocity, $v_0 = 20$ and the initial angle of $\theta_0 = 45^0$.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_S:
                int theta = 45;
                float v0 = 20.0f;
                m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta),
                    v0*sin(theta)));
                break;
        }
}
```

- To show the data of the position, velocity and the mass of the ball for our simulation, then print the velocity data into xterm / terminal.

```
void Step(Settings& settings) override
{
        b2MassData massData = m_ball->GetMassData();
        g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
            , massData.mass);
        m_textLine += m_textIncrement;

        b2Vec2 position = m_ball->GetPosition();
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, x
            = %.6f", position.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, y
            = %.6f", position.y);
        m_textLine += m_textIncrement;

        b2Vec2 velocity = m_ball->GetLinearVelocity();
        g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x
            = %.6f", velocity.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y
            = %.6f", velocity.y);
```

```
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time =
                    %6.2f ms",
                m_createTime);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f \n", velocity.x, velocity.y);
                //printf("%4.2f %4.2f \n", position.x, position.y);
                Test::Step(settings);
        }
```

After recompiling this, you can save it into textfile by opening the testbed with this command:
**./testbed > projectileoutput.txt**
After you close the testbed to record the result, then see it at the current working directory where **testbed** is located and see **projectileoutput.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 2 columns.

To plot the velocity in $x$ and $y$ axis for the projectile motion of the ball with respect to time, now open terminal at the directory containing the **projectileoutput.txt** and type:
**gnuplot**
**set xlabel "time"**
**set ylabel "v_{x}"**
**plot "projectileoutput.txt" using 1 title "" with lines**
**set ylabel "v_{y}"**
**plot "projectileoutput.txt" using 2 title "" with lines**

Now you can go back to the source code of the projectile motion and uncomment the line to print the position and comment the line to print the velocity:

```
//printf("%4.2f %4.2f \n", velocity.x, velocity.y);
printf("%4.2f %4.2f \n", position.x, position.y);
```

Now, we can plot the position of the ball with gnuplot, recompile the testbed to make the change occurs then type:
**./testbed > projectileoutput.txt**

Plot it with gnuplot from the working directory:
**gnuplot**
**set xlabel "time"**
**set ylabel "v_{x}"**
**plot "projectileoutput.txt" using 1:2 title "Ball trajectory" with lines**

## IV. Simulation for Projectile Dropped from Above with Box2D

This is coming from a sample problem in [6], but I modify it a little bit. Suppose that Europe is now in war and get help of food supplies from its allies, if there are two rescue drones, the first

**Figure 7.2:** *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the $v_x$ (the horizontal velocity).*

**Figure 7.3:** *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the $v_y$ (the horizontal velocity).*

**Figure 7.4:** *The "projectileoutput.txt" is being plotted by using gnuplot for the data of the x and y (position of the ball). You can see that the ball is still sliding forward if you see the Box2D simulation, since we make the restitution for the ball 0.15, a bit bouncy. This simulation is very useful to design a better golf ball or galleon canon for spaceship to hit the target faster and accurate.*

**Figure 7.5:** *The modified Box2D simulation of a ball being projected with initial angle $\theta_0 = 45^0$ and initial velocity of $v_0 = 20$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/projectile_motion.cpp****).*

drone is sending a cheese ball (shape of a circle in 2 dimension) and the second drone is sending boxes full of chocolates and medicine. Don't ask why they don't send vegetables and fruits that are healthier, it is at war now. Suppose both flying at different height, constant height of $h_1 = 40$ and $h_2 = 30$, but their speeds are the same, at certain speed of $v_0 = 20$. What should be the angle $\phi_1$ and $\phi_2$ of the first and second drone line of sight to the net below (the height of the net is 5, with width of 10, centered at $x = 0$) that will capture the cheese ball and boxes full of chocolate and medicine?

**Solution:**
We need to know that $\phi_1$ is given by

$$\phi_1 = \tan^{-1}\left(\frac{x_1}{h_1}\right)$$

and $\phi_2$ is given by

$$\phi_2 = \tan^{-1}\left(\frac{x_2}{h_2}\right)$$

where $x_1$ and $x_2$ are the horizontal coordinates of the net to capture the cheese ball and the chocolate boxes with medicine respectively from where the supplies being drop by each of the drones. We will have two cases and should be able to find $x_1$ and $x_2$ for each of the case with

$$x_1 - x_0 = (v_0 \cos\theta_0)t_1$$

$$x_2 - x_0 = (v_0 \cos\theta_0)t_2$$

we know that $x_0 = 0$ because the origin is placed at the point of release from each of the drone. $v_0$ is the drone velocity, and both drones fly at the same velocity of $v_0 = 20$, with the angle $\theta_0 = 0^0$ measured relative to the positive direction of the $x$ axis, since the supplies are being dropped not being projected / launched.

To find $t$, the time for the supply arrive at the net, we can use the vertical motion displacement equation for the first drone:

$$y_1 - y_0 = (v_0 \ \sin \ \theta_0)t_1 - \frac{1}{2}gt_1^2$$

and for the second drone vertical displacement equation:

$$y_2 - y_0 = (v_0 \ \sin \ \theta_0)t_2 - \frac{1}{2}gt_2^2$$

The vertical displacement for each of the drone will be negative, the negative value indicates that the supplies moves downward. Thus, solving for $t$ for the first drone

$$\begin{aligned}
y_1 - y_0 &= (v_0 \ \sin \ \theta_0)t_1 - \frac{1}{2}gt_1^2 \\
-40 &= (20 \ \sin \ 0^0)t_1 - \frac{1}{2}(9.8)t_1^2 \\
-40 &= -\frac{1}{2}(9.8)t_1^2 \\
t_1 &= \sqrt{\frac{80}{9.8}} \\
t_1 &= 2.8571
\end{aligned}$$

then for the second drone

$$
\begin{aligned}
y_2 - y_0 &= (v_0 \ \sin \ \theta_0)t_2 - \frac{1}{2}gt_2^2 \\
-30 &= (20 \ \sin \ 0^0)t_2 - \frac{1}{2}(9.8)t_2^2 \\
-30 &= -\frac{1}{2}(9.8)t_2^2 \\
t_2 &= \sqrt{\frac{60}{9.8}} \\
t_2 &= 2.4743
\end{aligned}
$$

After we obtain $t$ for both drones, we can estimate the horizontal distance between the drone to the net and the drone' line of sight

$$
\begin{aligned}
x_1 - x_0 &= (v_0 \cos\theta_0)t_1 \\
x_1 - 0 &= (20 \ \cos \ 0^0)(2.8571) \\
x_1 &= 57.141999
\end{aligned}
$$

$$\phi_1 = \tan^{-1}\left(\frac{x_1}{h_1}\right) = \tan^{-1}\left(\frac{57.141999}{40}\right) = \tan^{-1}(1.428549) = 0.960063 \text{ rad} \approx 55.007575^0$$

The horizontal distance for the second drone

$$
\begin{aligned}
x_2 - x_0 &= (v_0 \cos\theta_0)t_2 \\
x_2 - 0 &= (20 \ \cos \ 0^0)(2.4743) \\
x_2 &= 49.48716
\end{aligned}
$$

$$\phi_2 = \tan^{-1}\left(\frac{x_2}{h_2}\right) = \tan^{-1}\left(\frac{49.48716}{30}\right) = \tan^{-1}(1.649572) = 1.0258174 \text{ rad} \approx 58.775^0$$

Either the angles of depression $\phi_1$ and $\phi_2$ or the horizontal distance between the center of the net and the drone can be used as the data input to release or drop the supplies, as sensor for drone or mechatronics that can calculate horizontal distance toward a target and altitude is available along with one that can calculate angle of depression toward a designated target.

For the C++ simulation, you can copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

```
#include "test.h"
#include <iostream>

// This is used to test sensor shapes.
```

```
class ProjectileDropped : public Test
{
	public:

	enum
	{
		e_count = 10
	};

	ProjectileDropped()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f,
				10.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		// Create the net centering at x=0
		{
			b2PolygonShape shape;
			shape.SetAsBox(0.5f, 0.125f);

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.density = 20.0f;
			fd.friction = 0.2f;

			b2RevoluteJointDef jd;

			b2Body* prevBody = ground;
			for (int32 i = 0; i < e_count; ++i)
			{
				b2BodyDef bd;
				bd.type = b2_dynamicBody;
				bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0
```

```
                    f is the height
                b2Body* body = m_world->CreateBody(&bd);
                body->CreateFixture(&fd);

                b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
                     the chain ball anchor
                jd.Initialize(prevBody, body, anchor);
                m_world->CreateJoint(&jd);

                if (i == (e_count >> 1))
                {
                       m_middle = body;
                }
                prevBody = body;
        }

        b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
            right anchor
        jd.Initialize(prevBody, ground, anchor);
        m_world->CreateJoint(&jd);
}
// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-52.14199f, 40.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);
int theta = 0;
float v0 = 20.0f;
m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));

// Create Breakable dynamic body
{
        b2BodyDef bd;
```

```
                bd.type = b2_dynamicBody;
                bd.position.Set(-44.48716f, 30.0f);
                bd.angle = 0.85f * b2_pi;
                m_body1 = m_world->CreateBody(&bd);

                m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0
                    f);
                m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

                m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f
                    );
                m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

                m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
        }

        m_break = false;
        m_broke = false;
    }
    b2Body* m_middle;
    b2Body* m_ball;

    b2Body* m_body1;
    b2Vec2 m_velocity;
    float m_angularVelocity;
    b2PolygonShape m_shape1;
    b2PolygonShape m_shape2;
    b2Fixture* m_piece1;
    b2Fixture* m_piece2;
    bool m_broke;
    bool m_break;

    void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
        override
    {
        if (m_broke)
        {
                // The body already broke.
                return;
        }

        // Should the body break?
        int32 count = contact->GetManifold()->pointCount;

        float maxImpulse = 0.0f;
        for (int32 i = 0; i < count; ++i)
        {
                maxImpulse = b2Max(maxImpulse, impulse->normalImpulses
```

```
                        [i]);
                }

                if (maxImpulse > 40.0f)
                {
                        // Flag the body for breaking.
                        m_break = true;
                }
        }

        void Break()
        {
                // Create two bodies from one.
                b2Body* body1 = m_piece1->GetBody();
                b2Vec2 center = body1->GetWorldCenter();

                body1->DestroyFixture(m_piece2);
                m_piece2 = NULL;

                b2BodyDef bd;
                bd.type = b2_dynamicBody;
                bd.position = body1->GetPosition();
                bd.angle = body1->GetAngle();
                b2Body* body2 = m_world->CreateBody(&bd);
                m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

                // Compute consistent velocities for new bodies based on
                // cached velocity.
                b2Vec2 center1 = body1->GetWorldCenter();
                b2Vec2 center2 = body2->GetWorldCenter();

                b2Vec2 velocity1 = m_velocity + b2Cross(m_angularVelocity,
                    center1 - center);
                b2Vec2 velocity2 = m_velocity + b2Cross(m_angularVelocity,
                    center2 - center);

                body1->SetAngularVelocity(m_angularVelocity);
                body1->SetLinearVelocity(velocity1);

                body2->SetAngularVelocity(m_angularVelocity);
                body2->SetLinearVelocity(velocity2);
        }

        void Step(Settings& settings) override
        {
                b2MassData massData = m_ball->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
                    massData.mass);
```

```
                m_textLine += m_textIncrement;

                b2Vec2 position = m_ball->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
                    f", position.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
                    f", position.y);
                m_textLine += m_textIncrement;

                b2Vec2 positionbox1 = m_body1->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x =
                    %.6f", positionbox1.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y =
                    %.6f", positionbox1.y);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f %4.2f %4.2f \n", position.x, position.y,
                    positionbox1.x, positionbox1.y);

                if (m_break)
                {
                        Break();
                        m_broke = true;
                        m_break = false;
                }

                // Cache velocities to improve movement on breakage.
                if (m_broke == false)
                {
                        m_velocity = m_body1->GetLinearVelocity();
                        m_angularVelocity = m_body1->GetAngularVelocity();
                }

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new ProjectileDropped;
        }

};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Dropped",
    ProjectileDropped::Create);
```

**C++ Code 53:** *tests/projectile_dropped.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- Inside the **ProjectileDropped()**, to create the ground and the wall at the right side of the net

```
b2Body* ground = NULL;
{
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
            0.0f));
        ground->CreateFixture(&shape, 0.0f);
}
{
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f, 10.0f
            ));
        ground->CreateFixture(&shape, 0.0f);
}
// Create the net centering at x=0
{
        b2PolygonShape shape;
        shape.SetAsBox(0.5f, 0.125f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 20.0f;
        fd.friction = 0.2f;

        b2RevoluteJointDef jd;

        b2Body* prevBody = ground;
        for (int32 i = 0; i < e_count; ++i)
        {
                b2BodyDef bd;
                bd.type = b2_dynamicBody;
                bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0f
                    is the height
                b2Body* body = m_world->CreateBody(&bd);
                body->CreateFixture(&fd);

                b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
                    the chain ball anchor
                jd.Initialize(prevBody, body, anchor);
                m_world->CreateJoint(&jd);
```

```
                if (i == (e_count >> 1))
                {
                        m_middle = body;
                }
                prevBody = body;
        }

        b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
            right anchor
        jd.Initialize(prevBody, ground, anchor);
        m_world->CreateJoint(&jd);
}
```

- To create the cheese ball that is being dropped by the first drone at height of 40, the reason why I set the drop point as $52.14199f$ instead of $57.14199f$ is because the net is centered at $x = 0$ and has width of 10, 10 divided by 2 is 5, so the drone shall move forward 5 unit more before dropping the cheese ball, thus it can land at the center of the net, not too early or late. If you watch the Box2D simulation for this, both the cheese ball and chocolates with medicine boxes arrive exactly at the center of the net.

```
// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-52.14199f, 40.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
int theta = 0;
float v0 = 20.0f;
m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
```

- To create the 2 boxes of chocolates and medicine, the boxes are attached and prone to break

```
// Create Breakable dynamic body
{
        b2BodyDef bd;
```

```
            bd.type = b2_dynamicBody;
            bd.position.Set(-44.48716f, 30.0f);
            bd.angle = 0.85f * b2_pi;
            m_body1 = m_world->CreateBody(&bd);

            m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0f)
                ;
            m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

            m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f);
            m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

            m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
        }

        m_break = false;
        m_broke = false;
```

- To declare the body, polygon shape, fixtures, break boolean variable.

```
        b2Body* m_middle;
        b2Body* m_ball;

        b2Body* m_body1;
        b2Vec2 m_velocity;
        float m_angularVelocity;
        b2PolygonShape m_shape1;
        b2PolygonShape m_shape2;
        b2Fixture* m_piece1;
        b2Fixture* m_piece2;
        bool m_broke;
        bool m_break;
```

- The solver, function **PostSolve()** for the breaking apart of the boxes into two.

```
        void PostSolve(b2Contact* contact, const b2ContactImpulse*
            impulse) override
        {
            if (m_broke)
            {
                // The body already broke.
                return;
            }

            // Should the body break?
            int32 count = contact->GetManifold()->pointCount;

            float maxImpulse = 0.0f;
            for (int32 i = 0; i < count; ++i)
            {
```

```
                maxImpulse = b2Max(maxImpulse, impulse->
                    normalImpulses[i]);
            }

            if (maxImpulse > 40.0f)
            {
                // Flag the body for breaking.
                m_break = true;
            }
        }
```

- We initially have two boxes attached into one, then after landing on the net, get impulses, it breaks apart into two, that will each has its own velocity, position, and become different bodies in Box2D Physics world for this simulation. With function **Break()** we define how to separate **m_body1** into two separate bodies

```
        void Break()
        {
            // Create two bodies from one.
            b2Body* body1 = m_piece1->GetBody();
            b2Vec2 center = body1->GetWorldCenter();

            body1->DestroyFixture(m_piece2);
            m_piece2 = NULL;

            b2BodyDef bd;
            bd.type = b2_dynamicBody;
            bd.position = body1->GetPosition();
            bd.angle = body1->GetAngle();
            b2Body* body2 = m_world->CreateBody(&bd);
            m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

            // Compute consistent velocities for new bodies based on
            // cached velocity.
            b2Vec2 center1 = body1->GetWorldCenter();
            b2Vec2 center2 = body2->GetWorldCenter();

            b2Vec2 velocity1 = m_velocity + b2Cross(
                m_angularVelocity, center1 - center);
            b2Vec2 velocity2 = m_velocity + b2Cross(
                m_angularVelocity, center2 - center);

            body1->SetAngularVelocity(m_angularVelocity);
            body1->SetLinearVelocity(velocity1);

            body2->SetAngularVelocity(m_angularVelocity);
            body2->SetLinearVelocity(velocity2);
        }
```

- This will show the cheese ball mass, the position, and the boxes position for the first box, along with the condition if the boxes break apart then the function **Break()** will occur

```
void Step(Settings& settings) override
{
        b2MassData massData = m_ball->GetMassData();
        g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
            , massData.mass);
        m_textLine += m_textIncrement;

        b2Vec2 position = m_ball->GetPosition();
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, x
            = %.6f", position.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Ball Position, y
            = %.6f", position.y);
        m_textLine += m_textIncrement;

        b2Vec2 positionbox1 = m_body1->GetPosition();
        g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x
             = %.6f", positionbox1.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y
             = %.6f", positionbox1.y);
        m_textLine += m_textIncrement;

        printf("%4.2f %4.2f %4.2f %4.2f \n", position.x,
            position.y,positionbox1.x, positionbox1.y);

        if (m_break)
        {
                Break();
                m_broke = true;
                m_break = false;
        }

        // Cache velocities to improve movement on breakage.
        if (m_broke == false)
        {
                m_velocity = m_body1->GetLinearVelocity();
                m_angularVelocity = m_body1->GetAngularVelocity()
                    ;
        }

        Test::Step(settings);
}
```

Now, to plot the position of the cheese ball, chocolates and medicine supplies with gnuplot, recompile the testbed to make the change occurs then type:

**Figure 7.6:** *The modified Box2D simulation of a cheese ball and chocolates with medicine boxes being dropped from two drones with same velocity of* $v_0 = 20$ *but flying at different heights (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/projectile_dropped.cpp****).*

**Figure 7.7:** *The modified Box2D simulation clearly shows that both the supplies will arrive at the middle of the net.*

**./testbed > projectiledropped.txt**

Plot it with gnuplot from the working directory:
**gnuplot**
**set xlabel "x"**
**set ylabel "y"**
**plot "projectiledropped.txt" using 1:2 title "Cheese Ball" with lines, "projectiledropped.txt" using 3:4 title "Chocolates and Medicine" with lines**

**Figure 7.8:** *The gnuplot of the position of the supplies that are dropped from different height by each drone*

## V. Uniform Circular Motion

A particle is in uniform circular motion if it travels around a circle path or a circular arc at constant (uniform) speed. Although the speed does not vary, the particle is accelerating because the velocity changes in direction.

The velocity and acceleration vectors for uniform circular motion have constant magnitude but their directions change continuously. The velocity is always directed tangent to the circle in the direction of motion, while the acceleration is always directed radially inward, to the center of the circle. That is why it is called centripetal acceleration.

The scalar components of $\vec{v}$ can be written as

$$\vec{v} = v_x\hat{i} + v_y\hat{j} = (-v\ \sin\ \theta)\hat{i} + (v\ \cos\ \theta)\hat{j} \tag{7.10}$$

remember that since sine is an odd function and cosine is an even function, we will have $\sin(\theta) = -\sin(\theta)$ and $\cos(\theta) = \cos(\theta)$. Now, we can substitute

$$\sin\ \theta = \frac{y_p}{r}$$

$$\cos\ \theta = \frac{x_p}{r}$$

**Figure 7.9:** *The particle p moves in counter-clockwise uniform circular motion with its position and velocity $\vec{v}$ at a certain instant.*

thus

$$\begin{aligned}\vec{v} &= v_x\hat{i} + v_y\hat{j} \\ &= \left(-\frac{vy_p}{r}\right)\hat{i} + \left(\frac{vx_p}{r}\right)\hat{j}\end{aligned}$$

To find the acceleration of the particle, we must derivate the velocity with respect to time. Noting that speed $v$ and radius $r$ do not change with time, we obtain

$$\begin{aligned}\vec{a} &= \frac{d\vec{v}}{dt} \\ &= \left(-\frac{v}{r}\frac{dy_p}{dt}\right)\hat{i} + \left(\frac{v}{r}\frac{dx_p}{dt}\right)\hat{j} \\ \vec{a} &= \left(-\frac{v^2}{r}\cos\ \theta\right)\hat{i} + \left(-\frac{v^2}{r}\ \sin\ \theta\right)\hat{j}\end{aligned}$$

with $\frac{dy_p}{dt} = v_y = v\ \cos\theta$ and $\frac{dx_p}{dt} = v_x = -v\ \sin\ \theta$. Now, to calculate the acceleration magnitude, we will have

$$\begin{aligned}a &= \sqrt{a_x^2 + a_y^2} \\ &= \frac{v^2}{r}\sqrt{(\cos\ \theta)^2 + (\sin\ \theta)^2} \\ &= \frac{v^2}{r}(1) \\ a &= \frac{v^2}{r}\end{aligned}$$

Now to orient $a$, we want to prove that the angle $\phi$ will direct the acceleration toward the circle's center

$$\tan\ \phi = \frac{a_y}{a_x} = \frac{-(v^2/r)\ \sin\ \theta}{-(v^2/r)\ \cos\ \theta} = \tan\ \theta \tag{7.11}$$

Thus, $\phi = \theta$.

**Figure 7.10:** *The velocity $\vec{v}$ and acceleration $\vec{a}$ of a particle in uniform circular motion that is moving in counter-clockwise direction.*

## VI. Simulation for Uniform Circular Motion with Box2D

For the C++ simulation, you can copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Circular Motion**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CircularMotion : public Test
{
	public:

	CircularMotion()
	{
		m_world->SetGravity(b2Vec2(0.0f,0.0f));
		b2BodyDef bd;
		b2Body* ground = m_world->CreateBody(&bd);
		b2Body* b1;
		{
			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
				0.0f));
```

```
                b2BodyDef bd;
                b1 = m_world->CreateBody(&bd);
                b1->CreateFixture(&shape, 0.0f);
        }
        // Create the trail path
        {
                b2CircleShape shape;
                shape.m_radius = 3.0f;
                shape.m_p.Set(-3.0f, 25.0f);

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.isSensor = true;
                m_sensor = ground->CreateFixture(&fd);
        }

        // Create the ball
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 3.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(0.0f, 25.0f);

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);
        m_ball->SetAngularVelocity(1.0f);
        m_time = 0.0f;
}
b2Body* m_ball;
float m_time;
b2Fixture* m_sensor;
void Step(Settings& settings) override
{
        b2Vec2 v = m_ball->GetLinearVelocity();
        float r = 3.0f;
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
```

```
b2Vec2 position = m_ball->GetPosition();
float sin = sinf(angle);
float cos = cosf(angle);
float ball_vel = 3.0f;
float a = (ball_vel*ball_vel) / r;
m_time += 1.0f / 60.0f; // assuming we are using frequency of
     60 Hertz

float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
    massData.I * omega * omega;

m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
    ball_vel*cosf(angle)));

g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
    f", position.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
    f", position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
    .mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
    , ke);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
    ", v);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity, x =
    %.6f", v.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity, y =
    %.6f", v.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Acceleration, a = %.6f
    ", a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
    %.6f", angle*RADTODEG);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6f",
    sin);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6f",
```

```
                    cos);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, v.x, v.y, sin,
                    cos);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new CircularMotion;
        }

    };

    static int testIndex = RegisterTest("Motion in 2D", "Circular Motion",
        CircularMotion::Create);
```

**C++ Code 54:** *tests/circular_motion.cpp "Uniform Circular Motion Box2D"*

Some explanations for the codes:

- First, we create the physics world with gravity of 0 (not realistic on earth, but it can be used to show uniform circular motion on space), then we create the trail path for the particle that is moving, a circle shape with color of green.

```
            m_world->SetGravity(b2Vec2(0.0f,0.0f));
            b2BodyDef bd;
            b2Body* ground = m_world->CreateBody(&bd);
            b2Body* b1;
            {
                    b2EdgeShape shape;
                    shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                        0.0f));

                    b2BodyDef bd;
                    b1 = m_world->CreateBody(&bd);
                    b1->CreateFixture(&shape, 0.0f);
            }
            // Create the trail path
            {
                    b2CircleShape shape;
                    shape.m_radius = 3.0f;
                    shape.m_p.Set(-3.0f, 25.0f);

                    b2FixtureDef fd;
```

```
        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = ground->CreateFixture(&fd);
    }
```

- To create the circle (funny that we always call this circle as a ball in the code) with every details necessary, set the density, radius, restitution, friction, starting position. At the end we set the value for a new variable called **m_time** as 0 to calculate time since the beginning of the simulation.

```
    // Create the ball
    b2CircleShape ballShape;
    ballShape.m_p.SetZero();
    ballShape.m_radius = 0.5f;

    b2FixtureDef ballFixtureDef;
    ballFixtureDef.restitution = 0.75f;
    ballFixtureDef.density = 3.3f; // this will affect the ball
        mass
    ballFixtureDef.friction = 0.1f;
    ballFixtureDef.shape = &ballShape;

    b2BodyDef ballBodyDef;
    ballBodyDef.type = b2_dynamicBody;
    ballBodyDef.position.Set(0.0f, 25.0f);

    m_ball = m_world->CreateBody(&ballBodyDef);
    b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
        );
    m_ball->SetAngularVelocity(1.0f);
    m_time = 0.0f;
```

- The variable declarations

```
    b2Body* m_ball;
    float m_time;
    b2Fixture* m_sensor;
```

- We are showing some useful information on the moving particle that is in uniform circular motion, we can get the velocity and the position for the particle in $x$ and $y$ axis. The trick to make it revolving around a fixed center / in uniform circular motion, is to set the velocity to be changing with time to follow the formula from equation **(7.10)**, with $v = 3$, since the sine and cosine of an angle will always be bounded and periodic.

```
    void Step(Settings& settings) override
    {
        b2Vec2 v = m_ball->GetLinearVelocity();
        float r = 3.0f;
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
```

```
b2Vec2 position = m_ball->GetPosition();
float sin = sinf(angle);
float cos = cosf(angle);
float ball_vel = 3.0f;
float a = (ball_vel*ball_vel) / r;
m_time += 1.0f / 60.0f; // assuming we are using
    frequency of 60 Hertz

float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
    massData.I * omega * omega;

m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
     ball_vel*cosf(angle)));

g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
    = %.6f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, x
    = %.6f", position.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, y
    = %.6f", position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f",
    massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Kinetic energy =
    %.6f", ke);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
     %.6f", v);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
    x = %.6f", v.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
    y = %.6f", v.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Acceleration, a =
     %.6f", a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees
    ) = %.6f", angle*RADTODEG);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6
    f", sin);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6
```

```
                    f", cos);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                     graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, v.x, v.y,
                    sin, cos);

                Test::Step(settings);
            }
```

**Figure 7.11:** *The simulation of a particle doing circular motion with speed of $\vec{v} = 3$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/circular_motion.cpp****).*

After recompiling this, you can save it into textfile by opening the testbed with this command:
**./testbed > circularoutput.txt**
After you close the testbed to record the result, then see it at the current working directory where **testbed** is located and see **circularoutput.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only.

To plot the velocity in $x$ and $y$ axis for the uniform circular motion of the ball with respect to time, now open terminal at the directory containing the **circularoutput.txt** and type:
**gnuplot**
**set xlabel "time"**
**plot "circularoutput.txt" using 4 title "v_{x}" with lines, "circularoutput.txt" using 5 title "v_{y}" with lines**

**Figure 7.12:** *The gnuplot of the velocity of the ball particle is a negative sine function for $v_y$ and a cosine function for $v_x$*

# Chapter 8

# DFSimulatorC++ II: Force and Motion

*"Remember how it all started by Force? You(Freya) asked me to create something better than BLAS and LAPACK with FORTRAN, or sleep 24/7, but why am I writing this now?" - **DS Glanzsche***

What cause an object to move? The answer is Force. I was forced to study FORTRAN, thus I take a FORTRAN book and read it then study, there it is one good example, since the alternative is if I don't want to study FORTRAN, I better sleep 24/7, well it is not good for my soul, happiness and body to sleep 24/7 unless I am in a comma, so I read FORTRAN based on Force. But, C++ comes first before FORTRAN, as there is always a prelude of something, appetizer before the main course.

## I. Newtonian Mechanics

The relation between a force and the acceleration it causes was first understood by Isaac Newton(1642-1727), it is known as Newtonian mechanics. But, if the speed of the interacting bodies are too large, converging to the speed of light, we will use Einstein's theory of relativity instead.

The velocity of an object can change (the object can accelerate or deccelerate) when the object is acted on by one or more forces (pushes or pulls) from other objects. Newtonian mechanics relates accelerations and forces.

Forces are vector quantities. Masses are scalar quantities. Their magnitudes are defined in terms of the acceleration times the mass of the object. A force that accelerates the standard body by exactly 1 $m/s^2$ is defined to have a magnitude of 1 $N$. The direction of a force is the direction of the acceleration it causes. Forces are combined according to the rules of vector algebra. The net force on a body is the vector sum of all the forces acting on the body.

If there is no net force on a body, the body remains at rest. The body is called to be in motion when it moves in a straight line at constant speed.

Reference frames in which Newtonian mechanics holds are called inertial reference frames or inertial frames.

**Theorem 8.1: Newton's Laws**

**Newton's First Law**
If no net force acts on a body ($F_{net} = 0$), the body's velocity cannot change; that is, the body cannot accelerate.

**Newton's Second Law**
The net force on a body is equal to the product of the body's mass and its acceleration.

In equation form,

$$\vec{F}_{net} = m\vec{a} \tag{8.1}$$

The acceleration component along a given axis is caused only by the sum of the force components along that same axis, and not by force components along any other axis.

**Newton's Third Law**
When two bodies interact, the forces on the bodies from each other are always equal in magnitude and opposite in direction.

If a force $\vec{F}_{BC}$ acts on body $B$ due to body $C$, then there is a force $\vec{F}_{CB}$ on body $C$ due to body $B$:

$$\vec{F}_{BC} = \vec{F}_{CB}$$

The forces are equal in magnitude but opposite in direction.

## II. Simulation for Newton's First Law with Box2D

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class NewtonFirstlaw : public Test
{
        public:
        NewtonFirstlaw()
        {
                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
                            (-46.0f, 46.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                {
```

```
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2
            (46.0f, 46.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
            (46.0f, 0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2
            (46.0f, 12.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2
            (46.0f, 23.0f));
        ground->CreateFixture(&shape, 0.0f);
    }

    // Create the box as the movable object with friction
        0
    b2PolygonShape boxShape;
    boxShape.SetAsBox(0.5f, 0.5f);

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 7.3f; // this will affect the
        box mass
    boxFixtureDef.friction = 0.0f;
    boxFixtureDef.shape = &boxShape;
```

```
        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(5.0f, 0.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&
            boxFixtureDef);

        // Create the box 2 with friction 0.5
        b2PolygonShape boxShape2;
        boxShape2.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef2;
        boxFixtureDef2.restitution = 0.75f;
        boxFixtureDef2.density = 7.3f; // this will affect the
             box mass
        boxFixtureDef2.friction = 0.5f;
        boxFixtureDef2.shape = &boxShape2;

        b2BodyDef boxBodyDef2;
        boxBodyDef2.type = b2_dynamicBody;
        boxBodyDef2.position.Set(5.0f, 12.5f);

        m_box2 = m_world->CreateBody(&boxBodyDef2);
        b2Fixture *boxFixture2 = m_box2->CreateFixture(&
            boxFixtureDef2);

        // Create the box 3 with friction 1
        b2PolygonShape boxShape3;
        boxShape3.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef3;
        boxFixtureDef3.restitution = 0.75f;
        boxFixtureDef3.density = 7.3f; // this will affect the
             box mass
        boxFixtureDef3.friction = 1.0f;
        boxFixtureDef3.shape = &boxShape3;

        b2BodyDef boxBodyDef3;
        boxBodyDef3.type = b2_dynamicBody;
        boxBodyDef3.position.Set(5.0f, 23.5f);

        m_box3 = m_world->CreateBody(&boxBodyDef3);
        b2Fixture *boxFixture3 = m_box3->CreateFixture(&
            boxFixtureDef3);
        m_time = 0.0f;
    }
    b2Body* m_box;
```

```
b2Body* m_box2;
b2Body* m_box3;
float m_time;

void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
                m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                break;
                case GLFW_KEY_A:
                m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f))
                    ;
                m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                    );
                m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                    );
                break;
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                     true);
                m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f)
                    , true);
                m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f)
                    , true);
                break;
                case GLFW_KEY_I:
                m_box->ApplyLinearImpulseToCenter(b2Vec2(250.0f
                    , 0.0f), true);
                m_box2->ApplyLinearImpulseToCenter(b2Vec2(250.0
                    f, 0.0f), true);
                m_box3->ApplyLinearImpulseToCenter(b2Vec2(250.0
                    f, 0.0f), true);
                break;
        }
}
void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f; // assuming we are using
            frequency of 60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 position2 = m_box2->GetPosition();
        b2Vec2 position3 = m_box3->GetPosition();
        b2Vec2 velocity1 = m_box->GetLinearVelocity();
```

```
                        b2Vec2 velocity2 = m_box2->GetLinearVelocity();
                        b2Vec2 velocity3 = m_box3->GetLinearVelocity();

                        g_debugDraw.DrawString(5, m_textLine, "Time (in
                            seconds)= %.6f", m_time);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 position
                            = (%4.1f, %4.1f)", position.x, position.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity
                            = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                        m_textLine += m_textIncrement;

                        g_debugDraw.DrawString(5, m_textLine, "Box 2 position
                            = (%4.1f, %4.1f)", position2.x, position2.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity
                            = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 3 position
                            = (%4.1f, %4.1f)", position3.x, position3.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity
                            = (%4.1f, %4.1f)", velocity3.x, velocity3.y);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
                            Mass = %.6f", massData.mass);
                        m_textLine += m_textIncrement;
                        Test::Step(settings);

                        printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);

                }
                static Test* Create()
                {
                        return new NewtonFirstlaw;
                }

        };

        static int testIndex = RegisterTest("Force and Motion", "Newton's
            First Law", NewtonFirstlaw::Create);
```

**C++ Code 55:** *tests/newton_firstlaw.cpp "Newton's First Law Box2D"*

Some explanations for the codes:

- We are creating a closed 3 different grounds each at different $y$ axis value

```
        b2Body* ground = NULL;
```

```
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
                46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2(46.0f,
                12.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2(46.0f,
                23.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
```

- Next, we create three boxes with different coefficient of friction: $0, 0.5$, and $1$.

```
        // Create the box as the movable object with friction 0
```

```
            b2PolygonShape boxShape;
            boxShape.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef;
            boxFixtureDef.restitution = 0.75f;
            boxFixtureDef.density = 7.3f; // this will affect the box mass
            boxFixtureDef.friction = 0.0f;
            boxFixtureDef.shape = &boxShape;

            b2BodyDef boxBodyDef;
            boxBodyDef.type = b2_dynamicBody;
            boxBodyDef.position.Set(5.0f, 0.5f);

            m_box = m_world->CreateBody(&boxBodyDef);
            b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef);

            // Create the box 2 with friction 0.5
            b2PolygonShape boxShape2;
            boxShape2.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef2;
            boxFixtureDef2.restitution = 0.75f;
            boxFixtureDef2.density = 7.3f; // this will affect the box mass
            boxFixtureDef2.friction = 0.5f;
            boxFixtureDef2.shape = &boxShape2;

            b2BodyDef boxBodyDef2;
            boxBodyDef2.type = b2_dynamicBody;
            boxBodyDef2.position.Set(5.0f, 12.5f);

            m_box2 = m_world->CreateBody(&boxBodyDef2);
            b2Fixture *boxFixture2 = m_box2->CreateFixture(&boxFixtureDef2
                );

            // Create the box 3 with friction 1
            b2PolygonShape boxShape3;
            boxShape3.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef3;
            boxFixtureDef3.restitution = 0.75f;
            boxFixtureDef3.density = 7.3f; // this will affect the box mass
            boxFixtureDef3.friction = 1.0f;
            boxFixtureDef3.shape = &boxShape3;

            b2BodyDef boxBodyDef3;
            boxBodyDef3.type = b2_dynamicBody;
            boxBodyDef3.position.Set(5.0f, 23.5f);
```

```
m_box3 = m_world->CreateBody(&boxBodyDef3);
b2Fixture *boxFixture3 = m_box3->CreateFixture(&boxFixtureDef3
    );
m_time = 0.0f;
```

Don't forget to declare the variable of the boxes, and the time as well (**m_box, m_box2, m_box3, m_time**).

- We create keyboard press events when we press "A" the boxes will suddenly have velocity of 10, and when we press "D" the boxes will have velocity of $-10$ (in the direction of negative $x$ axis). If we press "F" then all the boxes will be given force toward the negative $x$ axis with power of 250, while if we press "G" the boxes will be given force toward the positive $x$ axis with power of 250, you can notice that box with highest friction barely move with such high force. Learn the difference of setting constant velocity on an object or particle with giving it some kind of force from the rest state.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
                m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
                break;
                case GLFW_KEY_A:
                m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                break;
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f),
                     true);
                m_box2->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                    , true);
                m_box3->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                    , true);
                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                    true);
                m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                    true);
                m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                    true);
                break;
        }
}
```

- Next, we will showing some position data of all the boxes, all of them share the same mass.

As usual, in the end we print the velocity of box 1, so that we can plot the graph of the velocity and the acceleration.

```
void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f; // assuming we are using
            frequency of 60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 position2 = m_box2->GetPosition();
        b2Vec2 position3 = m_box3->GetPosition();
        b2Vec2 velocity1 = m_box->GetLinearVelocity();
        b2Vec2 velocity2 = m_box2->GetLinearVelocity();
        b2Vec2 velocity3 = m_box3->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
            = %.6f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 position =
            (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity =
            (%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Box 2 position =
            (%4.1f, %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity =
            (%4.1f, %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 3 position =
            (%4.1f, %4.1f)", position3.x, position3.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity =
            (%4.1f, %4.1f)", velocity3.x, velocity3.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
            Mass = %.6f", massData.mass);
        m_textLine += m_textIncrement;
        Test::Step(settings);

        printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);

}
```

Now, we can plot the velocity of the box with gnuplot, recompile the testbed to make the change occurs then type:

**Figure 8.1:** *The simulation of three boxes being pulled to the left at the speed of $\vec{v} = 10$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/newton_firstlaw.cpp****).*

**./testbed > newtonfirstoutput.txt**

Plot it with gnuplot from the working directory:

```
gnuplot
set xlabel "time"
set ylabel ""
delta_v(x) = (vD = x - old_v, old_v = x, vD)
old_v = NaN
plot "newtonfirstoutput.txt" using 1 with lines title "v_x", "newtonfirstoutput.txt" using0:(delta_v($1))
with lines title "a_{x}"
```

Figure it would be trickier to plot the acceleration, compared to the previous chapter. We have velocity in $x$ axis, then we remember that instantaneous acceleration is defined as

$$\vec{a} = \frac{d\vec{v}}{dt}$$

and the average acceleration during $\triangle t$ is:

$$\vec{a}_{avg} = \frac{v_2 - v_1}{\triangle t}$$

Thus while the box is moving at constant velocity, the acceleration is read as 0 in the graph.

**Figure 8.2:** *The gnuplot of the velocity and the acceleration of box 1 that is being push with force of* 250 *toward positive x axis then hit the wall and go back then push with force of* 250 *again toward positive x axis.*

## III. Simulation for Newton's First Law and 2 Dimensional Forces with Box2D

We are going to simulate the movement of a box with forces from different directions:

1. South West
2. North East
3. North West
4. South East

Remember carefully that

$$\cos(\theta) = \cos(-\theta)$$

and

$$\cos(\theta) = -\cos(180^0 - \theta)$$

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class NewtonFirstlaw2dforces : public Test
{
        public:
        NewtonFirstlaw2dforces()
```

**Figure 8.3:** *The schemes of forces that are coming from 4 different directions, a force from North West is coming by pressing "E", a force from South West is coming by pressing "C" and both will push the box to the right. A force from South East is coming by pressing "B", a force from North East is coming by pressing "T" and both will push the box to the left. This is more of a geometry trick.*

```
{
        m_world->SetGravity(b2Vec2(0.0f,-9.8f));
        b2Body* ground = NULL;
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f
                    , 23.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                    23.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);
```

```
                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                    0.0f));
                ground->CreateFixture(&shape, 0.0f);
        }

        // Create the box as the movable object with friction 0.3
        b2PolygonShape boxShape;
        boxShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 3.3f; // this will affect the box
            mass
        boxFixtureDef.friction = 0.3f;
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(5.0f, 0.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
            ;

        m_time = 0.0f;
    }
    b2Body* m_box;
    float m_time;
    int theta = 30;
    float deginrad = theta*DEGTORAD;
    float deginrad2 = (180-theta)*DEGTORAD;
    void Keyboard(int key) override
    {
        switch (key)
        {
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f), true);

                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f), true
                    );
                break;
                case GLFW_KEY_E:
                m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                    deginrad), 0.0f), true);
                break;
```

```
                        case GLFW_KEY_C:
                        m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad)
                            , 0.0f), true);
                        break;
                        case GLFW_KEY_T:
                        m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                            deginrad2), 0.0f), true);
                        break;
                        case GLFW_KEY_B:
                        m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad2
                            ), 0.0f), true);
                        break;
                }
        }
        void Step(Settings& settings) override
        {
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz
                b2MassData massData = m_box->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                b2Vec2 velocity1 = m_box->GetLinearVelocity();

                g_debugDraw.DrawString(5, m_textLine, "Press E, C, T, or B
                    for adding force from 4 different directions");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
                     %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
                     %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "cos(150) = %.6f", cosf
                    (150*DEGTORAD));
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "cos(-150) = %.6f",
                    cosf(-150*DEGTORAD));
                m_textLine += m_textIncrement;
                Test::Step(settings);

                printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
        }
```

```
        static Test* Create()
        {
                return new NewtonFirstlaw2dforces;
        }

    };

    static int testIndex = RegisterTest("Force and Motion", "Newton's First Law
         2D Forces", NewtonFirstlaw2dforces::Create);
```

**C++ Code 56:** *tests/newton_firstlaw2dforces.cpp "Newton's First Law and 2D Forces Box2D"*

**Figure 8.4:** *The simulation of a box that we can push from 4 different directions of South West, North West, North East and South East, we set the friction to 0.3 thus the box will stop eventually (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/newton_firstlaw2dforces.cpp****).*

Some explanations for the codes:

- Create a box as dynamic body with friction 0.3 and will be located at $x = 5,\ y = 0.5$, it will fell to the ground at $y = 0$ due to the gravity.

```
        // Create the box as the movable object with friction 0.3
        b2PolygonShape boxShape;
        boxShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 3.3f; // this will affect the box mass
```

```
boxFixtureDef.friction = 0.3f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef);

m_time = 0.0f;
```

- Declare class variable for the box, the time and the degree of the forces, then we convert it into radian.

```
b2Body* m_box;
float m_time;
int theta = 30;
float deginrad = theta*DEGTORAD;
float deginrad2 = (180-theta)*DEGTORAD;
```

- To manage the keyboard press events,
  1. When we press "F", a force of 250 $N$ is going toward the positive $x$ direction.
  2. When we press "G", a force of 250 $N$ is going toward the negative $x$ direction, thus the force is written as negative.
  3. When we press "E", a force of 250 $N$ is coming from the North West direction going toward South East. Thus, we multiply the force with cosine of the angle between the force and the positive $x$ axis. The **cosf(-deginrad)** means we are computing the degree in clockwise direction, since the positive degree should represent computing degree in counter clockwise direction. Imagine that we are computing the degree from from positive $x$ axis, use vector $(1, 0)$, then rotate it clockwise direction till it become $\vec{F_1}$.
  4. When we press "C", a force of 250 $N$ is coming from the South West direction going toward North East. Thus, we multiply the force with cosine of the angle between the force and the positive $x$ axis. The **cosf(deginrad)** means we are computing the degree in counter clockwise direction. Imagine that we are computing the degree from from positive $x$ axis, use vector $(1, 0)$, then rotate it counter clockwise direction till it become $\vec{F_2}$.
  5. When we press "B", a force of 250 $N$ is coming from the South East direction going toward North West. Thus, we multiply the force with cosine of 180 - the angle between the force and the positive $x$ axis. The **cosf(deginrad2)** means we are computing the degree from from positive $x$ axis, since we are using the input of positive vector force ($F = 250$, **ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad2), 0.0f), true)**), thus use vector $(1, 0)$, then rotate it counter clockwise direction till it become $\vec{F_3}$ that is going toward North West. Remember that cosine on the second quadrant ($90 \leq \theta \leq 180$) is always negative, thus when we compute **cosf(deginrad2)**, we will get negative number times the force that is positive, will become negative, meaning the force coming from South East will push the box to go toward the negative $x$ axis.

6. When we press "T", a force of 250 $N$ is coming from the North East direction going toward South West. Thus, we multiply the force with cosine of 180 - the angle between the force and the positive $x$ axis. The **cosf(-deginrad2)** means we are computing the degree from from positive $x$ axis, since we are using the input of positive vector force ($F = 250$, **ApplyForceToCenter(b2Vec2(250.0f*cosf(-deginrad2), 0.0f), true)**), thus use vector $(1, 0)$, then rotate it clockwise direction till it become $\vec{F_4}$ that is going toward South West. Hence, we will get negative number times the force that is positive, will become negative, meaning the force coming from North East will push the box to go toward the negative $x$ axis.

   This command:
   **ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad2), 0.0f), true)**
   is equal with
   **ApplyForceToCenter(b2Vec2(-250.0f*cosf(deginrad), 0.0f), true)**
   with **deginrad =** $30^0$ and **deginrad2 =** $150^0$

$$F \times \cos(30) = -F \times \cos(150)$$

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f),
                     true);
                break;
                case GLFW_KEY_E:
                m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                    deginrad), 0.0f), true);
                break;
                case GLFW_KEY_C:
                m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(
                    deginrad), 0.0f), true);
                break;
                case GLFW_KEY_T:
                m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                    deginrad2), 0.0f), true);
                break;
                case GLFW_KEY_B:
                m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(
                    deginrad2), 0.0f), true);
                break;
        }
}
```

- To print the time, mass, position and velocity of the box on the GUI

```
void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f; // assuming we are using
            frequency of 60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity1 = m_box->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Press E, C, T, or
             B for adding force from 4 different directions");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
            = %.6f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box position =
            (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box velocity =
            (%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
             massData.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "cos(150) = %.6f",
             cosf(150*DEGTORAD));
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "cos(-150) = %.6f"
            , cosf(-150*DEGTORAD));
        m_textLine += m_textIncrement;
        Test::Step(settings);

        printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
}
```

## IV. Some Particular Forces

[DF*] **The Gravitational Force**

A gravitational force $\vec{F_g}$ on a body is a certain type of pull that is directed toward a second body. Thus, $\vec{F_g}$ means a force that pulls that body toward the center of the earth. We shall assume that the ground is an inertial frame. The magnitude of the force is

$$F_g = mg \tag{8.2}$$

The same gravitational force still acts on the body even when the body is not in free fall, or at rest.

[DF*] **Weight**

The weight $W$ of a body is the magnitude of the upward force needed to balance the gravitational force on the body. A body's weight is related to the body's mass by

$$W = mg \tag{8.3}$$

[DF*] **The Normal Force**

A normal force $\vec{F_N}$ is the force on a body from a surface against which the body presses. The normal force is always perpendicular to the surface.

If there is a block of mass $m$ presses down on a table, deforming the table because of the gravitational force $F_g$ on the block, then we can write Newton's second law for a positive $y$ axis as

$$\begin{aligned} F_N - F_g &= ma_y \\ F_N - mg &= ma_y \\ F_N &= mg + ma_y \\ F_N &= m(g + a_y) \end{aligned}$$

If the block is not accelerated then $a = 0$ and $F_N = mg$. The acceleration can be toward positive $y$ axis or negative $y$ axis.

[DF*] **Friction**

If we attempt to slide a body on a surface, the motion is resisted by bonding between the body and the surface. That's why after we push a box will stop, but if the surface is smooth or frictionless, it can keeps going, like sliding a puck on an ice rink.

[DF*] **Tension**

When a cord (or a rope, cable) is attached to a body and pulled taut, the cord pulls on the body with a force $\vec{T}$ directed away from the body and along the cord.

For a massless cord (a cord with negligible mass), the pulls at both ends of the cord have the same magnitude $T$, even if the cord runs around a massless, frictionless pulley (a pulley with negligible mass and negligible friction on its axle to oppose its rotation).

## V. Simulation for Applying Newton's Law with Box2D

I took the example from chapter 5′ problem no 57 of [6]. Why? this reminds me of my high school year physics, Now we can simulate how the block moves, it is amazing, like a dream comes true.

Here it is the problem:
A box of mass $m_1 = 3.70\ kg$ on a frictionless plane inclined at angle $\theta = 30^0$ is connected by a cord over a massless, frictionless pulley to a second box of mass $m_2 = 2.30\ kg$. What are

(a) The magnitude of the acceleration of each box

(b) The direction of the acceleration of the hanging box

(c) The tension in the cord

**Figure 8.5:** *A box with mass $m_1$ on inclined plane of $\theta = 30^0$, is connected by a cord to hanging box on the right with mass $m_2$ (T is the tension in the cord). The middle picture is the body diagram of the first box, while the right picture is the body diagram of the second box.*

**Solution:**

(a) We take the positive $x$ direction to be up the incline and the positive $y$ direction to be in the direction of the normal force $\vec{F}_N$ that the plane exerts on the box.

For the second box, we take the positive $y$ direction to be down. In this way, the accelerations of the two blocks can be represented by the same symbol $a$, without ambiguity. Applying Newton's second law to the $x$ and $y$ axes for box 1 and to the $y$ axis of box 2, we obtain

$$\begin{aligned} \sum F_{x_1} &= 0 \\ T_1 - m_1 g\ \sin\ \theta &= m_1 a \\ \sum F_{y_1} &= 0 \\ F_N - m_1 g\ \cos\ \theta &= 0 \\ \sum F_{y_2} &= 0 \\ m_2 g - T_2 &= m_2 a \end{aligned}$$

with $F_{N_2} = T_2$, the second equation $\sum F_{y_1} = F_N - m_1 g\ \cos\ \theta = 0$ is not needed in this problem, since the normal force is neither asked nor needed as part of the computation. We only use the first and third equations to obtain the values of $a$ and $T$.

The tension for the cord connecting the first and second box will be the same, $T_1 = T_2 = T$, by applying Newton's laws, we add the first and third equations:

$$\begin{aligned} T_1 &= T_2 \\ m_2 g - m_1 g \ \sin\ \theta &= m_1 a + m_2 a \\ (m_2 - m_1 \ \sin\ \theta) g &= (m_1 + m_2) a \\ a &= \frac{(m_2 - m_1 \ \sin\ \theta) g}{m_1 + m_2} \\ a &= 0.735 \end{aligned}$$

the magnitude of the acceleration of the box is 0.735 $m/s^2$

(b) Since the value of $a$ is positive, the direction of the acceleration of the hanging box is indeed up the incline plane and bringing box 2 vertically down.

(c) After we obtain the value of $a$, we can substitute to the equation 1 or 3 from part (a), thus

$$\begin{aligned} T &= m_1 a + m_1 g \ \sin\ \theta \\ &= (3.70)(0.735) + (3.70)(9.80) \sin\ 30^0 \\ T &= 20.8 \end{aligned}$$

the tension in the cord is 20.8 $N$.

Now, the fun begins when we can see it simulated with computer, C++ and Box2D save our day.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class PulleyJointTriangle : public Test
{
        public:
        PulleyJointTriangle()
        {
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                float L = 10.0f;
                float a = 1.0f;
                float b = 1.5f;

                b2Body* ground = NULL;
                {// Create the pulley
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2CircleShape circle;
                        circle.m_radius = 1.0f;

                        circle.m_p.Set(0.0f, b + L); // circle with center of
                            (0,b+L)
```

```
        ground->CreateFixture(&circle, 0.0f);
    }
    // Create the triangle
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
        (0,10)};
    chainShape.CreateLoop(vertices, 3);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    {
        // Create the box on a triangle
        b2PolygonShape boxShape;
        boxShape.SetAsBox(a, b); // width and length of the
            box

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 0.616985f; // this will affect
             the box mass
        boxFixtureDef.friction = 0.0f; // frictionless plane
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(-3.0f, L);
        // boxBodyDef.fixedRotation = true;

        m_boxl = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_boxl->CreateFixture(&
            boxFixtureDef);

        // Create the box hanging on the right
        b2PolygonShape boxShape2;
        boxShape2.SetAsBox(a, b); // width and length of the
            box

        b2FixtureDef boxFixtureDef2;
        boxFixtureDef2.restitution = 0.75f;
```

```
                        boxFixtureDef2.density = 0.3835f; // this will affect
                            the box mass, mass = density*5.9969
                        boxFixtureDef2.friction = 0.3f;
                        boxFixtureDef2.shape = &boxShape;

                        b2BodyDef boxBodyDef2;
                        boxBodyDef2.type = b2_dynamicBody;
                        boxBodyDef2.position.Set(3.0f, L);
                        // boxBodyDef2.fixedRotation = true;

                        m_boxr = m_world->CreateBody(&boxBodyDef2);
                        b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
                            boxFixtureDef2);

                        // Create the Pulley
                        b2PulleyJointDef pulleyDef;
                        b2Vec2 anchor1(-3.0f, L-1.5f); // the position of the
                             end string of the left cord is (-3.0f,L-1.5)
                        b2Vec2 anchor2(2.0f, L); // the position of the end
                            string of the right cord is (2.0f,L)
                        b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
                            the cord is tightened at (-1.0f,b+L)
                        b2Vec2 groundAnchor2(1.5f, b + L); // the string of
                            the cord is tightened at (1.5f, b+L)
                        // the last float is the ratio
                        pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
                            groundAnchor2, anchor1, anchor2, 1.0f);

                        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
                            pulleyDef);
                }
        }
        b2Body* m_boxl;
        b2Body* m_boxr;
        b2PulleyJoint* m_joint1;
        void Step(Settings& settings) override
        {
                Test::Step(settings);
                b2MassData massData1 = m_boxl->GetMassData();
                b2Vec2 position1 = m_boxl->GetPosition();
                b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
                b2MassData massData2 = m_boxr->GetMassData();
                b2Vec2 position2 = m_boxr->GetPosition();
                b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
                float m1 = massData1.mass;
                float m2 = massData2.mass;
                float g = 9.8f;
```

```
                float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
                float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

                g_debugDraw.DrawString(5, m_textLine, "Left Box position =
                    (%4.1f, %4.1f)", position1.x, position1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
                    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
                     massData1.mass);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Right Box position =
                    (%4.1f, %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
                    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
                    , massData2.mass);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
                    a);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The tension of the
                    cord = %.6f", T);
                m_textLine += m_textIncrement;

                float ratio = m_joint1->GetRatio();
                float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
                    GetCurrentLengthB();
                g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
                    f", (float) ratio, (float) L);
                m_textLine += m_textIncrement;
        }

        static Test* Create()
        {
                return new PulleyJointTriangle;
        }
};

static int testIndex = RegisterTest("Force and Motion", "Pulley Tension 1",
     PulleyJointTriangle::Create);
```

**C++ Code 57:** *tests/pulley_jointtriangle.cpp "Applying Newton's Law Box2D"*

Now go figures, when we play game with amazing graphics and physics engine like GTA series, Star Ocean, The Quarry, we then realize details are very important, to make it more realistic, you need to add more physics in every single objects in the game. The book does not state the length and width of the box, thus I input whatever I think is correct in the C++ codes, now when you run the simulation the boxes are rolling first at the initial state, then stabilize. By putting the masses as stated, box 1 has no friction with the inclined plane, the simulation is correct, the box 1 is being pulled up and the box 2 is going down even if the mass of box 2 is less than that of box 1. The equation can explain the world look-like / the graphics / the geometry of our system. Algebra solves first, then we can construct the geometry view.

If you try to pull the box 1 down the plane, with mouse click, it will sliding upward again.

**Figure 8.6:** *The simulation of a box on an inclined plane of $\theta = 30^0$ being connected with a cord and a pulley to another box on the right hanging at the right side of the plane (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_jointtriangle.cpp****).*

Some explanations for the codes:

- To create the physics world with gravity going toward negative $y$ axis of magnitude 9.8, the float variables $a$ and $b$ are to define the width and length of the boxes, while $L$ is used to define the $y$ positions of the cord connected to the left box and the right box, as well to the position of the cord where it will be tightened and connected to each of the box.

```
m_world->SetGravity(b2Vec2(0.0f,-9.8f));
float L = 10.0f;
float a = 1.0f;
float b = 1.5f;
```

- Create the pulley with a shape of a circle.

```
b2Body* ground = NULL;
{// Create the pulley
       b2BodyDef bd;
       ground = m_world->CreateBody(&bd);

       b2CircleShape circle;
       circle.m_radius = 1.0f;

       circle.m_p.Set(0.0f, b + L); // circle with center of
           (0,b+L)
       ground->CreateFixture(&circle, 0.0f);
}
```

- Create the inclined plane that has a shape of a triangle.

```
// Create the triangle
b2ChainShape chainShape;
b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
    (0,10)};
chainShape.CreateLoop(vertices, 3);

b2FixtureDef groundFixtureDef;
groundFixtureDef.density = 0;
groundFixtureDef.shape = &chainShape;

b2BodyDef groundBodyDef;
groundBodyDef.type = b2_staticBody;

b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
    groundFixtureDef);
```

- Create the boxes and then the PulleyJoint, we are connecting 2 boxes, 2 pulley' anchor, and 2 pulley' ground anchor.

  The **groundAnchor()** is used to show the pixel position of the cord where it is tightened or become static before connected to the box. While the **anchor()** is used to define pixel position for the end string of the cord to the box.

**Figure 8.7:** *If the ratio is 2, then* ***length1*** *will vary at twice the rate of* ***length2****. The force in the rope attached to* ***body1*** *will have half the constraint force as the rope attached to* ***body2****.*

```
        {
            // Create the box on a triangle
            b2PolygonShape boxShape;
            boxShape.SetAsBox(a, b); // width and length of the box

            b2FixtureDef boxFixtureDef;
            boxFixtureDef.restitution = 0.75f;
            boxFixtureDef.density = 0.616985f; // this will affect
                the box mass
            boxFixtureDef.friction = 0.0f; // frictionless plane
            boxFixtureDef.shape = &boxShape;

            b2BodyDef boxBodyDef;
            boxBodyDef.type = b2_dynamicBody;
            boxBodyDef.position.Set(-3.0f, L);
            // boxBodyDef.fixedRotation = true;

            m_box1 = m_world->CreateBody(&boxBodyDef);
            b2Fixture *boxFixture = m_box1->CreateFixture(&
                boxFixtureDef);

            // Create the box hanging on the right
            b2PolygonShape boxShape2;
            boxShape2.SetAsBox(a, b); // width and length of the box

            b2FixtureDef boxFixtureDef2;
            boxFixtureDef2.restitution = 0.75f;
            boxFixtureDef2.density = 0.3835f; // this will affect
                the box mass, mass = density*5.9969
            boxFixtureDef2.friction = 0.3f;
            boxFixtureDef2.shape = &boxShape;

            b2BodyDef boxBodyDef2;
```

```
        boxBodyDef2.type = b2_dynamicBody;
        boxBodyDef2.position.Set(3.0f, L);
        // boxBodyDef2.fixedRotation = true;

        m_boxr = m_world->CreateBody(&boxBodyDef2);
        b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
            boxFixtureDef2);

        // Create the Pulley
        b2PulleyJointDef pulleyDef;
        b2Vec2 anchor1(-3.0f, L-1.5f); // the position of the
            end string of the left cord is (-3.0f,L-1.5)
        b2Vec2 anchor2(2.0f, L); // the position of the end
            string of the right cord is (2.0f,L)
        b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
            the cord is tightened at (-1.0f,b+L)
        b2Vec2 groundAnchor2(1.5f, b + L); // the string of the
            cord is tightened at (1.5f, b+L)
        // the last float is the ratio
        pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef);
    }
```

A pulley is used to create an idealized pulley. It connects two bodies to ground and to each other. As one body goes up, the other goes down.

- To show the mass, position, linear velocity, acceleration of the box, and the tension in the cord.

```
    void Step(Settings& settings) override
    {
        Test::Step(settings);
        b2MassData massData1 = m_boxl->GetMassData();
        b2Vec2 position1 = m_boxl->GetPosition();
        b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float m1 = massData1.mass;
        float m2 = massData2.mass;
        float g = 9.8f;

        float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
        float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

        g_debugDraw.DrawString(5, m_textLine, "Left Box position
             = (%4.1f, %4.1f)", position1.x, position1.y);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box velocity
                     = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box Mass =
                    %.6f", massData1.mass);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Right Box
                    position = (%4.1f, %4.1f)", position2.x, position2.y
                    );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Box
                    velocity = (%4.1f, %4.1f)", velocity2.x, velocity2.y
                    );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Box Mass =
                    %.6f", massData2.mass);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Acceleration =
                    %.6f", a);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The tension of
                    the cord = %.6f", T);
                m_textLine += m_textIncrement;

                float ratio = m_joint1->GetRatio();
                float L = m_joint1->GetCurrentLengthA() + ratio *
                    m_joint1->GetCurrentLengthB();
                g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 =
                     %4.2f", (float) ratio, (float) L);
                m_textLine += m_textIncrement;
            }
```

## VI. Friction, Drag Force, and Centripetal Force

[DF*] Frictional forces are unavoidable in our daily lives. It can stop every moving object and bring to a halt every rotating wheel, given a sufficient friction coefficient. If friction is absent, everything will be on the loose, like a puck sliding non-stop till forever on an ice rink.

[DF*] When a force $\vec{F}$ tends to slide a body along a surface, a frictional force from the surface acts on the body. The frictional force is parallel to the surface and directed so as to oppose the sliding. It is due to bonding between the atoms on the body and the atoms on the surface, an effect called cold-welding.

[DF*] If the body does not slide, the frictional force is a static frictional force $\vec{f}_s$. If there is sliding, the frictional force is a kinetic frictional force $\vec{f}_k$.

[DF*] Properties of Friction:

1. If a body does not move, the static frictional force, $\vec{f_s}$ and the component of $\vec{F}$ parallel to the surface are equal in magnitude, and $\vec{f_s}$ is directed opposite that component. If the component increases, $\vec{f_s}$ also increases.
2. The magnitude of $\vec{f_s}$ has a maximum value $f_{s,\text{max}}$ / maximum static frictional force, that is given by
$$f_{s,\text{max}} = \mu_s F_N \tag{8.4}$$
where $\mu_s$ is the coefficient of static friction and $F_N$ is the magnitude of the normal force. If the component of $\vec{F}$ parallel to the surface exceeds $f_{s,\text{max}}$, the static friction is overwhelmed and the body slides on the surface. In other words, a force that is parallel to the surface should be bigger than the coefficient of static friction times the normal force to make the body move from its' rest position.
3. If the body begins to slide on the surface, the magnitude of the frictional force rapidly decreases to a constant value $f_k$ given by
$$f_k = \mu_k F_N \tag{8.5}$$
In general,$\mu_s \geq \mu_k$, thus it is harder to move a stationary object than it is to keep a moving object in motion.

[DF*] Static friction is subtle because the static friction force is variable and depends on the external forces acting on an object. That is, $f_s \leq \mu_s N$, while $(f_{s,\text{max}}) = \mu_s N$.

[DF*] When an applied force has overwhelmed the static frictional force. The object slides and accelerates.

## VII. Simulation for Static Frictional Forces with Box2D

I took the example from chapter 6' sample problem no 6.01 of [6].

Suppose there is a force coming from North West / tilted applied force. The force and the positive $x$ axis makes an angle of $30^0$, the magnitude of the force is 50 $N$. The box has a mass of 3.3 kg, the coefficient of static friction between the box and the ground gloor is $\mu_s = 0.3$. What is the minimum magnitude of force needed to move the box from its' rest position?

**Figure 8.8:** *Angled force is applied to an initially stationary box from North West direction. These are the vertical force components: $\vec{F}_N$, $\vec{F}_g$ and $\vec{F}_y$*

**Solution:**

**Figure 8.9:** *The horizontal force components will include $\vec{f}_s$ and $\vec{F}_x$.*

To see if the block slides, we must compare the applied force component $F_x$ with the maximum magnitude $f_{s,\text{max}}$ that the static friction can have. We see that

$$\begin{aligned} F_x &= F\ \cos\ \theta \\ &= 50(\cos\ 30) \\ F_x &= 43.30 \end{aligned}$$

To calculate $f_{s,\text{max}}$, we need the magnitude $F_N$ of the normal force. Thus,

$$\begin{aligned} \sum F_y &= ma_y \\ F_N - F_g - F\ \sin\ \theta &= ma_y \\ F_N &= mg + F\ \sin\ \theta + m(0) \\ F_x &= mg + F\ \sin\ \theta \end{aligned}$$

Now,we can evaluate $f_{s,\text{max}}$,

$$\begin{aligned} f_{s,\text{max}} &= \mu_s F_N \\ &= (0.3)(mg + F\ \sin\ \theta) \\ &= (0.3)((3.3)(9.8) + 50 \sin\ 30^0) \\ &= 34.702 \end{aligned}$$

The minimum force needed to push the box to move is 34.702 $N$, now since the force is 50 $N$ tilted with angle of $30^0$, we are going to use $F_x$ and see whether it will move or not (when $F_x - f_{s,\text{max}} > 0$ the box will move),

$$\begin{aligned} \sum F_x &= ma_x \\ F_x - f_{s,\text{max}} &= 43.30 - 34.702 \\ 43.30 - 34.702 &> 0 \end{aligned}$$

The box will move with that force, since we will obtain $a_x > 0$. Now, we can try to prove the physics equation above for another case, if we push the box with another force that is parallel toward the surface, it will be easier to make the box move, as the $f_{s,\text{max}}$ will be smaller due to $\sin\ \theta = 0$ for parallel force. We can see it from the simulation with Box2D.

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
```

```
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class FrictionalForces : public Test
{
        public:
        FrictionalForces()
        {
                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f
                            , 46.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                            46.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                            0.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                // Create the box as the movable object with friction 0.3
                b2PolygonShape boxShape;
                boxShape.SetAsBox(0.5f, 0.5f);

                b2FixtureDef boxFixtureDef;
                boxFixtureDef.restitution = 0.75f;
                boxFixtureDef.density = 3.3f; // this will affect the box
                    mass
                boxFixtureDef.friction = 0.3f;
                boxFixtureDef.shape = &boxShape;
```

```
        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(5.0f, 0.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
            ;

        m_time = 0.0f;
    }
    b2Body* m_box;
    float m_time;
    float F = 50.0f;
    int theta = 30;
    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
            m_box->ApplyForceToCenter(b2Vec2(F*cosf(theta*
                DEGTORAD), F*sinf(theta*DEGTORAD)), true);
            break;
            case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(50.0f, 0.0f), true);

            break;
            case GLFW_KEY_D:
            m_box->ApplyForceToCenter(b2Vec2(100.0f, 0.0f), true);

            break;
            case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(150.0f, 0.0f), true);

            break;
            case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f), true);

            break;
        }
    }
    void Step(Settings& settings) override
    {
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
            60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity1 = m_box->GetLinearVelocity();
        float m = massData.mass;
```

```
                float fs_max = (0.3 * m * 9.8) + (F*sinf(theta*DEGTORAD));

                g_debugDraw.DrawString(5, m_textLine, "Press A to push the
                    box with magnitude 50 N at a downward angle of 30 degree"
                    );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Press S, D, F, G to
                    push the box with magnitude 50 N, 100 N, 150 N, 250 N
                    respectively, with direction toward positive x axis");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The maximum static
                    frictional force (f_{s,max})= %.6f", fs_max);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 position = (%4.1
                    f, %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity = (%4.1
                    f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;
                Test::Step(settings);

                printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
        }
        static Test* Create()
        {
                return new FrictionalForces;
        }

    };

    static int testIndex = RegisterTest("Force and Motion", "Frictional Forces"
        , FrictionalForces::Create);
```

**C++ Code 58:** *tests/frictional_forces.cpp "Static Frictional Forces Box2D"*

Some explanations for the codes:

- For the keyboard press events:
  1. If you press "A" you will push the box from North West, with tilted angle of $\theta = 30^0$ toward the positive $x$ axis direction.
  2. If you press "S" you will push the box with force of 50 $N$ parallel to the surface toward the positive $x$ axis direction.

3. If you press "D" you will push the box with force of 150 $N$ parallel to the surface toward the positive $x$ axis direction.

4. If you press "F" you will push the box with force of 150 $N$ parallel to the surface toward the positive $x$ axis direction.

5. If you press "G" you will push the box with force of 250 $N$ parallel to the surface toward the positive $x$ axis direction.

Notice that you need to push the key repeatedly to apply the force more than once so it will keep moving, since **ApplyForceToCenter()** only give force in an instant not constantly giving it power to move / translate toward $x$ and $y$ axis all the time like **SetLinearVelocity()**.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_A:
                m_box->ApplyForceToCenter(b2Vec2(F*cosf(theta*
                    DEGTORAD), F*sinf(theta*DEGTORAD)), true);
                break;
                case GLFW_KEY_S:
                m_box->ApplyForceToCenter(b2Vec2(50.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_D:
                m_box->ApplyForceToCenter(b2Vec2(100.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(150.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                    true);
                break;
        }
}
```

**Figure 8.10:** *The simulation of a box being pushed with diferent forces, first when pressing "A" with force coming from North West with a tilted angle of* $\theta = 30^0$*, then another key presses of "S", "D", "F", "G" to push the box parallel to the surface toward positive x axis with force of* $50, 100, 150, 250$ $N$ *respectively (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_jointtriangle.cpp****).*

## VIII. Simulation for Kinetic Friction with Box2D

Taken from problem no 23 chapter 6 of [6], where there are 3 boxes, one hanging on the left, one on the table, and the third box is hanging on the right, the left and center box is connected with a cord and a pulley on the left side of the table, while the center and the right box is connected with a cord and the pulley on the right side of the table.

This simulation wants to show that boxes that are moving will make our first concern toward kinetic friction, since we already pass the force to make the resting bodies move, say goodbye to static friction and hello kinetic friction.

When the three boxes are released from rest, they accelerate with a magnitude of 0.500 $m/s^2$. Box 1(on the left) has mass $M$, box 2(the center box) has mass $2M$, and box 3 (on the right) has mass $2M$. What is the coefficient of kinetic friction between box 2 and the table?

**Solution:**
Let the tensions on the strings connecting $m_2$ and $m_3$ be $T_{23}$ and that connecting $m_2$ and $m_1$ be $T_{12}$,respectively. Applying Newton's second law to the system we have

**Figure 8.11:** *The force and kinetic friction components on the system.*

For the box on the left:

$$\sum F_y = m_1 a$$
$$T_{12} - m_1 g = m_1 a$$

For the box on the table

$$\sum F_x = m_3 a$$
$$T_{23} - \mu_k m_2 g - T_{12} = m_2 a$$

For the box on the right:

$$\sum F_y = m_3 a$$
$$m_3 g - T_{23} = m_3 a$$

Since this is a system, all 3 boxes connected, we know that $a$ has one value, which is $a = 0.500$, thus by adding three equations for each boxes,

$$
\begin{aligned}
T_{12} - m_1 g + (T_{23} - \mu_k m_2 g - T_{12}) + (m_3 g - T_{23}) &= m_1 a + m_2 a + m_3 a \\
-Mg - 2\mu_k Mg + 2Mg &= 5Ma \\
2g - g - 2\mu_k g &= 5a \\
2(9.8) - 9.8 - 2(\mu_k)(9.8) &= 5(0.500) \\
\mu_k &= 0.3724
\end{aligned}
$$

Thus, the coefficient of kinetic friction in this system is $\mu_k = 0.3724$. Now for the simulation with Box2D:

```
#include <iostream>
#include "test.h"

class PulleyKineticFriction : public Test
{
	public:
	PulleyKineticFriction()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		float L = 10.0f;
		float a = 1.0f;
		float b = 1.5f;

		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shapeGround;
			shapeGround.SetTwoSided(b2Vec2(-30.0f, 0.0f), b2Vec2
				(30.0f, 0.0f));
			ground->CreateFixture(&shapeGround, 0.0f);
		}
		{// Create the pulley on the right
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2CircleShape circle;
			circle.m_radius = 1.0f;

			circle.m_p.Set(0.0f, b + L); // circle with center of
				(0,b+L)
			ground->CreateFixture(&circle, 0.0f);
		}
		{// Create the pulley on the left
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);
```

```
        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(-17.0f, b + L); // circle with center
            of (0,b+L)
        ground->CreateFixture(&circle, 0.0f);
}

// Create the table
b2ChainShape chainShape;
b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
    (0,10),b2Vec2(-17.3205,10)};
chainShape.CreateLoop(vertices, 4);

b2FixtureDef groundFixtureDef;
groundFixtureDef.density = 0;
groundFixtureDef.shape = &chainShape;

b2BodyDef groundBodyDef;
groundBodyDef.type = b2_staticBody;

b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
    groundFixtureDef);

{
        // Create the box hanging on the left
        b2PolygonShape boxShape0;
        boxShape0.SetAsBox(a/2, a/2); // width and length of
            the box

        b2FixtureDef boxFixtureDef0;
        boxFixtureDef0.restitution = 0.75f;
        boxFixtureDef0.density = 2.001034f; // this will
            affect the box mass
        boxFixtureDef0.friction = 0.0f; // frictionless plane
        boxFixtureDef0.shape = &boxShape0;

        b2BodyDef boxBodyDef0;
        boxBodyDef0.type = b2_dynamicBody;
        boxBodyDef0.position.Set(-18.0f, L-a);
        // boxBodyDef.fixedRotation = true;

        m_boxl = m_world->CreateBody(&boxBodyDef0);
        b2Fixture *boxFixture0 = m_boxl->CreateFixture(&
            boxFixtureDef0);

        // Create the box on a table
```

```
b2PolygonShape boxShape;
boxShape.SetAsBox(a, a); // width and length of the
    box

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 1.0005169f; // this will
    affect the box mass
boxFixtureDef.friction = 0.0f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-10.0f, L);
// boxBodyDef.fixedRotation = true;

m_boxc = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxc->CreateFixture(&
    boxFixtureDef);

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a,a); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 1.0005169f; // this will
    affect the box mass, mass depends on dimension too
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(3.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley on the right
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-9.1f, L+a/2); // the position of the
    end string of the left cord is (-3.0f, L-1.5)
b2Vec2 anchor2(2.0f, L); // the position of the end
    string of the right cord is (2.0f, L)
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
```

```
                    the cord is tightened at (-1.0f, b+L)
                b2Vec2 groundAnchor2(1.5f, b + L); // the string of
                    the cord is tightened at (1.5f, b+L)
                // the last float is the ratio
                pulleyDef.Initialize(m_boxc, m_boxr, groundAnchor1,
                    groundAnchor2, anchor1, anchor2, 1.0f);

                m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
                    pulleyDef);

                // Create the Pulley on the left
                b2PulleyJointDef pulleyDef1;
                b2Vec2 anchor3(-18.0f, L-1.5f);
                b2Vec2 anchor4(-11.0f, L+a/2);
                b2Vec2 groundAnchor3(-18.5f, b + L );
                b2Vec2 groundAnchor4(-15.5f, b + L);
                pulleyDef1.Initialize(m_boxl, m_boxc, groundAnchor3,
                    groundAnchor4, anchor3, anchor4, 1.0f);

                m_joint2 = (b2PulleyJoint*)m_world->CreateJoint(&
                    pulleyDef1);
        }
}
b2Body* m_boxc;
b2Body* m_boxl;
b2Body* m_boxr;
b2PulleyJoint* m_joint1;
b2PulleyJoint* m_joint2;

void Step(Settings& settings) override
{
        Test::Step(settings);
        b2MassData massData0 = m_boxl->GetMassData();
        b2Vec2 position0 = m_boxl->GetPosition();
        b2Vec2 velocity0 = m_boxl->GetLinearVelocity();
        b2MassData massData1 = m_boxc->GetMassData();
        b2Vec2 position1 = m_boxc->GetPosition();
        b2Vec2 velocity1 = m_boxc->GetLinearVelocity();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float m1 = massData1.mass;
        float m2 = massData2.mass;
        float g = 9.8f;
        float a = 0.500;
        float fk = (5*a-g)/(-2*g);

        g_debugDraw.DrawString(5, m_textLine, "Left Box position =
```

```
                    (%4.1f, %4.1f)", position0.x, position0.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
                (%4.1f, %4.1f)", velocity0.x, velocity0.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
                 massData0.mass);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "Center Box position =
                (%4.1f, %4.1f)", position1.x, position1.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Center Box velocity =
                (%4.1f, %4.1f)", velocity1.x, velocity1.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Center Box Mass = %.6f
                ", massData1.mass);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "Right Box position =
                (%4.1f, %4.1f)", position2.x, position2.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
                (%4.1f, %4.1f)", velocity2.x, velocity2.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
                , massData2.mass);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
                a);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Coefficient Kinetic
                Friction= %.6f", fk);
            m_textLine += m_textIncrement;

            float ratio = m_joint1->GetRatio();
            float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
                GetCurrentLengthB();
            g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
                f", (float) ratio, (float) L);
            m_textLine += m_textIncrement;
        }

        static Test* Create()
        {
            return new PulleyKineticFriction;
        }
```

```
    };

    static int testIndex = RegisterTest("Force and Motion", "Pulley Kinetic
        Friction", PulleyKineticFriction::Create);
```

**C++ Code 59:** *tests/pulley_kineticfriction.cpp "Pulley Kinetic Friction Box2D"*

What interesting is coefficient of kinetic friction for this system is not depending on the floating number of each mass. Since the mass cancel itself, it depends on the proportion between the masses in the system.

**Figure 8.12:** *The simulation of a physical system, with 3 boxes, one hanging on the left, one on the table, and one hanging on the right, all connected with cords and pulleys (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_kineticfriction.cpp****).*

Some explanations for the codes:

- To create the correct Pulley Joint, for the left and right pulley and the anchor as well, I need to measure the location of the box on the table, thus know the $x$ position needs to put on **anchor1()** and **anchor4()** that will connect the cord from the left (**anchor4()**) and the right (**anchor1()**) to the box on the table. Quite trial and error while coding this for Box2D.

```
            // Create the Pulley on the right
            b2PulleyJointDef pulleyDef;
            b2Vec2 anchor1(-9.1f, L+a/2); // the position of the end
                string of the left cord is (-3.0f, L-1.5)
            b2Vec2 anchor2(2.0f, L); // the position of the end string of
                the right cord is (2.0f, L)
            b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of the cord
                 is tightened at (-1.0f, b+L)
```

```
b2Vec2 groundAnchor2(1.5f, b + L); // the string of the cord is
     tightened at (1.5f, b+L)
// the last float is the ratio
pulleyDef.Initialize(m_boxc, m_boxr, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.0f);

m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

// Create the Pulley on the left
b2PulleyJointDef pulleyDef1;
b2Vec2 anchor3(-18.0f, L-1.5f);
b2Vec2 anchor4(-11.0f, L+a/2);
b2Vec2 groundAnchor3(-18.5f, b + L );
b2Vec2 groundAnchor4(-15.5f, b + L);
pulleyDef1.Initialize(m_boxl, m_boxc, groundAnchor3,
    groundAnchor4, anchor3, anchor4, 1.0f);

m_joint2 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef1);
```

Most of the codes are already explained on the previous chapters, I believe readers will already be familiar even far smarter than me and wish for something better than repeating the same explanations. We will proceed with less repetition and more substances. This is also the main reason to read step by step from beginning not jumping to chapter 10 directly.

## IX. Drag Force and Terminal Speed

A fluid is anything that can flow, either a gas or a liquid. When there is a relative motion between air (or some other fluid) and a body, the body experiences a drag force $\vec{D}$ that opposes the relative motion and points in the direction in which the fluid flows relative to the body. The magnitude of $\vec{D}$ is related to the relative speed $v$ by an experimentally determined drag coefficient $C$ according to

$$D = \frac{1}{2} C \rho A v^2 \tag{8.6}$$

where $\rho$ is the fluid density (mass per unit volume) and $A$ is the effective cross-sectional area of the body (the area of a cross section taken perpendicular to the relative velocity $\vec{v}$).

When a blunt object has fallen far enough through air, the magnitude of the drag force $\vec{D}$ and the gravitational force $\vec{F}_g$ on the body become equal. The body then falls at a constant terminal speed $v_t$ given by

$$v_t = \sqrt{\frac{2F_g}{C\rho A}} \tag{8.7}$$

## X. Simulation for Downhill Skiing with Box2D

Taken from example problem no 40 on chapter 6 of [6].

In downhill speed skiing a skier is retarded by both the air drag force on the body and the kinetic frictional force on the skis. Suppose the slope angle is $\theta = 40^0$, the snow is dry snow with

a coefficient of kinetic friction $\mu_k = 0.0400$, the mass of the skier and equipment is $m = 85.0\ kg$, the cross-sectional area of the (tucked) skier is $A = 1.30\ m^2$, the drag force coefficient is $C = 0.150$, and the air density is $1.20\ kg/m^3$.

(a) What is the terminal speed?

(b) If a skier can vary $C$ by a slight amount $dC$ by adjusting, say, the hand positions, what is the corresponding variation in the terminal speed?

**Solution:**

(a) The force along the slope is given by

$$\begin{aligned} F_g &= mg\ \sin\ \theta - \mu F_N \\ &= mg\ \sin\ \theta - \mu mg\ \cos\ \theta \\ &= mg(\sin\ \theta - \mu\ \cos\ \theta) \\ &= (85)(9.8)(\sin\ 40^0 - (0.0400)\cos\ 40^0) \\ F_g &= 510 \end{aligned}$$

Thus, the terminal speed of the skier is

$$\begin{aligned} v_t &= \sqrt{\frac{2F_g}{C\rho A}} \\ &= \sqrt{\frac{2(510)}{(0.150)(1.20)(1.30)}} \\ &= 66 \end{aligned}$$

the terminal speed is 66 $m/s$.

(b) In order to know how much speed is changed when we change $C$ for a small amount / infinitesimal amount of $dC$, we will use differentiation. Differentiating $v_t$ with respect to $C$, we obtain

$$\begin{aligned} \frac{dv_t}{dC} &= \frac{d}{dC}\sqrt{\frac{2F_g}{C\rho A}} \\ &= \frac{d}{dC}\sqrt{\frac{2F_g}{\rho A}}C^{-1/2} \\ \frac{dv_t}{dC} &= -\frac{1}{2}\sqrt{\frac{2F_g}{\rho A}}C^{-3/2} \\ dv_t &= -\frac{1}{2}\sqrt{\frac{2F_g}{\rho A}}C^{-3/2}\ dC \\ &= -\frac{1}{2}\sqrt{\frac{2(510)}{(1.20)(1.30)}}dC \\ &= -(2.20 \times 10^2)\ dC \end{aligned}$$

a slight change of $dC$ will resulting in change of terminal speed of $-(2.20 \times 10^2)\ m/s$. The minus sign means the speed is decreasing, thus the perfect body position when skiing has to

be maintained to reach highest terminal speed, and not moving too much, since not only decreasing the terminal speed, changing position or gesture while skiing might make you get into accident or hit a tree.

```
#include "test.h"
#include <iostream>

class DragforceSkier : public Test
{
	public:
	DragforceSkier()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			float const PlatformWidth = 8.0f;

			/*
			First angle is from the horizontal and should be
			    negative for a downward slope.
			Second angle is relative to the preceding slope, and
			    should be positive, creating a kind of
			loose 'Z'-shape from the 3 edges.
			If A1 = -10, then A2 <= ~1.5 will result in the
			    collision glitch.
			If A1 = -30, then A2 <= ~10.0 will result in the
			    glitch.
			*/
			float const Angle1Degrees = -40.0f;
			float const Angle2Degrees = 0.0f;

			/*
			The larger the value of SlopeLength, the less likely
			    the glitch will show up.
			*/
			float const SlopeLength = 5100.0f;

			float const SurfaceFriction = 0.04f;

			// Convert to radians
			float const Slope1Incline = -Angle1Degrees * b2_pi /
			    180.0f;
			float const Slope2Incline = Slope1Incline -
			    Angle2Degrees * b2_pi / 180.0f;
			//
```

```
        m_platform_width = PlatformWidth;

        // Horizontal platform
        b2Vec2 v1(-PlatformWidth, 0.0f);
        b2Vec2 v2(0.0f, 0.0f);
        b2Vec2 v3(SlopeLength * cosf(Slope1Incline), -
            SlopeLength * sinf(Slope1Incline));
        b2Vec2 v4(v3.x + SlopeLength * cosf(Slope2Incline), v3
            .y - SlopeLength * sinf(Slope2Incline));
        b2Vec2 v5(v4.x, v4.y - 1.0f);

        b2Vec2 vertices[5] = { v5, v4, v3, v2, v1 };

        b2ChainShape shape;
        shape.CreateLoop(vertices, 5);
        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 0.0f;
        fd.friction = SurfaceFriction;

        ground->CreateFixture(&fd);
}
float const BodyWidth = 1.0f;
float const BodyHeight = 2.5f;
float const SkiLength = 3.0f;

/*
Larger values for this seem to alleviate the issue to some
    extent.
*/
float const SkiThickness = 0.3f;
float const SkiFriction = 0.04f;
float const SkiRestitution = 0.15f;

b2BodyDef skiBodyDef;
b2Body* skier = m_world->CreateBody(&skiBodyDef);

// Create the ski geometry from vertices
b2PolygonShape ski;
b2Vec2 verts[4];
verts[0].Set(-SkiLength / 2 - SkiThickness, -BodyHeight /
    2);
verts[1].Set(-SkiLength / 2, -BodyHeight / 2 - SkiThickness
    );
verts[2].Set(SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[3].Set(SkiLength / 2 + SkiThickness, -BodyHeight / 2);
ski.Set(verts, 4);
```

```
        b2FixtureDef skiFixtureDef;
        skiFixtureDef.restitution = SkiRestitution;
        skiFixtureDef.density = 85.8586019936297f; // density = mass
            *1.010101199925053
        skiFixtureDef.friction = SkiFriction;
        skiFixtureDef.shape = &ski;

        skiBodyDef.type = b2_dynamicBody;
        float initial_y = BodyHeight / 2 + SkiThickness;
        skiBodyDef.position.Set(-m_platform_width / 2, initial_y);

        m_skier = m_world->CreateBody(&skiBodyDef);
        b2Fixture *skiFixture = m_skier->CreateFixture(&
            skiFixtureDef);
        m_skier->SetLinearVelocity(b2Vec2(4.5f, 0.0f));

        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 3.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(-m_platform_width / 2 - 1.0f,
            initial_y);

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);
        m_ball->SetAngularVelocity(1.0f);
        m_ball->SetLinearVelocity(b2Vec2(4.5f, 0.0f));

        g_camera.m_center = b2Vec2(m_platform_width / 2.0f, 0.0f);
        g_camera.m_zoom = 1.1f;
        m_fixed_camera = true;
        m_time = 0.0f;
    }
    b2Body* m_ball;
    b2Body* m_skier;
    float m_platform_width;
    float m_time;
    bool m_fixed_camera;
```

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                        g_camera.m_center = b2Vec2(m_platform_width /
                            2.0f, 0.0f);
                }
                break;
        }
}

void Step(Settings& settings) override
{
        b2Vec2 v = m_skier->GetLinearVelocity();
        b2MassData massData1 = m_skier->GetMassData();
        b2Vec2 position = m_skier->GetPosition();
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz
        float omega = m_skier->GetAngularVelocity();
        float ke = 0.5f * massData1.mass * b2Dot(v, v) + 0.5f *
            massData1.I * omega * omega;

        g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
            /tracking");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Position = (%4.1f,
            %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Velocity = (%4.1f,
            %4.1f)", v.x, v.y); // the velocity for x stop at 91.9
            then not increasing anymore
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Mass = %.1f",
            massData1.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
             , ke);
        m_textLine += m_textIncrement;

        printf("%4.2f %4.2f %4.2f %4.2f %4.2f\n", position.x,
```

```
                    position.y, v.x, v.y, ke);
                if(!m_fixed_camera)
                {
                        g_camera.m_center = m_skier->GetPosition();
                }
                Test::Step(settings);
        }

        static Test* Create()
        {
                return new DragforceSkier;
        }

};

static int testIndex = RegisterTest("Force and Motion", "Drag Force Skier",
     DragforceSkier::Create);
```

**C++ Code 60:** *tests/dragforce_skier.cpp "Drag Force Skier Box2D"*

The simulation run well, imagine you are playing ski on Mt Logan, Yukon, Canada with your beloved wife. This is just the simulation before the real life comes in. I add a ball as an extra to see what will happens if there is a ball rolling down like a snowball from top of the mountain along with a skier. Turns out the ball can be outrun by the skier, since the ball needs to rotating while moving, and by cross section $A$, ski is slimmer, thus skier makes the terminal velocity higher than the ball.

What went wrong is the velocity stop at 91.9 $m/s$ toward $x$ axis and $-77.1$ $m/s$ toward $y$ axis, this would make the velocity magnitude of 119.9584 $m/s$ toward South East / downhill.

Some explanations for the codes:

- To create the skier from vertices, in Box2D this is another way to create an object / body, besides calling for Box or Circle shape object.

```
float const BodyWidth = 1.0f;
float const BodyHeight = 2.5f;
float const SkiLength = 3.0f;

/*
Larger values for this seem to alleviate the issue to some
    extent.
*/
float const SkiThickness = 0.3f;
float const SkiFriction = 0.04f;
float const SkiRestitution = 0.15f;

b2BodyDef skiBodyDef;
b2Body* skier = m_world->CreateBody(&skiBodyDef);
```

```
// Create the ski geometry from vertices
b2PolygonShape ski;
b2Vec2 verts[4];
verts[0].Set(-SkiLength / 2 - SkiThickness, -BodyHeight / 2);
verts[1].Set(-SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[2].Set(SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[3].Set(SkiLength / 2 + SkiThickness, -BodyHeight / 2);
ski.Set(verts, 4);

b2FixtureDef skiFixtureDef;
skiFixtureDef.restitution = SkiRestitution;
skiFixtureDef.density = 85.8586019936297f; // density = mass
    *1.010101199925053
skiFixtureDef.friction = SkiFriction;
skiFixtureDef.shape = &ski;

skiBodyDef.type = b2_dynamicBody;
float initial_y = BodyHeight / 2 + SkiThickness;
skiBodyDef.position.Set(-m_platform_width / 2, initial_y);

m_skier = m_world->CreateBody(&skiBodyDef);
b2Fixture *skiFixture = m_skier->CreateFixture(&skiFixtureDef)
    ;
m_skier->SetLinearVelocity(b2Vec2(4.5f, 0.0f));
```

- A keyboard press event to track the movement of the skier.

```
void Keyboard(int key) override
{
       switch (key)
       {
              case GLFW_KEY_C:
              m_fixed_camera = !m_fixed_camera;
              if(m_fixed_camera)
              {
                     g_camera.m_center = b2Vec2(m_platform_width
                          / 2.0f, 0.0f);
              }
              break;
       }
}
```

- To print out the data of mass, velocity, and position of the skier, and when "C" is pressed the camera will follow the movement of the skier by using the **GetPosition()** function.

```
void Step(Settings& settings) override
{
       b2Vec2 v = m_skier->GetLinearVelocity();
       b2MassData massData1 = m_skier->GetMassData();
       b2Vec2 position = m_skier->GetPosition();
```

```
                m_time += 1.0f / 60.0f; // assuming we are using
                    frequency of 60 Hertz
                float omega = m_skier->GetAngularVelocity();
                float ke = 0.5f * massData1.mass * b2Dot(v, v) + 0.5f *
                    massData1.I * omega * omega;

                g_debugDraw.DrawString(5, m_textLine, "Press C = Camera
                    fixed/tracking");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
                    = %.6f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Position = (%4.1f
                    , %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Velocity = (%4.1f
                    , %4.1f)", v.x, v.y); // the velocity for x stop at
                    91.9 then not increasing anymore
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %.1f",
                    massData1.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Kinetic energy =
                    %.6f", ke);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f\n", position.x,
                    position.y, v.x, v.y, ke);
                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_skier->GetPosition();
                }
                Test::Step(settings);
            }
```

If you know the chapter in Donald Duck comic, where his cousin Didi Duck rolling fresh milk from top of the mountain to the bottom, so he can then sell the milk to the store in the village at the bottom of the mountain. Imagine that the ball in this simulation is the milk' jar, we can add more complexity and details, filled with milk or anything else, and see maximum velocity based on the cross section of the milk to preserve the milk, so the milk can still be fresh when arrived at the village.

The story from Donald Duck is a tragedy actually, what happened when milk is being rolled down from top of the mountain continuously? At the store, the shopkeeper checks, it becomes butter, thus it can't be sold. What a waste! Maybe reader of this book has ever encountered that Donald Duck chapter. A great comic, my favorite is Uncle Scrooge.

**Figure 8.13:** *The simulation of a skier going downhill with a ball on top of it, then the ball goes downhill first but the skier can catch up with the ball (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/dragforce_skier.cpp****).*

## XI. Uniform Circular Motion with Force

We have discussed about uniform circular motion, that is, a body moves in a circle (or a circular arc) at constant speed $v$, with a centripetal acceleration of

$$a = \frac{v^2}{R} \tag{8.8}$$

where $R$ is the radius of the circle. A centripetal force is not a new kind of force. It can, in fact, be a frictional force, a gravitational force, the force from a car wall or a string, or any other force.

A centripetal force accelerates a body by changing the direction of the body's velocity without changing the body's speed.

From Newton's second law, we can write the magnitude $F$ of a centripetal force as

$$F = m\frac{v^2}{R} \tag{8.9}$$

You need to remember that

- The speed $v$ is constant, the magnitudes of the acceleration and the force are also constant.
- The directions of the centripetal acceleration and force are not constant. They vary continuously so as to always point toward the center of the circle.
- The force and acceleration vetors are sometimes drawn along a radial axis $r$ that moves with the body and always extends from the center of the circle to the body.

- The positive direction of the axis is radially outward, but the acceleration and force vectors point radially inward.

## XII. Simulation for Airplane Uniform Circular Motion with Box2D

Taken from problem no 51 chapter 6 of [6].

An airplane is flying in a horizontal circle at a speed of 480 $km/h$. If its wings are tilted at angle $\theta = 40^0$ to the horizontal, what is the radius of the circle in which the plane is flying? Assume that the required force is provided by an "aerodynamic lift" that is perpendicular to the wing surface.

**Solution:**

First, you have to be able to imagine an airplane flying in a horizontal circle. Then tilted at angle $\theta = 40^0$ means that airplane is doing roll movement toward the circle' center of $\theta = 40^0$. I asked a lot of people who are the best in science, the key is their imagination is the best, they can imagine without the need for textbooks full of pictures. Isn't Einstein like this as well?

We note that $\vec{F}_1$ is the force of aerodynamic lift and $\vec{a}$ points rightwards in the figure. We also note that $|\vec{a}| = \frac{v^2}{R}$.

**Figure 8.14:** *The free-body diagram for the airplane of mass m.*

The key to comprehend this is by imagining from Figure **8.14**, on the left the airplane is shaped like that when tilted, tilted itself means roll movement toward the circle center (for uniform circular motion) about the angle of $\theta$, we play some geometry tricks then we know since it is tilted, the force is tilted as well, that's explains on the right side, where I only draw the body diagram, we need to decompose $\vec{F}_1$ into the $x$ and $y$ components to be connected with the gravitational force and centripetal force, thus able to use Newton's law. The force $\vec{F}_1$ can be seen as the normal force $\vec{F}_N$ that is tilted about the angle of $\theta$.

Moving counterclockwise or clockwise does not matter since it is only a matter of orientation, negative sign or not, the force still will be computed the same. Now, by applying Newton's law to

**Figure 8.15:** *The three dimensional airplane diagram moving in circular motion with mass of m. The creation of image is guided by Hamzst.*

the axes of the problem ($+x$ rightward and $+y$ upward), we obtain

$$\begin{aligned} \sum F_x &= 0 \\ F_1 \sin\theta - m\frac{v^2}{R} &= 0 \\ F_1 \sin\theta &= m\frac{v^2}{R} \end{aligned}$$

The equation above means the balance of the force in the horizontal axis, between $F_1 \sin\theta$ and the centripetal force that is always directed toward the circle' center for circular motion. This equation preserve the condition of circular motion since the force pulling the object inside (centripetal force) and the force pushing the object away from the circle' center is the same, thus the object(the airplane) will keep moving in circular motion with same radius.

$$\begin{aligned} \sum F_y &= 0 \\ F_1 \cos\theta - mg &= 0 \\ F_1 \cos\theta &= mg \end{aligned}$$

The equation above means the balance of the force in the vertical axis, between $F_1 \cos\theta$ and the gravitational force for the airplane. This equation preserves the height of the airplane, since the force that bring the airplane to that height is the same as the gravitational force, thus the airplane won't go higher or lower if this equation applied.

Afterwards, by eliminating mass from these equations leads to

$$\tan\theta = \frac{v^2}{gR}$$

The equation allows us to solve for the radius $R$. With $v = 480\ km/h = 133m/s$ and $\theta = 40^0$, we

find

$$\begin{aligned} R &= \frac{v^2}{g\ \tan\ \theta} \\ &= \frac{133^2}{(9.8)(\tan\ 40^0)} \\ &= 2151 \end{aligned}$$

the radius $R$ is 2151 $m$. If you are wondering why the radius is quite big? It is natural and logical since $R$ and $v$ has linear relationship. If you are using Box2D and try the circular motion simulation, try to make the velocity very high 133 for example, but let the trail path stay with small radius (around 3), what will happens? The object rotating wider and move very fast, since the speed is increasing highly. This is the basic of rocket science, satellite and beyond.

We can also say that tilted airplane will have wider radius than airplane that is moving in circular motion without tilting anywhere. For untilted airplane it will be like this:

$$\begin{aligned} \sum F_x &= 0 \\ F_{untilted} - m\frac{v^2}{R} &= 0 \\ F_{untilted} &= m\frac{v^2}{R} \end{aligned}$$

$$\begin{aligned} \sum F_y &= 0 \\ F_{untilted} - mg &= 0 \\ F_{untilted} &= mg \end{aligned}$$

$$\begin{aligned} R_{untilted} &= \frac{v^2}{g} \\ &= \frac{133^2}{9.8} \\ &= 1805 \end{aligned}$$

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CircularMotionairplane : public Test
{
	public:

	CircularMotionairplane()
	{
		m_world->SetGravity(b2Vec2(0.0f,0.0f));
		b2BodyDef bdtrail;
		b2Body* bodytrail = m_world->CreateBody(&bdtrail);

		// Create the trail path
```

```
{
        b2CircleShape shape;
        shape.m_radius = 2.1510f;
        shape.m_p.Set(-3.0f, 25.0f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = bodytrail->CreateFixture(&fd);
}
// Create shape
b2Transform xf1;
xf1.q.Set(1.3496f * b2_pi); // rotate the shape 1.3496 pi
xf1.p = xf1.q.GetXAxis();

b2Vec2 vertices[3];
vertices[0] = b2Mul(xf1, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf1, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf1, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly1;
poly1.Set(vertices, 3);

b2FixtureDef sd1;
sd1.shape = &poly1;
sd1.density = 200.0f;

b2Transform xf2;
xf2.q.Set(-1.3496f * b2_pi); // rotate the shape - 1.3496 pi
xf2.p = -xf2.q.GetXAxis();

vertices[0] = b2Mul(xf2, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf2, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf2, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly2;
poly2.Set(vertices, 3);

b2FixtureDef sd2;
sd2.shape = &poly2;
sd2.density = 200.0f;

b2BodyDef bd1;
bd1.type = b2_dynamicBody;

bd1.position.Set(-5.1540f, 24.0f);
bd1.angle = b2_pi;
bd1.allowSleep = false;
```

```
        m_body = m_world->CreateBody(&bd1);
        m_body->CreateFixture(&sd1);
        m_body->CreateFixture(&sd2);
        m_body->SetAngularVelocity(1.0f); // Counter clockwise
            movement
        // I Love my Wife!!!
        m_time = 0.0f;
    }
    b2Body* m_body;
    float m_time;
    b2Fixture* m_sensor;
    void Step(Settings& settings) override
    {
        b2Vec2 v = m_body->GetLinearVelocity();
        float r = 2.1510f;
        float omega = m_body->GetAngularVelocity();
        float angle = m_body->GetAngle();
        b2MassData massData = m_body->GetMassData();
        b2Vec2 position = m_body->GetPosition();
        float sin = sinf(angle);
        float cos = cosf(angle);
        float body_vel = 2.1510f;
        float a = (body_vel*body_vel) / r;
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz

        float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
            massData.I * omega * omega;

        m_body->SetLinearVelocity(b2Vec2(-body_vel*sinf(angle),
            body_vel*cosf(angle)));

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Airplane position =
            (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
            .mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
            , ke);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Total linear velocity=
             %.6f", v);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
```

```
                    (%4.1f, %4.1f)", v.x, v.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Acceleration, a = %.6f
                    ", a);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
                    %.6f", angle*RADTODEG);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6f",
                    sin);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6f",
                    cos);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, v.x, v.y, sin,
                    cos);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new CircularMotionairplane;
        }

};

static int testIndex = RegisterTest("Motion in 2D", "Circular Motion for
    Airplane", CircularMotionairplane::Create);
```

**C++ Code 61:** *tests/circular_motionairplane.cpp "Circular Motion Airplane Box2D"*

**Figure 8.16:** *The simulation of an airplane circling in 2-dimension (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/circular_motionairplane.cpp****).*

Some explanations for the codes:

- To create the airplane shape we are combining 3 vertices into triangle as one shape, then create similar copy. Rotate the first shape clockwise, and the other counterclockwise in the same degree. Et Voila. The function **CreateFixture()** able to join two shapes into one.

**Figure 8.17:** *The triangle shape is made out of 3 vertices, we choose it* $(1,0), (-1,0), (0,0.5)$.

```
b2Transform xf1;
xf1.q.Set(1.3496f * b2_pi); // rotate the shape 1.3496 pi
xf1.p = xf1.q.GetXAxis();

b2Vec2 vertices[3];
vertices[0] = b2Mul(xf1, b2Vec2(—1.0f, 0.0f));
vertices[1] = b2Mul(xf1, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf1, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly1;
poly1.Set(vertices, 3);
```

```
b2FixtureDef sd1;
sd1.shape = &poly1;
sd1.density = 200.0f;

b2Transform xf2;
xf2.q.Set(-1.3496f * b2_pi); // rotate the shape - 1.3496 pi
xf2.p = -xf2.q.GetXAxis();

vertices[0] = b2Mul(xf2, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf2, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf2, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly2;
poly2.Set(vertices, 3);

b2FixtureDef sd2;
sd2.shape = &poly2;
sd2.density = 200.0f;

b2BodyDef bd1;
bd1.type = b2_dynamicBody;

bd1.position.Set(-5.1540f, 24.0f);
bd1.angle = b2_pi;
bd1.allowSleep = false;
m_body = m_world->CreateBody(&bd1);
m_body->CreateFixture(&sd1);
m_body->CreateFixture(&sd2);
m_body->SetAngularVelocity(1.0f);
```

- When create the trail path for the circular movement, I set the radius to be in ratio, like when you see a map, the distance between two point in a map is a ratio of real life distance. Thus instead of making 2,151 meter of radius, I prefer to use 2.151 meter of radius. Still limited with computer monitor, one day we can use better graphics display.

```
{
        b2CircleShape shape;
        shape.m_radius = 2.1510f;
        shape.m_p.Set(-3.0f, 25.0f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = bodytrail->CreateFixture(&fd);
}
```

- The code for the simulation is not perfect in my opinion for this problem, someday I will come back to this, but for the mean time I want to proceed to other physics problem.

# Chapter 9

# DFSimulatorC++ III: Kinetic Energy and Work

*"Ouais bla.. tu me connais"* - ***Sweden Sexy (after being wild running around then comes to me and say that words)***

When we talk about kinetic energy we will think of something that moves will require energy. Everything moves, even if you sit still the earth is moving / rotating for 24 hours till it face the same side of the sun again, and revolving toward the sun for 365 days till it back to its original position on its' own orbit.

Energy can be transformed from one type to another and transferred from one object to another, but the total amount is always the same. That is the conservation law of energy.

## I. Kinetic Energy

Kinetic energy $K$ is energy associated with the state of motion of an object, the faster the object moves, the greater is its kinetic energy. When the object is stationary, its kinetic energy is zero.

The formula for kinetic energy

$$K = \frac{1}{2}mv^2 \tag{9.1}$$

with $v$ is well below the speed of light.

## II. Simulation for Kinetic Energy, Train Crash with Box2D

This is taken form sample problem 7.01 from [6].

In 1896 in Waco, Texas, William Crush parked two locomotives at opposite ends of a 6.4 km-long track, fired them up, tied their throttles, and then allowed them to crash head-on at full speed in front of 30,000 spectators. Hundreds of people were hurt by flying debris; several were killed. Assuming each locomotive weighed $1.2 \times 10^6$ $N$ and its acceleration was a constant $0.26$ $m/s^2$, what was the total kinetic energy of the two locomotives just before the collision?

**Solution:**

We can assume each locomotive had constant acceleration, to find its speed before the collision:

$$\begin{aligned} v^2 &= v_0^2 + 2a(x - x_0) \\ &= 0 + 2(0.26)(3.2 \times 10^3) \\ v &= 40.8 \end{aligned}$$

the speed before the collision is 40.8 $m/s$ or 147 $km/h$.

We can find the mass of each locomotive by

$$m = \frac{w}{g} = \frac{1.2 \times 10^6}{9.8} = 1.22 \times 10^5$$

The total kinetic energy of two locomotives just before the collision is

$$K = 2\frac{1}{2}mv^2 = (1.22 \times 10^5)(40.8)^2 = 2 \times 10^8$$

This collision with kinetic energy of 200 million Joule ($2 \times 10^8$) was like an exploding bomb.

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Kinetic Energy and Work/Kinetic Energy: Collision**.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class KineticenergyCollision : public Test
{
	public:
	KineticenergyCollision()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
```

```
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f
            , 46.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
            46.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
            0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    // Create the middle train body
    b2BodyDef bd1;
    bd1.type = b2_dynamicBody;
    bd1.angularDamping = 0.1f;

    bd1.position.Set(1.0f, 0.5f);
    b2Body* springbox = m_world->CreateBody(&bd1);

    b2PolygonShape shape1;
    shape1.SetAsBox(0.5f, 0.5f);
    springbox->CreateFixture(&shape1, 5.0f);
    // Create the last train body
    b2BodyDef bd2;
    bd2.type = b2_dynamicBody;
    bd2.angularDamping = 0.1f;

    bd2.position.Set(-1.0f, 0.5f);
    b2Body* springbox2 = m_world->CreateBody(&bd2);

    b2PolygonShape shape2;
    shape2.SetAsBox(0.5f, 0.5f);
    springbox2->CreateFixture(&shape2, 5.0f);

    // Create the first train / the head as the movable object
        going to positive x axis
    b2PolygonShape boxShape;
```

```
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 122000.0f; // this will affect the
    box mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(3.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
    ;
m_box->SetLinearVelocity(b2Vec2(40.8f, 0.0f));

// Make a distance joint for the head train with the middle
    train
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 2.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create distance joint / chain that connect the last train
    to the middle
b2DistanceJointDef jd2;
jd2.Initialize(springbox, springbox2, b2Vec2(1.0f, 0.5f), bd2
    .position);
jd2.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd2.length;
m_minLength = 2.0f;
m_maxLength = 2.0f;
```

```
b2LinearStiffness(jd2.stiffness, jd2.damping, m_hertz,
    m_dampingRatio, jd2.bodyA, jd2.bodyB);
m_joint2 = (b2DistanceJoint*)m_world->CreateJoint(&jd2);
m_joint2->SetMinLength(m_minLength);
m_joint2->SetMaxLength(m_maxLength);

// For the second train going toward negative x axis
// Create the middle train body
b2BodyDef bd3;
bd3.type = b2_dynamicBody;
bd3.angularDamping = 0.1f;

bd3.position.Set(31.0f, 0.5f);
b2Body* springbox3 = m_world->CreateBody(&bd3);

b2PolygonShape shape3;
shape3.SetAsBox(0.5f, 0.5f);
springbox3->CreateFixture(&shape3, 5.0f);
// Create the last train body
b2BodyDef bd4;
bd4.type = b2_dynamicBody;
bd4.angularDamping = 0.1f;

bd4.position.Set(29.0f, 0.5f);
b2Body* springbox4 = m_world->CreateBody(&bd4);

b2PolygonShape shape4;
shape4.SetAsBox(0.5f, 0.5f);
springbox4->CreateFixture(&shape4, 5.0f);

// Create the first train / the head
b2PolygonShape boxShape2;
boxShape2.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 122000.0f; // this will affect the
    box mass
boxFixtureDef2.friction = 0.1f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(33.0f, 0.5f);

m_box2 = m_world->CreateBody(&boxBodyDef2);
b2Fixture *box2Fixture = m_box2->CreateFixture(&
    boxFixtureDef2);
```

```
        m_box2->SetLinearVelocity(b2Vec2(-40.8f, 0.0f));

        // Make a distance joint for the head train with the middle
            train
        m_hertz = 1.0f;
        m_dampingRatio = 0.1f;

        b2DistanceJointDef jd3;
        jd3.Initialize(springbox3, m_box2, b2Vec2(31.0f, 0.5f),
            boxBodyDef2.position);
        jd3.collideConnected = true; // In this case we decide to
            allow the bodies to collide.
        m_length = jd3.length;
        m_minLength = 2.0f;
        m_maxLength = 2.0f;
        b2LinearStiffness(jd3.stiffness, jd3.damping, m_hertz,
            m_dampingRatio, jd3.bodyA, jd3.bodyB);

        m_joint3 = (b2DistanceJoint*)m_world->CreateJoint(&jd3);
        m_joint3->SetMinLength(m_minLength);
        m_joint3->SetMaxLength(m_maxLength);

        // Make a distance joint that connect the last train to the
            middle
        b2DistanceJointDef jd4;
        jd4.Initialize(springbox3, springbox4, b2Vec2(31.0f, 0.5f),
            bd4.position);
        jd4.collideConnected = true; // In this case we decide to
            allow the bodies to collide.
        m_length = jd4.length;
        m_minLength = 2.0f;
        m_maxLength = 0.0f;
        b2LinearStiffness(jd4.stiffness, jd4.damping, m_hertz,
            m_dampingRatio, jd4.bodyA, jd4.bodyB);
        m_joint4 = (b2DistanceJoint*)m_world->CreateJoint(&jd4);
        m_joint4->SetMinLength(m_minLength);
        m_joint4->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_box2;
    b2DistanceJoint* m_joint;
    b2DistanceJoint* m_joint2;
    b2DistanceJoint* m_joint3;
    b2DistanceJoint* m_joint4;
    float m_length;
    float m_minLength;
```

```
        float m_maxLength;
        float m_hertz;
        float m_dampingRatio;
        float m_time;
        float F = 50.0f;
        int theta = 30;

        void Step(Settings& settings) override
        {
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
                 60 Hertz
            b2MassData massData = m_box->GetMassData();
            b2MassData massData2 = m_box2->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 position2 = m_box2->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2Vec2 velocity2 = m_box2->GetLinearVelocity();

            g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                f", m_time);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Train 1 position =
                (%4.1f, %4.1f)", position.x, position.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Train 2 position =
                (%4.1f, %4.1f)", position2.x, position2.y);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "Train 1 velocity =
                (%4.1f, %4.1f)", velocity.x, velocity.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Train 2 velocity =
                (%4.1f, %4.1f)", velocity2.x, velocity2.y);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "Train 1 Mass = %.6f",
                massData.mass);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Train 2 Mass = %.6f",
                massData2.mass);
            m_textLine += m_textIncrement;

            Test::Step(settings);

            printf("%4.2f %4.2f %4.2f %4.2f \n", velocity.x, velocity.y,
                velocity2.x, velocity2.y);
        }
        static Test* Create()
```

```
        {
                return new KineticenergyCollision;
        }

    };

    static int testIndex = RegisterTest("Kinetic Energy and Work", "Kinetic
        Energy: Collision", KineticenergyCollision::Create);
```

**C++ Code 62:** *tests/kineticenergy_collision.cpp "Kinetic Energy Train Crash Box2D"*

**Figure 9.1:** *The simulation of two trains heading toward each other at speed of $v = 40.8\ m/s$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/kineticenergy_collision.cpp****).*

The simulation is very deterministic since we can repeat it and how the train collides then being thrown into the air is pretty much the same all the time. If in 1896, Box2D already exists, maybe there will be no need to make the demonstration of real locomotives collide into each other, if it is only for science discovery satisfaction.

## III. Work and Kinetic Energy

[DF*] Work, $W$, is energy transferred to or from an object via a force acting on the object. Energy transferred to the object is positive work, while energy transferred from the object is negative work.

[DF*] The work done on a particle by a constant force $\vec{F}$ during displacement $d$ is

$$W = Fd\ \cos\ \phi = \vec{F} \cdot \vec{d}$$

in which $\phi$ is the constant angle between the direction of $\vec{F}$ and $\vec{d}$.
Only the component of $\vec{F}$ that is along the displacement $\vec{d}$ can do work on the object.

[DF*] When two or more forces act on an object, their net work is the sum of the individual works done by the forces, which is also equal to the work that would be done on the object by the net force $\vec{F}_{net}$ of those forces.

[DF*] For a particle, a change $\triangle K$ in the kinetic energy equals the net work, $W$, done on the particle:

$$\triangle K = K_f - K_i = W$$

in which $K_i$ is the initial kinetic energy of the particle and $K_f$ is the kinetic energy after the work is done. The equation rearrange gives us

$$K_f = K_i + W$$

[DF*] Work has the SI unit of the joule, the same as the kinetic energy. The corresponding unit in the British system is the foot-pound ($ft \cdot lb$)

$$1\ J = 1\ kg \cdot m/s^2 = 1\ N \cdot m = 0.738\ ft \cdot lb \tag{9.2}$$

## IV. Simulation for Work By Two Constant Forces with Box2D

I take this from sample problem 7.02 of [6], but I modify everything.

Two inventors-engineers sliding an initially stationary 230 $kg$ cutting tool that is just arrive from the front yard near their garden and ranch toward their home laboratory, for a displacement $\vec{d}$ of magnitude 46 $m$. This newly wed spouse in Mt Logan, Yukon, Canada are very happy to be able to build their dream to be engineers, inventors and scientists together, not to mention they are supported by the government of Canada along with the community that is very friendly and open-minded to same-sex couple like them. The push $\vec{F}_1$ of DS Glanzsche is 23.0 $N$ at an angle $30.0^0$ downward from the horizontal; the pull $\vec{F}_2$ of DS Glanzsche' wife is 24.0 $N$ at $33.0^0$ above the horizontal. The magnitudes and directions of these forces do not change as the cutting tool moves, and the floor and the cutting tool make frictionless contact.

What is the net work done on the cutting tool by forces $\vec{F}_1$ and $\vec{F}_2$ during the displacement $\vec{d}$.

**Solution:**

The net work, $W$, done on the cutting tool by the two forces is the sum of the works they do individually.

We can treat the cutting tool as a particle and the forces are constant in both magnitude and direction.

The work done by $\vec{F}_1$ is

$$\begin{aligned} W_1 &= F_1 d \cos\ \phi_1 \\ &= (30)(46)\cos(30.0^0) \\ &= 1195.12 \end{aligned}$$

The work done by $\vec{F}_2$ is

$$\begin{aligned} W_2 &= F_2 d \cos\ \phi_2 \\ &= (24)(46)\cos(33.0^0) \\ &= 925.89 \end{aligned}$$

**Figure 9.2:** *The free-body diagram for the cutting tool.*

Thus, the net work $W$ is

$$\begin{aligned} W &= W_1 + W_2 \\ &= 1195.12 + 925.89 \\ &= 2121.01 \end{aligned}$$

During the 46 $m$ displacement, the newly wed transfer 2121.01 $J$ of energy to the kinetic energy of the cutting tool.

$$\begin{aligned} W &= \triangle K \\ 2121.01 &= K_f - K_i \\ 2121.01 &= K_f - 0 \\ 2121.01 &= \frac{1}{2}mv_f^2 \\ 2121.01 &= \frac{1}{2}230v_f^2 \\ v_f^2 &= \frac{4242.02}{230} \\ v_f &= \sqrt{18.4436} \\ v_f &= 4.3 \end{aligned}$$

$K_i$ is the initial kinetic energy, since the cutting tool at rest initially it will be 0, thus the cutting tool will be moving 4.3 $m/s$, thus in less than 11 seconds the cutting tool will be arriving in the home laboratory.

From $v_f$ we can do backward checking to see if the speed is valid, from equation **(7.3)**:

$$\begin{aligned} v_f^2 &= v_i^2 + 2a(x_f - x_i) \\ 4.3^2 &= 0 + 2a(46) \\ a &= 0.200978 \end{aligned}$$

then by using Newton's Second Law

$$\begin{aligned} F &= ma \\ F &= (230)0.200978 \\ F &= 46.225 \end{aligned}$$

With the known force, we can find the work from the result above:

$$\begin{aligned} W &= Fs \\ W &= (46.225)46 \\ W &= 2126.35 \end{aligned}$$

There's a bit of error margin here, when we compute from the push and pull forces we obtain work of 2121.01 $J$, when we compute from the velocity $v_f$, the obtain $a$ then $F$ and we can compute $W$ it becomes 2126.35 $J$, the relative error compared to the $W = 2121.01$ is 0.0025.

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class KineticenergyWork : public Test
{
	public:
	KineticenergyWork()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f
				, 46.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
				46.0f));
```

```
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                    0.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        // Create the cutting tool going to positive x axis
        b2PolygonShape boxShape;
        boxShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 230.0f; // this will affect the box
            mass
        boxFixtureDef.friction = 0.0f; // frictionless
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(-23.0f, 0.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
            ;
        //m_box->SetLinearVelocity(b2Vec2(40.8f, 0.0f));

        m_time = 0.0f;
}
b2Body* m_box;
float m_time;
float F1 = 30.0f;
float F2 = 24.0f;
float d = 46.0f;
int theta = 30;
int theta2 = 33;

float fsum = F1*d*cosf(theta*DEGTORAD) + F2*d*cosf(theta2*DEGTORAD);
float vf = sqrt(2*fsum/230);
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
```

```
                for (int i = 0; i <= 46.0f; ++i)
                {
                        //m_box->ApplyForceToCenter(b2Vec2(2121.0f, 0.0
                            f), true);
                        m_box->ApplyForceToCenter(b2Vec2(F1*i*cosf(
                            theta*DEGTORAD) + F2*i*cosf(theta2*DEGTORAD
                            ), 0.0f), true); // resulting in same work
                            for 46 meter displacement
                }
                break;
                case GLFW_KEY_F:
                m_box->SetLinearVelocity(b2Vec2(vf, 0.0f));
                break;
        }
}
void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity = m_box->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Press D to make the
            cutting tool move toward positive x axis with force");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press F to make the
            cutting tool move toward positive x axis with constant
            velocity");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Cutting tool position
            = (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Cutting tool velocity
            = (%4.1f, %4.1f)", velocity.x, velocity.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Work done to move the
            cutting tool 46 meter = %.4f", fsum);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Cutting tool Mass =
            %.6f", massData.mass);
        m_textLine += m_textIncrement;

        Test::Step(settings);
```

```
                printf("%4.2f %4.2f \n", position.x, position.y);
        }
        static Test* Create()
        {
                return new KineticenergyWork;
        }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Kinetic
    Energy: Work", KineticenergyWork::Create);
```

**C++ Code 63:** *tests/kineticenergy_work.cpp "Kinetic Energy Work Box2D"*

Some explanations for the codes:

- In the keyboard press events, I create two options, one by using **ApplyForceToCenter()** and the other is **SetLinearVelocity()**. By manual computation we obtain that the kinetic energy shall give velocity of $v_f = 4.3\ m/s$, but when using the **ApplyForceToCenter()**, the speed of movement is different as it produces another number for the constant velocity that is working on the body. This might be caused by the force is converted into Newton's second Law that gives the acceleration to the body, then the velocity will be obtained from it. The formula behind the **ApplyForceToCenter()** should be look into more. Why it produces different velocity than what it should be ? For a while, the keypress "D" is still under thinking to make it right. The keypress "F" is the right one for this problem.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
                for (int i = 0; i <= 46.0f; ++i)
                {
                        //m_box->ApplyForceToCenter(b2Vec2(2121.0f
                            , 0.0f), true);
                        m_box->ApplyForceToCenter(b2Vec2(F1*i*cosf
                            (theta*DEGTORAD) + F2*i*cosf(theta2*
                            DEGTORAD), 0.0f), true); // resulting
                            in same work for 46 meter displacement

                }
                break;
                case GLFW_KEY_F:
                m_box->SetLinearVelocity(b2Vec2(vf, 0.0f));
                break;
        }
}
```

**Figure 9.3:** *The simulation of a cutting tool being moved for* 46 *m by two forces of push and pull with different angles, thus resulting in constant velocity of* 4.3 *m/s (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/kineticenergy_work.cpp****).*

Now, we can plot the position of the cutting tool with gnuplot, recompile the testbed to make the change occurs then type:
**./testbed > work.txt**

Plot it with gnuplot from the working directory:
**gnuplot**
**set xlabel "time"**
**plot "work.txt" using 1 with lines title "x_t"**

## V. Work Done By The Gravitational Force

[DF*] The work $W_g$ done by the gravitational force $\vec{F}_g$ on a particle-like object of mass $m$ as the object moves through a displacement $d$ is given by

$$W_g = mgd \ \cos \ \phi \tag{9.3}$$

in which $\phi$ is the angle between $\vec{F}_g$ and $\vec{d}$.

[DF*] The work $W_a$ done by an applied force as a particle-like object is either lifted or lowered is related to the work $W_g$ done by the gravitational force and the change $\triangle K$ in the object's kinetic energy by

$$\triangle K = K_f - K_i = W_a + W_g \tag{9.4}$$

**Figure 9.4:** *The gnuplot of the position toward x axis for the cutting tool till it reach the distance of* 46 *m. The time is in Hertz, thus* 60 *time* = 1 *seconds, around 11 seconds will the cutting tool arrives in the home laboratory.*

If $K_f = K_i$, then the equation reduces to

$$W_a = -W_g$$

which tells us that the applied force transfers as much energy to the object as the gravitational force transfers from it.

We can rewrite it as

$$W_a = -mgd \ \cos \ \phi \tag{9.5}$$

it is the work done in lifting and lowering, with $\phi$ being the angle between $\vec{F}_g$ and $\vec{d}$. If the displacement is vertically upward, then $\phi = 180^0$ and the work done by the applied force equals $mgd$. If the displacement is vertically downward, then $\phi = 0^0$ and the work done by the applied force equals $-mgd$.

## VI. Simulation for Work in Pulling and Pushing a Sack up a Frictionless Slope

Taken from sample problem 7.04 and Problem no 24 chapter 7 of [6], I modify it a lot.

In Valhalla Projection, it divided into two parts: Midgard Area (where villagers with low education and high laziness throw trashes there when they pass by), and the other one is Asgard Area, the forest where DS Glanzsche has been contracted by Berlin to work as Grass cutter, 7 chakras builder, maintaining the forest. The Midgard area is descending from main road, that probably why based on Feng Shui, you should not have a home below the main road. While the Asgard area is ascending from the main road, meaning will lift you higher, like heaven and sky.

Now after cleaning up Midgard area, DS Glanzsche has a problem to bring a sack full of trashes that is too heavy into the main road, to be transported then by motorcycle to the trash colletion point down at the city, the sack is filled with diapers, leftover clothes, etc that make the

sack has mass of 50 $kg$. Now, learning physics should not come to a waste, she has to learn how this sack full of trashes can be brought to the main road, is it better to pull it from above? if it is pulled along a ramp, the sack starts and ends at rest. She can create the ramp frictionless from glass material with angle of $30^0$, through distance $d = 20\ m$.

Another option is to push the sack through the ramp, with a horizontal force $F_a$ of minimum magnitude that we can determined, the sack has zero kinetic energy at the start of the displacement.

Calculate how much work is done by each force acting on the sack full of trashes with pulling method and the minimum amount of force in pushing method.

**Solution:**

The first thing to do with most physics problems involving forces is to draw a free-body diagram so we can organize our thoughts.

**Figure 9.5:** *The free-body diagram of the sack being pulled from above with a rope.*

**Pull the Sack**

1. Work $W_N$ by the normal force:
   The normal force is perpendicular to the slope and thus also to the sack's displacement. Thus the normal force does not affect the sleigh's motion and does zero work.

$$W_N = F_N\ d\ \cos\ 90^0 = 0$$

2. Work $W_g$ by the gravitational force:
   We use the full gravitational force $\vec{F}_g$, the angle between $\vec{F}_g$ and $\vec{d}$ is $120^0$. This gives us

$$\begin{aligned} W_g &= F_g\ d\ \cos\ 120^0 \\ &= mgd\ \cos\ 120^0 \\ &= (50)(9.8)(20)(-0.5) \\ &= -4900 \end{aligned}$$

3. Work $W_T$ by the rope's force:
   The quickest way to calculate this is by using the work-kinetic energy theorem ($\triangle K = W$), where the net work $W$ done by the forces is $W_N + W_g + W_T$ and the change $\triangle K$ in the kinetic

energy is just zero (because the initial and final kinetic energies are the same). This gives us

$$\begin{aligned} \triangle K &= W_N + W_g + W_T \\ 0 &= 0 - 4900 + W_T \\ W_T &= 4900 \end{aligned}$$

The work needed to pull the sack is 4900 $J$.

The second way to calculate $W_T$ is by applying Newton's second law for motion along the $x$ axis to find the magnitude $F_T$ of the rope's force. Assuming that the acceleration along the slope is zero,

$$\begin{aligned} F_{net,x} &= ma_x \\ F_T - mg\ \sin\ 30^0 &= m(0) \\ F_T &= mg\ \sin\ 30^0 \end{aligned}$$

We have the magnitude of the force $F_T$, now we can find $W_T$. Because the force and the displacement are both have the same direction, up the slope, the angle between $\vec{F}_T$ and $\vec{d}$ is zero. To find the work done by the rope's force:

$$\begin{aligned} W_T &= F_T\ d\ \cos\ 0^0 \\ &= (mg\ \sin\ 30^0)d\ \cos\ 0^0 \\ &= (50)(9.8)(0.5)(20)(1) \\ &= 4900 \end{aligned}$$

Confirmed with the second method, we need the same work of 4900 $J$ to pull the sack. Combine with nutrition and biology, what kind of intake needed for that amount of energy in calorie terms, is a Koko Krunch cereal enough? or DS Glanzsche' favorite Choipanku enough to pull that sack for dinner the previous night before working in the morning for this? Tell you what, Big Tree is happy and want to give you all with that peanuts that you love, Kacang Bali and Cashew along with Polong. You get more than symbiosis mutualism when working with Nature. All science are related and can be fun to work on, all we do are all still science, even how people fall in love. Still based on astronomical places of planets and asteroids.

Another way of thinking, this is in Mechanical Engineering area, but can be noted here since this might be useful for all kind of applications. Instead of pulling the rope with human' labour or animal labour, my dogs will be wild and won't do any of this, we can use a mechanical rotation wheel so the rope will be pulled when the wheel rotating, just tie the rope to the wheel, and make it rotating automatically to the same direction till the sack arrived. One day this can be created and demonstrated, with minimal cost. It is called rotational leverage.

**Push the Sack**

Now for the pushing part, we will calculate how much force is needed to push that sack to the main road.

1. Work $W_g$ by the gravitational force
   This work bring down the sack to the bottom of the ramp due to gravity.

$$\begin{aligned} W_g &= F_g\ d\ \sin\ \theta \\ &= mgd\ \sin\ 30^0 \\ &= (50)(9.8)(20)(0.5) \\ &= 4900 \end{aligned}$$

   The work is still the same in the pull method, since this work by gravitational force will stay the same with same condition.

2. Work $W_a$ by the pushing horizontal force
   To push the sack till it went up the ramp we can use Newton's second law

$$\begin{aligned} \sum F &= 0 \\ F_a - F_g &= 0 \\ F_a &= mg\ \sin\ \theta \\ &= (50)(9.8)(0.5) \\ &= 245 \end{aligned}$$

   The minimum horizontal force, $F_a$ needed is 245 $N$. The work from this is

$$\begin{aligned} W_a &= F_a d \\ &= 245(20) \\ &= 4900 \end{aligned}$$

   The same amount of work of 4900 $J$ with the pulling method is needed to bring the sack to the main road.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class WorkGravitationalforce : public Test
{
	public:
	WorkGravitationalforce()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		float L = 10.0f;
		float a = 1.0f;
		float b = 1.5f;
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);
```

```
        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f,0.0f), b2Vec2(46.0f,
            0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(0.0f,8.0f), b2Vec2(46.0f, 8.0
            f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {// Create the pulley
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(0.0f, b + L); // circle with center of
            (0,b+L)
        ground->CreateFixture(&circle, 0.0f);
    }
    // Create the triangle
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
        (0,10)};
    chainShape.CreateLoop(vertices, 3);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    {
        // Create the sack on a triangle
        b2PolygonShape boxShape;
        boxShape.SetAsBox(a, b); // width and length of the
            box
```

```
b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 8.334f; // this will affect
    the box mass
boxFixtureDef.friction = 0.0f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-17.0f, 0.5);
// boxBodyDef.fixedRotation = true;

m_boxl = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxl->CreateFixture(&
    boxFixtureDef);

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a, b); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 16.667f; // this will affect
    the box mass, mass = density*5.9969
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(3.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-16.0f, 0.5f); // the position of the
    end string of the left cord to connect to the sack
b2Vec2 anchor2(2.0f, L); // the position of the end
    string of the right cord to connect to the right
    puller
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
    the cord is tightened at (-1.0f, b+L)
b2Vec2 groundAnchor2(1.5f, b + L); // the string of
    the cord is tightened at (1.5f, b+L)
```

```
                // the last float is the ratio
                pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
                    groundAnchor2, anchor1, anchor2, 1.0f);

                m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
                    pulleyDef);
        }
    }
    b2Body* m_boxl;
    b2Body* m_boxr;
    float F1 = 50.0f;
    int theta = 30;
    b2PulleyJoint* m_joint1;
    void Keyboard(int key) override
    {
        switch (key)
        {
                case GLFW_KEY_D:
                m_boxr->ApplyForceToCenter(b2Vec2(20000.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_F:
                m_boxr->SetLinearVelocity(b2Vec2(14.0f, 0.0f));
                break;
                case GLFW_KEY_G:
                for (int i = 0; i <= 46.0f; ++i)
                {
                        m_boxl->ApplyForceToCenter(b2Vec2(F1*i*cosf(
                            theta*DEGTORAD), 0.0f), true); // resulting
                             in same work for 46 meter displacement
                }
                break;
        }
    }
    void Step(Settings& settings) override
    {
        Test::Step(settings);
        b2MassData massData1 = m_boxl->GetMassData();
        b2Vec2 position1 = m_boxl->GetPosition();
        b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float m1 = massData1.mass;
        float m2 = massData2.mass;
        float g = 9.8f;

        float ke1 = 0.5*m1*((velocity1.x)*(velocity1.x));
```

```
float ke2 = 0.5*m2*((velocity2.x)*(velocity2.x));
float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

g_debugDraw.DrawString(5, m_textLine, "Press D to pull the
    sack with force of 20,000 N");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press F to pull the
    sack with linear velocity of 14 m/s");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press G to push the
    sack with force of 50 N");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box position =
    (%4.1f, %4.1f)", position1.x, position1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
     massData1.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box Kinetic
    Energy = %.6f", ke1);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Right Box position =
    (%4.1f, %4.1f)", position2.x, position2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
    , massData2.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box Kinetic
    Energy = %.6f", ke2);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
    a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The tension of the
    cord = %.6f", T);
m_textLine += m_textIncrement;

float ratio = m_joint1->GetRatio();
float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
```

```
                    GetCurrentLengthB();
                g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
                    f", (float) ratio, (float) L);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f %4.2f %4.2f \n", position1.y, position2.x
                    , ke1, ke2);
        }

        static Test* Create()
        {
                return new WorkGravitationalforce;
        }
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "
    Gravitational Force", WorkGravitationalforce::Create);
```

**C++ Code 64:** *tests/work_gravitationalforce.cpp "Push and Pull along a Ramp Box2D"*

**Figure 9.6:** *The simulation of sack of trashes being push from below or being pulled from above along a frictionless ramp with angle of elevation of $\theta = 30^0$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_gravitationalforce.cpp****).*

Some explanations for the codes:

- For the push force with key press "G", I use the force of $F1 = 50\ N$, because if I put 245 $N$, the box will jump out of sudden, yes we need that amount of 245 $N$ based on the calculation

from above, but it should be gained after repeated pushing, as when we push anything it is continuous giving of forces, not only once.

For the pull with key press "D" and "F", we can set the analogy for "D" as pull with a mechanical machine, and for "F" it is like tie up the end of the cord into a DC motor, motorcycle or a horse then make it run with velocity of 14 $m/s$ for certain meter to pull the sack up.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
                m_boxr->ApplyForceToCenter(b2Vec2(20000.0f, 0.0f)
                    , true);
                break;
                case GLFW_KEY_F:
                m_boxr->SetLinearVelocity(b2Vec2(14.0f, 0.0f));

                break;
                case GLFW_KEY_G:
                for (int i = 0; i <= 46.0f; ++i)
                {
                        m_boxl->ApplyForceToCenter(b2Vec2(F1*i*
                            cosf(theta*DEGTORAD), 0.0f), true); //
                             resulting in same work for 46 meter
                            displacement
                }
                break;
        }
}
```

## VII. Work Done By A Spring Force

Spring will be explained extensively in Oscillation chapter far below. We will only put minimal needed to know. This is the introduction on Spring Force.

[DF*] The force $\vec{F}_s$ from a spring is proportional to the displacement $\vec{d}$ of the free end from its position when the spring is in the relaxed state (neither compressed nor extended). The spring force is given by

$$\vec{F}_s = -k\vec{d} \tag{9.6}$$

which is known as Hooke's law. The minus sign indicates that the direction of the spring force is always opposite the displacement of the spring's free end. The constant $k$ is the spring constant to measure the spring's stiffness.

Now if we put the $x$ axis parallel to the length of the spring, with the origin ($x = 0$) at the position of the free end when the spring is in its relaxed state. Then we will have

$$F_x = -kx \tag{9.7}$$

If $x$ is positive (the spring is stretched toward the positive $x$ axis), then $F_x$ is negative. If $x$ is negative (the spring is compressed toward the negative $x$ axis), then $F_x$ is positive.
Hookes' law is a linear relationship between $F_x$ and $x$

[DF*] A spring force is a variable force. It varies with the displacement of the spring's free end.

[DF*] Two assumptions commonly used in spring problem:

1. The spring is massless; its mass is negligible relative to the block's mass.
2. It is an ideal spring; that is, it obeys Hooke's law.

[DF*] Let a block attached to a spring' free end, the block' initial position be $x_i$ and its later position be $x_f$. We can create partitions between the initial and later position of same width. Thus, we can approximate the force magnitude as being constant within the partition. Then, we can find the work done within each segment

$$\begin{aligned} W_s &= \sum -F_{xj} \triangle x \\ &= \int_{x_i}^{x_f} -F_x \, dx \\ &= \int_{x_i}^{x_f} -kx \, dx \\ &= -k \int_{x_i}^{x_f} x \, dx \\ &= \left(-\frac{1}{2}k\right) [x^2]_{x_i}^{x_f} \\ &= \left(-\frac{1}{2}k\right) \left(x_f^2 - x_i^2\right) \end{aligned}$$

thus

$$W_s = \frac{1}{2}kx_i^2 - \frac{1}{2}kx_f^2 \tag{9.8}$$

## VIII. Simulation for Spring Work with Box2D

Taken from sample problem 7.06 from [6].

A box of mass $m = 14\ kg$ located at $x = 15$ slides across a horizontal frictionless counter with speed $v = 10\ m/s$. It then runs into and compresses a spring of spring constant $k = 330N/m$, with its' free end located at $x = 5$. When the box is momentarily stopped by the spring, by what distance $d$ is the spring compressed?

**Solution:**

The work $W_s$ done on the box by the spring force is related to the requested distance $d$

$$W_s = -\frac{1}{2}kx^2$$

with $d$ replacing $x$. The work, $W_s$ is also related to the kinetic energy of the box by

$$K_f - K_i = W$$

The bos's kinetic energy has an initial value of

$$K = \frac{1}{2}mv^2$$

and a value of zero when the box is momentarily at rest.

Now, we write the work-kinetic energy theorem for the box as

$$\begin{aligned} K_f - K_i &= -\frac{1}{2}kd^2 \\ 0 - \frac{1}{2}mv^2 &= -\frac{1}{2}kd^2 \\ d &= v\sqrt{\frac{m}{k}} \\ &= (10)\sqrt{\frac{14}{330}} \\ d &= 2.0597 \end{aligned}$$

the spring is compressed till 2.0597 $m$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringWork : public Test
{
	public:
	SpringWork()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		// Create a static body as the box for the spring
		b2BodyDef bd1;
		bd1.type = b2_staticBody;
		bd1.angularDamping = 0.1f;

		bd1.position.Set(1.0f, 0.5f);
		b2Body* springbox = m_world->CreateBody(&bd1);

		b2PolygonShape shape1;
```

```
                    shape1.SetAsBox(0.5f, 0.5f);
                    springbox->CreateFixture(&shape1, 5.0f);

                    // Create the box as the movable object
                    b2PolygonShape boxShape1;
                    boxShape1.SetAsBox(0.5f, 0.5f);

                    b2FixtureDef boxFixtureDef1;
                    boxFixtureDef1.restitution = 0.75f;
                    boxFixtureDef1.density = 14.0f; // this will affect the box
                        mass
                    boxFixtureDef1.friction = 0.10;
                    boxFixtureDef1.shape = &boxShape1;

                    b2BodyDef boxBodyDef1;
                    boxBodyDef1.type = b2_dynamicBody;
                    boxBodyDef1.position.Set(15.0f, 0.5f);

                    m_box1 = m_world->CreateBody(&boxBodyDef1);
                    b2Fixture *boxFixture1 = m_box1->CreateFixture(&
                        boxFixtureDef1);

                    // Create the box attached to the spring free end
                    b2PolygonShape boxShape;
                    boxShape.SetAsBox(0.1f, 0.5f);

                    b2FixtureDef boxFixtureDef;
                    boxFixtureDef.restitution = 0.75f;
                    boxFixtureDef.density = 7.3f; // this will affect the box
                        mass
                    boxFixtureDef.friction = 0.0f;
                    boxFixtureDef.shape = &boxShape;

                    b2BodyDef boxBodyDef;
                    boxBodyDef.type = b2_dynamicBody;
                    boxBodyDef.position.Set(5.0f, 0.5f);

                    m_box = m_world->CreateBody(&boxBodyDef);
                    b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
                        ;
                    //m_box->SetGravityScale(-7); // negative means it will goes
                         upward, positive it will goes downward
                    // Make a distance joint for the box / ball with the static
                        box above
                    m_hertz = 1.0f;
                    m_dampingRatio = 0.1f;

                    b2DistanceJointDef jd;
```

```
        jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
            boxBodyDef.position);
        jd.collideConnected = true; // In this case we decide to
            allow the bodies to collide.
        m_length = jd.length;
        m_minLength = 2.0f;
        m_maxLength = 10.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
            m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_box1;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
            m_box1->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
            break;
            case GLFW_KEY_S:
            m_box1->SetLinearVelocity(b2Vec2(-5.0f, 0.0f));
            break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 150.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
            ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
            f"))
        {
```

```
                m_length = m_joint->SetLength(m_length);
            }

            if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                ))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            ImGui::End();
        }
        void Step(Settings& settings) override
        {
            b2MassData massData1 = m_box1->GetMassData();
            b2MassData massData = m_box->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 position1 = m_box1->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2Vec2 velocity1 = m_box1->GetLinearVelocity();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
                 60 Hertz

            g_debugDraw.DrawString(5, m_textLine, "Press A/S to apply
                different speed to the box");
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                f", m_time);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Spring Mass position =
                 (%4.1f, %4.1f)", position.x, position.y);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Spring Mass velocity =
                     (%4.1f, %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Spring Block Mass =
                    %.6f", massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Moving Box position =
                    (%4.1f, %4.1f)", position1.x, position1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Moving Box velocity =
                    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Moving Box Mass = %.6f
                    ", massData1.mass);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f\n", position.x);

                Test::Step(settings);
            }
            static Test* Create()
            {
                return new SpringWork;
            }

    };

    static int testIndex = RegisterTest("Kinetic Energy and Work", "Spring Work
        ", SpringWork::Create);
```

**C++ Code 65:** *tests/spring_work.cpp "Work Done by A Spring Force Box2D"*

Now, we can plot the position of the spring free end with gnuplot, recompile the testbed to make the change occurs then type:
**./testbed > springwork.txt**

Plot it with gnuplot from the working directory:
**gnuplot**
**set xlabel "time"**
**plot "springwork.txt" using 1 with lines title "d"**

**Figure 9.7:** *The spring is compressed for d distance when a box hit it with speed of $v = 10\ m/s$ on the frictionless floor (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_work.cpp****).*

**Figure 9.8:** *The plot of the spring position after being hit by a box with speed of $v = 10\ m/s$, it will goes to negative x axis then to positive x axis, oscillates a bit then goes back to its' equilibrium position.*

## IX. Work Done By A General Variable Force

[DF*] When the force $\vec{F}$ on a particle-like object depends on the position of the object, the work done by $\vec{F}$ on the object while the object moves from an initial position $r_i$ with coordinates $(x_i, y_i, z_i)$ to a final position $r_f$ with coordinates $(x_f, y_f, z_f)$ must be found by integrating the force.
If we assume that component $F_x$ may depend on $x$ but not on $y$ or $z$, component $F_y$ may depend on $y$ but not on $x$ or $z$, and component $F_z$ may depend on $z$ but not on $x$ or $y$, then the work is

$$W = \int_{x_i}^{x_f} F_x \, dx + \int_{y_i}^{y_f} F_y \, dy + \int_{z_i}^{z_f} F_z \, dz + \tag{9.9}$$

[DF*] If $\vec{F}$ has only an $x$ component, then this reduces to

$$W = \int_{x_i}^{x_f} F(x) \, dx \tag{9.10}$$

[DF*] **Example: Work, Two Dimensional Integration**
Force $\vec{F} = (3x^2 N)\hat{i} + (4N)\hat{j}$, with $x$ in meters, acts on a particle, changing only the kinetic energy of the particle. How much work is done on the particle as it moves from coordinates $(2\ m, 3\ m)$ to $(3\ m, 0\ m)$? Does the speed of the particle increasee, decrease or remain the same?

**Solution:**
The force is a variable force because its $x$ component depends on the value of $x$. Thus, we use

$$\begin{aligned} W &= \int_2^3 3x^2 \, dx + \int_3^0 4 \, dy \\ &= 3\int_2^3 x^2 \, dx + 4\int_3^0 dy \\ &= 3\left[\frac{1}{3}x^3\right]_2^3 + 4\,[y]_3^0 \\ &= \left[3^3 - 2^3\right] + 4[0-3] \\ &= 7 \end{aligned}$$

The positive 7 $J$ result means that energy is transferred to the particle by force $\vec{F}$. Thus, the kinetic energy of the particle increases and, because $K = \frac{1}{2}mv^2$, its speed must also increase. If the work had come out negative, the kinetic energy and speed would have decreased.

## X. Simulation for Work Done by a General Variable Force with Box2D

Taken from problem number 42, chapter 7 from [6].

A cord is attached to a card that can slide along a frictionless horizontal rail aligned along an $x$ axis. The left end of the cord is pulled over a pulley, of negligible mass and friction and at cord height $h = 1.20\ m$, so the cart slides from $x_1 = 3.00\ m$ to $x_2 = 1.00\ m$. During the move, the tension in the cord is a constant 25.0 $N$. What is the change in the kinetic energy of the cart during

the move?

**Solution:**

We solve the problem using the work-kinetic energy theorem, which states that the change in kinetic energy is equal to the work done by the applied force, $\triangle K = W$. In our problem, the work done is

$$W = Fd$$

where $F$ is the tension in the cord and $d$ is the length of the cord pulled as the cart slides from $x_1$ to $x_2$. We have

$$\begin{aligned} d &= \sqrt{x_1^2 + h^2} - \sqrt{x_2^2 + h^2} \\ &= \sqrt{(3)^2 + (1.2)^2} - \sqrt{(1)^2 + (1.2)^2} \\ &= 3.23 - 1.56 \\ &= 1.67 \end{aligned}$$

which yields

$$\begin{aligned} \triangle K &= Fd \\ &= (25)(1.67) \\ &= 41.7 \end{aligned}$$

the change in the kinetic energy of the cart is 41.7 $J$.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class WorkGeneralvariable : public Test
{
        public:
        WorkGeneralvariable()
        {
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                float L = 10.0f;
                float a = 1.0f;
                float b = 1.5f;
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f,0.0f), b2Vec2(46.0f,
                            0.0f));
```

```
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(0.0f,8.8f), b2Vec2(46.0f, 8.8
            f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {// Create the pulley
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(0.0f, b + L); // circle with center of
            (0,b+L)
        ground->CreateFixture(&circle, 0.0f);
    }
    // Create the triangle
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-23,10), b2Vec2(0,0), b2Vec2
        (0,10)};
    chainShape.CreateLoop(vertices, 3);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    {
        // Create the box on the left
        b2PolygonShape boxShape;
        boxShape.SetAsBox(a, b); // width and length of the
            box

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 8.334f; // this will affect
```

```
    the box mass
boxFixtureDef.friction = 0.3f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-7.0f, 11.5);
// boxBodyDef.fixedRotation = true;

m_boxl = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxl->CreateFixture(&
    boxFixtureDef);

// Create the cart on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a, b); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 16.667f; // this will affect
    the box mass, mass = density*5.9969
boxFixtureDef2.friction = 0.0f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(5.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-6.0f, 11.5f); // the position of the
    end string of the left cord to connect to the sack
b2Vec2 anchor2(4.0f, L); // the position of the end
    string of the right cord to connect to the right
    puller
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
    the cord is tightened at (-1.0f, b+L)
b2Vec2 groundAnchor2(1.5f, b + L); // the string of
    the cord is tightened at (1.5f, b+L)
// the last float is the ratio
pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.0f);
```

```
                m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
                    pulleyDef);
        }
}
b2Body* m_boxl;
b2Body* m_boxr;
float F1 = 50.0f;
int theta = 30;
b2PulleyJoint* m_joint1;
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_D:
                m_boxr->ApplyForceToCenter(b2Vec2(-13500.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_F:
                m_boxr->SetLinearVelocity(b2Vec2(-14.0f, 0.0f));
                break;
        }
}
void Step(Settings& settings) override
{
        Test::Step(settings);
        b2MassData massData1 = m_boxl->GetMassData();
        b2Vec2 position1 = m_boxl->GetPosition();
        b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float m1 = massData1.mass;
        float m2 = massData2.mass;
        float g = 9.8f;

        float ke1 = 0.5*m1*((velocity1.x)*(velocity1.x));
        float ke2 = 0.5*m2*((velocity2.x)*(velocity2.x));
        float T = 25.0f;

        g_debugDraw.DrawString(5, m_textLine, "Press D to push the
            cart to negative x axis with force of 13,500 N");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press F to push the
            cart to negative x axis with linear velocity of 14 m/s");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box position =
            (%4.1f, %4.1f)", position1.x, position1.y);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
                    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
                     massData1.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Box Kinetic
                    Energy = %.6f", ke1);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Right Cart position =
                    (%4.1f, %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Cart velocity =
                    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Cart Mass = %.6f
                    ", massData2.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Cart Kinetic
                    Energy = %.6f", ke2);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "The tension of the
                    cord = %.6f", T);
                m_textLine += m_textIncrement;

                float ratio = m_joint1->GetRatio();
                float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
                    GetCurrentLengthB();
                g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
                    f", (float) ratio, (float) L);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f %4.2f %4.2f \n", position1.y, position2.x
                    , ke1, ke2);
        }

        static Test* Create()
        {
                return new WorkGeneralvariable;
        }
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "General
    Variable Force", WorkGeneralvariable::Create);
```

**C++ Code 66:** *tests/work_generalvariable.cpp "Work Done by a General Variable Box2D"*

In this Box2D simulation, we set the friction coefficient for the box on the left to be 0.3 so when we push the cart to the left it will eventually stop, while the floor for the cart that is moving is frictionless. The force to push the cart is $13,500\ N$, and it stops after displacement of $\triangle x = 2\ m$, the value of $x_1$ and $x_2$ will still be the same because it represents the distance to the pulley, the pulley shall be at $x = 0$, even if the starting $x$ position and ending position at the simulation is not at $x = 1$ or $x = 3$. With the displacement is the same of $2\ m$, the resulting kinetic energy will stays the same.

**Figure 9.9:** *The cart on the right is being pushed with a force $F = 13,500\ N$ on the frictionless floor from $x_1 = 3.00\ m$ to $x_2 = 1.00\ m$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_generalvariable.cpp****).*

Some explanations for the codes:

- 

- 

# XI. Power

[DF*] The power due to a force is the rate at which that force does work on an object.

[DF*] If the force does work $W$ during a time interval $\triangle t$, the average power due to the force over that time interval is

$$P_{avg} = \frac{W}{\triangle t} \tag{9.11}$$

[DF*] Instantaneous power is the instantaneous rate of doing work

$$P = \frac{dW}{dt} \tag{9.12}$$

[DF*] For a force $\vec{F}$ at an angle $\phi$ to the direction of travel of the instantaneous velocity $\vec{v}$, the instantaneous power is

$$P = Fv \cos \phi = \vec{F} \cdot \vec{v} \tag{9.13}$$

[DF*] In the British system, the unit of power is the foot-pound per second. Often the horsepower is used. These are related by

$$1 \text{ watt} = 1W = 1\ J/s = 0.738\ ft \cdot lb/s \tag{9.14}$$

$$1 \text{ horsepower} = 1\ hp = 550\ ft \cdot lb/s = 746\ W \tag{9.15}$$

[DF*] Work can be expressed as power multiplied by time

$$\begin{aligned} 1 \text{ kilowatt-hour} &= 1kW \cdot h \\ &= (10^3\ W)(3600\ s) \\ &= 3.60\ \times 10^6\ J \\ &= 3.60\ MJ \end{aligned} \tag{9.16}$$

The watt and the kilowatt-hour have become identified as electrical units.

[DF*] **Example:**

(a) At a certain instant, a particle-like object is acted on by a force $\vec{F} = (4)\hat{i} - (2)\hat{j} + (9)\hat{k}$ while the object's velocity is $\vec{v} = -(2)\hat{i} + (4)\hat{k}$. What is the instantaneous rate at which the force does work on the object?

(b) At some other time, the velocity consists of only a $y$ component. If the force is unchanged and the instantaneous power is $-12\ W$, what is the velocity of the object?

**Solution:**

(a) We obtain

$$\begin{aligned} P &= \vec{F} \cdot \vec{v} \\ &= (4)(-2) + (-2)(0) + (9)(4) \\ &= 28 \end{aligned}$$

the force does work on the object for 28 $W$

(b) Focusing with a one-component velocity

$$\vec{v} = v\hat{j}$$

thus

$$\begin{aligned} P &= \vec{F} \cdot \vec{v} \\ -12 &= (-2)v \\ v &= 6 \end{aligned}$$

the velocity is 6 $m/s$.

[DF*] **Example:**
A force $\vec{F} = (3)\hat{i} + (7)\hat{j} + (7)\hat{k}$ acts on a 2 $kg$ mobile object that moves from an initial position of $\vec{d_i} = (3)\hat{i} - (2)\hat{j} + (5)\hat{k}$ to a final position of $\vec{d_f} = -(5)\hat{i} + (4)\hat{j} + (7)\hat{k}$ in 4 seconds.

(a) The work done on the object by the force in the 4 $s$ interval.

(b) The average power due to the force during that interval.

(c) The angle between vectors $\vec{d_i}$ and $\vec{d_f}$

**Solution:**

(a) The object displacement is

$$\begin{aligned} \vec{d} &= \vec{d_f} - \vec{d_i} \\ &= (-8)\hat{i} + (6)\hat{j} + (2)\hat{k} \end{aligned}$$

Thus,

$$\begin{aligned} W &= \vec{F} \cdot \vec{d} \\ &= (3)(-8) + (7)(6) + (7)(2) \\ &= 32 \end{aligned}$$

(b) The average power is

$$P_{avg} = \frac{W}{t} = \frac{32}{4} = 8$$

(c) The distance from the coordinate origin to the initial position is

$$||d_1|| = \sqrt{(3)^2 + (-2)^2 + (5)^2} = 6.16$$

and the magnitude of the distance from the coordinate origin to the final position is

$$||d_f|| = \sqrt{(-5)^2 + (4)^2 + (7)^2} = 9.49$$

Their scalar (dot) product is

$$\begin{aligned} \vec{d_i} \cdot \vec{d_f} &= (3)(-5) + (-2)(4) + (5)(7) \\ &= 12 \end{aligned}$$

Thus, the angle between the two vectors is

$$\begin{aligned} \phi &= \cos^{-1}\left(\frac{\vec{d_i} \cdot \vec{d_f}}{||d_i||||d_f||}\right) \\ &= \cos^{-1}\left(\frac{12.0}{(6.16)(9.49)}\right) \\ &= 78.2^0 \end{aligned}$$

[DF*] **Example:**
A funny car accelerates from rest through a measured track distance in time $T$ with the engine operating at a constant power $P$. If the track crew can increase the engine power by a differential amount $dP$, what is the change in the time required for the run?
**Solution:**
According to the problem statement, the power of the car is

$$P = \frac{dW}{dt} = \frac{d}{dt}\left(\frac{1}{2}mv^2\right) = mv\frac{dv}{dt}$$

with $mv\frac{dv}{dt}$ is a constant. The condition implies $dt = mv\frac{dv}{P}$, which can be integrated afterwards

$$\begin{aligned} P &= mv\frac{dv}{dt} \\ dt &= \frac{mv}{P}\,dv \\ \int_0^T dt &= \int_0^{v_T} \frac{mv}{P}\,dv \\ T &= \frac{mv_T^2}{2P} \end{aligned}$$

where $v_T$ is the speed of the car at $t = T$. On the other hand, the total distance traveled can be written as

$$\begin{aligned} L &= \int_0^T v\,dt \\ &= \int_0^{v_T} v\,\frac{mv}{P}\,dv \\ &= \frac{m}{P}\int_0^{v_T} v^2\,dv \\ &= \frac{mv_T^3}{3P} \end{aligned}$$

By squaring the expression for $L$ and substituting the expression for $T$, we obtain

$$\begin{aligned} L^2 &= \left(\frac{mv_T^3}{3P}\right)^2 \\ &= \frac{8P}{9m}\left(\frac{mv_T^2}{2P}\right)^3 \\ &= \frac{8PT^3}{9m} \end{aligned}$$

which implies that

$$PT^3 = \frac{9}{8}mL^2$$

with $\frac{9}{8}mL^2$ is a constant. Differentiating the above equation gives

$$\begin{aligned} dPT^3 + 3PT^2\,dt &= 0 \\ dt &= -\frac{T}{3P}\,dP \end{aligned}$$

## XII. Simulation for Power: Moving Elevator with Box2D

Taken from problem number 49 chapter 7 of [6].

A fully loaded, slow-moving freight elevator has a cab with a total mass of 1200 kg, which is required to travel upward 54 meters in 3 minutes, starting and ending at rest. The elevator's counterweight has a mass of only 950 kg, and so the elevator motor must help. What average power is required of the force the motor exerts on the cab via the cable?

**Solution:**

We have a loaded elevator moving upward at a constant speed. The forces involved are: gravitational force on the elevator, gravitational force on the counterweight, and the force by the motor via cable. The total work is the sum of the work done by gravity on the elevator, the work done by gravity on the counterweight, and the work done by the motor on the system:

$$W = W_e + W_c + W_m$$

Since the elevator moves at constant velocity, its kinetic energy does not change and according to the work-kinetic energy theorem the total work done is zero,

$$W = \triangle K = 0$$

The elevator moves upward through 54 $m$, so the work done by gravity on it is

$$\begin{aligned} W_e &= -m_e g d \\ &= -(1200)(9.8)(54) \\ &= -6.35 \times 10^5 \end{aligned}$$

The counterweight moves downward the same distance, so the work done by gravity on it is

$$\begin{aligned} W_c &= -m_c g d \\ &= -(950)(9.8)(54) \\ &= 5.03 \times 10^5 \end{aligned}$$

Since $W = 0$, the work done by the motor on the system is

$$\begin{aligned} W &= 0 \\ W_e + W_c + W_m &= 0 \\ W_m &= -W_e - W_c \\ &= 6.35 \times 10^5 - 5.03 \times 10^5 \\ &= 1.32 \times 10^5 \end{aligned}$$

The work is done in a time interval of $\triangle t = 3\ min = 180\ s$, so the power supplied by the motor to lift the elevator is

$$\begin{aligned} P &= \frac{W_m}{\triangle t} \\ &= \frac{1.32 \times 10^5}{180} \\ &= 7.4 \times 10^2 \end{aligned}$$

The power needed is 740 Watt.

```
#include "test.h"
#include "imgui/imgui.h"
class WorkPulleyelevator : public Test
{
        public:
        WorkPulleyelevator()
        {
                float y = 16.0f;
                float z = -28.0f;
                float L = 12.0f;
                float a = 1.0f;
                float b = 2.0f;

                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2CircleShape circle;
                        circle.m_radius = 2.0f;

                        circle.m_p.Set(-10.0f, y + b + L);
                        ground->CreateFixture(&circle, 0.0f);

                        circle.m_p.Set(10.0f, y + b + L);
                        ground->CreateFixture(&circle, 0.0f);
                }
                {

                        // Create the box hanging on the left
                        b2PolygonShape boxShape1;
                        boxShape1.SetAsBox(a, b); // width and length of the
                            left box / left elevator

                        b2FixtureDef boxFixtureDef1;
                        boxFixtureDef1.restitution = 0.75f;
                        boxFixtureDef1.density = 950.0f/8.0f; // this will
                            affect the left elevator mass
                        boxFixtureDef1.friction = 0.3f;
                        boxFixtureDef1.shape = &boxShape1;

                        b2BodyDef boxBodyDef1;
                        boxBodyDef1.type = b2_dynamicBody;
                        boxBodyDef1.position.Set(-10.0f, z);
                        // boxBodyDef2.fixedRotation = true;
```

```
                m_boxl = m_world->CreateBody(&boxBodyDef1);
                b2Fixture *boxFixture1 = m_boxl->CreateFixture(&
                    boxFixtureDef1);


                // Create the box hanging on the right
                b2PolygonShape boxShape2;
                boxShape2.SetAsBox(a, b); // width and length of the
                    right box / right elevator

                b2FixtureDef boxFixtureDef2;
                boxFixtureDef2.restitution = 0.75f;
                boxFixtureDef2.density = 150.0f; // this will affect
                    the right elevator mass
                boxFixtureDef2.friction = 0.3f;
                boxFixtureDef2.shape = &boxShape2;

                b2BodyDef boxBodyDef2;
                boxBodyDef2.type = b2_dynamicBody;
                boxBodyDef2.position.Set(10.0f, z); // the distance
                    traveled by the right elevator is 54 meter
                // boxBodyDef2.fixedRotation = true;

                m_boxr = m_world->CreateBody(&boxBodyDef2);
                b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
                    boxFixtureDef2);
                m_boxr->SetLinearVelocity(b2Vec2(0.0f, 5.0f)); //
                    Apply the right elevator motor to set linear
                    velocity upward

                b2PulleyJointDef pulleyDef;
                b2Vec2 anchor1(-10.0f, z + b);
                b2Vec2 anchor2(10.0f, z + b );
                b2Vec2 groundAnchor1(-10.0f, y + b + L);
                b2Vec2 groundAnchor2(10.0f, y + b + L);
                pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
                    groundAnchor2, anchor1, anchor2, 1.5f);

                m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
                    pulleyDef);

                m_time = 0.0f;
            }
        }
        b2Body* m_boxl;
        b2Body* m_boxr;
        float m_vel1;
        float m_vel2;
```

```
        float l_mass;
        float r_mass;
        float m_time;
        bool m_fixed_camera;

        void Keyboard(int key) override
        {
            switch (key)
            {
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                    g_camera.m_center = b2Vec2(2.0f, 10.0f);
                    g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
            }
        }
        void UpdateUI() override
        {
            ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
            ImGui::SetNextWindowSize(ImVec2(420.0f, 150.0f));
            ImGui::Begin("Elevator Controls", nullptr,
                ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

            if (ImGui::SliderFloat("Left Elevator velocity", &m_vel1,
                -50.0f, 100.0f, "%.0f"))
            {
                m_boxl->SetLinearVelocity(b2Vec2(0.0f, m_vel1));
            }
            if (ImGui::SliderFloat("Right Elevator velocity", &m_vel2,
                -50.0f, 100.0f, "%.0f"))
            {
                m_boxr->SetLinearVelocity(b2Vec2(0.0f, m_vel2));
            }
            if (ImGui::SliderFloat("Left Elevator mass", &l_mass, 0.0f,
                5000.0f, "%.0f"))
            {
                b2MassData boxMassData1;
                boxMassData1.mass = l_mass;
                boxMassData1.I = 30.0f;
                m_boxl->SetMassData(&boxMassData1);
            }
            if (ImGui::SliderFloat("Right Elevator mass", &r_mass, 0.0f,
                5000.0f, "%.0f"))
            {
                b2MassData boxMassData2;
```

```
                boxMassData2.mass = r_mass;
                boxMassData2.I = 30.0f;
                m_boxr->SetMassData(&boxMassData2);
        }
        ImGui::End();
    }
    void Step(Settings& settings) override
    {
        Test::Step(settings);
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz
        float m = m_boxl->GetMass(); // same result as GetMassData()

        b2MassData massData1 = m_boxl->GetMassData();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position1 = m_boxl->GetPosition();
        b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float ratio = m_joint1->GetRatio();
        float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
            GetCurrentLengthB();

        g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
            /tracking");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Elevator Mass =
            %4.2f", massData1.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Elevator Mass
            Center = %4.2f", massData1.center);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Elevator Inertia
            = %4.2f", massData1.I);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Elevator position
             = (%4.1f, %4.1f)", position1.x, position1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Elevator velocity
             = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Elevator Mass =
            %4.2f", massData2.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Elevator
```

```
                    position = (%4.1f, %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Elevator
                    velocity = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
                    f", (float) ratio, (float) L);
                m_textLine += m_textIncrement;
                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_boxr->GetPosition();
                       g_camera.m_zoom = 3.3f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new WorkPulleyelevator;
        }

        b2PulleyJoint* m_joint1;
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Freight
    Elevator", WorkPulleyelevator::Create);
```

**C++ Code 67:** *tests/work_pulleyelevator.cpp "Work Pulley Elevator Box2D"*

Some explanations for the codes:

- This is the elevator controls GUI. We can change the left and right elevator' velocity and mass to see their dynamic interaction. We put **SetMassData(&boxMassData1)** because we need to use **const b2MassData**, when you read b2Body Class reference in the Public Member functions for **SetMassData** you will have to follow the syntax rule, it is more or less like creating a fixture for a body.

```
void UpdateUI() override
{
       ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
       ImGui::SetNextWindowSize(ImVec2(420.0f, 150.0f));
       ImGui::Begin("Elevator Controls", nullptr,
           ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize)
           ;

       if (ImGui::SliderFloat("Left Elevator velocity", &m_vel1
           , -50.0f, 100.0f, "%.0f"))
       {
              m_boxl->SetLinearVelocity(b2Vec2(0.0f, m_vel1));
       }
```

**Figure 9.10:** *The simulation of a freight elevator on the right with mass of* 1200 *kg travel upward 54 meters (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_pulleyelevator.cpp****).*

```
                if (ImGui::SliderFloat("Right Elevator velocity", &
                    m_vel2, -50.0f, 100.0f, "%.0f"))
                {
                      m_boxr->SetLinearVelocity(b2Vec2(0.0f, m_vel2));
                }
                if (ImGui::SliderFloat("Left Elevator mass", &l_mass,
                    0.0f, 5000.0f, "%.0f"))
                {
                      b2MassData boxMassData1;
                      boxMassData1.mass = l_mass;
                      boxMassData1.I = 30.0f;
                      m_boxl->SetMassData(&boxMassData1);
                }
                if (ImGui::SliderFloat("Right Elevator mass", &r_mass,
                    0.0f, 5000.0f, "%.0f"))
                {
                      b2MassData boxMassData2;
                      boxMassData2.mass = r_mass;
                      boxMassData2.I = 30.0f;
                      m_boxr->SetMassData(&boxMassData2);
                }
                //ImGui::SliderFloat("Mass", &l_mass, 0.0f, 360.0f, "%.0
                    f");
                ImGui::End();
            }
```

**b2Body Class Reference**

A rigid body. These are created via b2World::CreateBody. More...

```
#include <b2_body.h>
```

Public Member Functions

```
        b2Fixture *  CreateFixture (const b2FixtureDef *def)
        b2Fixture *  CreateFixture (const b2Shape *shape, float density)
              void  DestroyFixture (b2Fixture *fixture)
              void  SetTransform (const b2Vec2 &position, float angle)
const b2Transform &  GetTransform () const
     const b2Vec2 &  GetPosition () const
             float  GetAngle () const
     const b2Vec2 &  GetWorldCenter () const
                     Get the world position of the center of mass.
     const b2Vec2 &  GetLocalCenter () const
                     Get the local position of the center of mass.
              void  SetLinearVelocity (const b2Vec2 &v)
     const b2Vec2 &  GetLinearVelocity () const
              void  SetAngularVelocity (float omega)
             float  GetAngularVelocity () const
              void  ApplyForce (const b2Vec2 &force, const b2Vec2 &point, bool wake)
              void  ApplyForceToCenter (const b2Vec2 &force, bool wake)
              void  ApplyTorque (float torque, bool wake)
              void  ApplyLinearImpulse (const b2Vec2 &impulse, const b2Vec2 &point, bool wake)
              void  ApplyLinearImpulseToCenter (const b2Vec2 &impulse, bool wake)
```

**Figure 9.11:** *The b2Body class reference with its public member functions, you can learn what you can modify and get data from b2Body.*

From the official Box2D source code you can see the mass data from **../box2d/include/box2d/b2_shape.h**, they are in the form of struct, the mass data are: **mass, center, I**.

**b2Body Class Reference**

A rigid body. These are created via b2World::CreateBody. More...

```
#include <b2_body.h>
```

Public Member Functions

```
        b2Fixture *  CreateFixture (const b2FixtureDef *def)
        b2Fixture *  CreateFixture (const b2Shape *shape, float density)
              void  DestroyFixture (b2Fixture *fixture)
              void  SetTransform (const b2Vec2 &position, float angle)
const b2Transform &  GetTransform () const
     const b2Vec2 &  GetPosition () const
             float  GetAngle () const
     const b2Vec2 &  GetWorldCenter () const
                     Get the world position of the center of mass.
     const b2Vec2 &  GetLocalCenter () const
                     Get the local position of the center of mass.
              void  SetLinearVelocity (const b2Vec2 &v)
     const b2Vec2 &  GetLinearVelocity () const
              void  SetAngularVelocity (float omega)
             float  GetAngularVelocity () const
              void  ApplyForce (const b2Vec2 &force, const b2Vec2 &point, bool wake)
              void  ApplyForceToCenter (const b2Vec2 &force, bool wake)
              void  ApplyTorque (float torque, bool wake)
              void  ApplyLinearImpulse (const b2Vec2 &impulse, const b2Vec2 &point, bool wake)
              void  ApplyLinearImpulseToCenter (const b2Vec2 &impulse, bool wake)
```


**Figure 9.12:** *The **SetMassData** public member functions has the same properties as **CreateFixture**, thus the syntax are more or less quite the same.*

- It is quite hard to make the elevator goes up in 180 *seconds* since by default (without addition of linear velocity) from Box2D simulation the right elevator will travel upward 54 meters and arrive at the top in 13 *seconds*, with linear velocity of 5 $m/s$ toward positive $y$ axis the time to travel upward 54 meters is around 10 *seconds*. Thus we are going to use the simulation with linear mapping, so we can determine the force the motor exerts in real world condition from the simulation. By setting

```
m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&boxFixtureDef2
    );
m_boxr->SetLinearVelocity(b2Vec2(0.0f, 5.0f)); // Apply the
    right elevator motor to set linear velocity upward
```

We create a motor that exerts a power to push the elevator with linear velocity of 5 $m/s$ toward positive $y$ axis, hence it will make

$$t_{real} \approx 18t_{simulation}$$

and from equation **(7.1)** we have

$$v = v_0 + at$$

since the relation between $v$ and $t$ is linear, thus

$$v_{real} \approx 18v_{simulation}$$

with this we know the required power of the force the motor exerts in real life.

## XIII. Simulation for Power: Pendulum Crate with Box2D

Taken from problem number 57 chapter 7 from [6].

A 230 kg crate hangs from the end of a rope of length $L = 12.0\ m$. You push horizontally on the crate with a varying force $\vec{F}$ to move it distance $d = 4.00\ m$ to the side.

(a) What is the magnitude of $\vec{F}$ when the crate is in this final position?

(b) During the crate's displacement, what is the total work done on it?

(c) During the crate's displacement, what is the work done by the gravitational force on the crate?

(d) During the crate's displacement, what is the work done by the pull on the crate from the rope?

(e) Knowing that the crate is motionless before and after its displacement, use the answers to (b), (c) and (d) to find the work your force $\vec{F}$ does on the crate.

(f) Why is the work of your force not equal to the product of the horizontal displacement and the answer to (a)?

**Solution:**

(a) To hold the crate at equilibrium in the final situation, $\vec{F}$ must have the same magnitude as the horizontal component of the rope's tension $T\ \sin\ \theta$, where $\theta$ is the angle between the rope (in the final position) and vertical:

$$\theta = \sin^{-1}\left(\frac{d}{L}\right) = \sin^{-1}\left(\frac{4.00}{12.0}\right) = 19.5^0$$

But the vertical component of the tension supports against the weight:

$$T\ \cos\ \theta = mg$$

Thus,

$$\begin{aligned} T &= \frac{mg}{\cos\ \theta} \\ &= \frac{(230)(9.8)}{\cos\ 19.5^0} \\ &= 2390.728 \\ &\approx 2391 \end{aligned}$$

**Figure 9.13:** *The body diagram and the illustration of the crate hangs from the end of a rope of length $L = 12\ m$, then pushed horizontallly with a force $\vec{F}$ to move it distance $d$.*

and

$$F = T\ \sin\ \theta = (2391)\sin\ 19.5^0 = 797$$

the tension is 2391 $N$ and the force is 797 $N$.

(b) Since there is no change in kinetic energy, the net work on it is zero.

(c) The work done by gravity is

$$W_g = \vec{F}_g \cdot \vec{d} = -mgh$$

The ratio of the vertical displacement toward the rope length is

$$\frac{h}{L} = 1 - \cos\ \theta$$

when the angle $\theta = 90^0$ the vertical displacement will be the same as the rope length, $L$. Thus, for this problem the vertical component of the displacement is $h = L(1 - \cos\ \theta)$. With $L = 12\ m$, we will obtain

$$W_g = -mgh = -(230)(9.8)(12(1 - \cos\ 19.5^0)) = -1547$$

the work done by the gravitational force on the crate is 1547 $J$.

(d) The tension vector is everywhere perpendicular to the direction of motion, so its work is zero (since $\cos\ 90^0 = 0$).

(e) The implication of the previous three parts is that the work due to $\vec{F}$ is $-W_g$ (so the net work turns out to be zero). Thus,

$$W_F = -W_g = 1547$$

(f) Since $\vec{F}$ does not have constant magnitude, we cannot expect equation

$$W = \vec{F} \cdot \vec{d}$$

to apply, since that equation is a work done by a constant force.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CratePendulum : public Test
{
	public:

	CratePendulum()
	{
		b2Body* b1;
		{
			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
				(40.0f, 0.0f));

			b2BodyDef bd;
			b1 = m_world->CreateBody(&bd);
			b1->CreateFixture(&shape, 0.0f);
		}
		// the two blocks below for creating boxes are only
			for sweetener.
		{
			b2PolygonShape shape;
			shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, 0.7f);
				// Gradient is 0.7

			b2BodyDef bd;
			bd.position.Set(6.0f, 6.0f);
			b2Body* ground = m_world->CreateBody(&bd);
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2PolygonShape shape;
			shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, -0.7f)
				; // Gradient is -0.7

			b2BodyDef bd;
			bd.position.Set(-6.0f, 6.0f);
			b2Body* ground = m_world->CreateBody(&bd);
			ground->CreateFixture(&shape, 0.0f);
		}

		b2Body* b2; // the hanging bar for the pendulum
		{
```

```
                b2PolygonShape shape;
                shape.SetAsBox(7.25f, 0.25f);

                b2BodyDef bd;
                bd.position.Set(0.0f, 37.0f);
                b2 = m_world->CreateBody(&bd);
                b2->CreateFixture(&shape, 0.0f);
            }

            b2RevoluteJointDef jd;
            b2Vec2 anchor;

            // Create the pendulum ball
            b2PolygonShape boxShape1;
            boxShape1.SetAsBox(1.0f, 0.5f);

            b2FixtureDef boxFixtureDef1;
            boxFixtureDef1.restitution = 0.75f;
            boxFixtureDef1.density = 115.0f; // this will affect
                the left elevator mass
            boxFixtureDef1.friction = 0.3f;
            boxFixtureDef1.shape = &boxShape1;

            b2BodyDef boxBodyDef1;
            boxBodyDef1.type = b2_dynamicBody;
            boxBodyDef1.position.Set(0.0f, 24.0f);
            // boxBodyDef2.fixedRotation = true;
            // boxBodyDef1.angularDamping = 0.2f;

            m_box = m_world->CreateBody(&boxBodyDef1);
            b2Fixture *boxFixture1 = m_box->CreateFixture(&
                boxFixtureDef1);

            // Create the anchor and connect it to the crate
            anchor.Set(0.0f, 36.0f); // x and y axis position for
                the Pendulum anchor
            jd.Initialize(b2, m_box, anchor);
            //jd.collideConnected = true;
            m_world->CreateJoint(&jd); // Create the Pendulum
                anchor

            m_time = 0.0f;
        }
        b2Body* m_box;
        float m_time;

        void Keyboard(int key) override
        {
```

```
                switch (key)
                {
                        case GLFW_KEY_S:
                        m_box->SetTransform(b2Vec2(4.0f, 24.0f),0); //
                            warp or teleport it to move with
                            displacement of 4
                        break;
                        case GLFW_KEY_T:
                        m_time = 0.0f;
                        break;
                }
        }
        void Step(Settings& settings) override
        {
                m_time += 1.0f / 60.0f; // assuming we are using
                    frequency of 60 Hertz
                b2Vec2 v = m_box->GetLinearVelocity();
                float omega = m_box->GetAngularVelocity();
                float angle = m_box->GetAngle();
                b2MassData massData = m_box->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                float m = massData.mass;
                float g = 9.8f;
                float L = 12.0f;
                float d = 4.0f;
                float theta = asin(d/L);
                float T = (m*g)/(cos(theta));
                float F = T*sin(theta);
                float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
                     massData.I * omega * omega;

                g_debugDraw.DrawString(5, m_textLine, "Press S to
                    apply force 10,000 N to positive x axis");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Press T to
                    reset time");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in
                    seconds)= %.6f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Ball position,
                    x,y =( %4.1f, %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %4.1f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Theta = %4.2f",
                     theta*RADTODEG);
```

```
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Rope tension =
                            %4.2f", T);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Pushing Force =
                             %4.2f", F);
                        m_textLine += m_textIncrement;

                        g_debugDraw.DrawString(5, m_textLine, "Kinetic energy
                            = %4.1f", ke);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Linear velocity
                             = %4.1f", v);
                        m_textLine += m_textIncrement;
                        g_debugDraw.DrawString(5, m_textLine, "Angle (in
                            degrees) = %4.1f", angle*RADTODEG);
                        m_textLine += m_textIncrement;
                        // Print the result in every time step then plot it
                            into graph with either gnuplot or anything

                        printf("%4.2f %4.2f %4.2f\n", position.x, position.y,
                            angle*RADTODEG);
                        //std::ofstream MyFile("/root/output.txt"); ;
                        //MyFile << angle << " " << position.x << " " <<
                            position.y;

                        Test::Step(settings);
                }

                static Test* Create()
                {
                        return new CratePendulum;
                }

        };

        static int testIndex = RegisterTest("Kinetic Energy and Work", "
            Crate Pendulum", CratePendulum::Create);
```

**C++ Code 68:** *tests/work_cratependulum.cpp "Crate Pendulum Box2D"*

From the Box2D simulation I got $\theta = 18.35^0$ for making displacement of $d = 4$ for the crate, it is probably because I set the $y = 24$ for that displacement.

Some explanations for the codes:

- We are using **SetTransform** to make the displacement, there is a bug when we click "S" the crate does not move, but when we click the crate it will move normally like pendulum crate.

```
        void Keyboard(int key) override
        {
```

```
                switch (key)
                {
                        case GLFW_KEY_S:
                        m_box->SetTransform(b2Vec2(4.0f, 24.0f),0); //
                            warp or teleport it to move with displacement
                             of 4
                        break;
                        case GLFW_KEY_T:
                        m_time = 0.0f;
                        break;
                }
        }
```

**Figure 9.14:** *The simulation a crate being pushed with force $\vec{F}$ to move a distance of $d = 4$ m (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_cratependulum.cpp****).*

## XIV. Simulation for Power: Vertical Facing Upward Spring with Box2D

A 250 $g$ block is dropped onto a relaxed vertical spring that has a spring constant of $k = 2.5\ N/cm$. The block becomes attached to the spring and compresses the spring 12 $cm$ before momentarily stopping. While the spring is being compressed, what work is done on the block by

(a) The gravitational force on it

(b) The spring force

(c) What is the speed of the block just before it hits the spring? (Assume that friction is negligible)

(d) If the speed at impact is doubled, what is the maximum compression of the spring?

**Solution:**

(a) The compression of the spring is $d = 0.12\ m$. The work done by the force of gravity (acting on the block) is,

$$\begin{aligned} W_1 &= mgd \\ &= (0.25)(9.8)(0.12) \\ &= 0.29 \end{aligned}$$

the work done by the force of gravity is $0.29\ J$.

(b) For the spring constant we will convert it into MKS system, $k = 2.5\ N/cm = 250\ N/m$. Thus, the work done by the spring is,

$$\begin{aligned} W_2 &= -\frac{1}{2}kd^2 \\ &= -\frac{1}{2}(250)(0.12)^2 \\ &= -1.8 \end{aligned}$$

(c) The speed $v_i$ of the block just before it hits the spring is found from the work-kinetic energy theorem:

$$\begin{aligned} \triangle K &= W_1 + W_2 \\ 0 - \frac{1}{2}mv_i^2 &= W_1 + W_2 \\ v_i &= \sqrt{\frac{(-2)(W_1 + W_2)}{m}} \\ &= \sqrt{\frac{(-2)(0.29 - 1.8)}{0.25}} \\ &= 3.5 \end{aligned}$$

the speed $v_i = 3.5\ m/s$.

(d) If we instead had $v_i' = 7\ m/s$, we reverse the above steps and solve for $d'$. Recalling the theorem used in part (c), we have

$$\begin{aligned} \triangle K' &= W_1' + W_2' \\ 0 - \frac{1}{2}mv_i'^2 &= W_1' + W_2' \\ -\frac{1}{2}mv_i'^2 &= mgd' - \frac{1}{2}kd'^2 \\ -\frac{1}{2}kd'^2 + mgd' + \frac{1}{2}mv_i'^2 &= 0 \\ d' &= \pm\frac{mg + \sqrt{m^2g^2 + mkv_i'^2}}{k} \end{aligned}$$

now by choosing the positive root we will have

$$d' = \frac{mg + \sqrt{m^2g^2 + mkv_i'^2}}{k} = \frac{(0.25)(9.8) + \sqrt{(0.25)^2(9.8)^2 + (0.25)(250)(7)^2}}{250} = 0.23$$

the maximum compression of the spring for speed of $v_i' = 7\ m/s$ is $d' = 0.23\ m$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class WorkVerticalspringupward : public Test
{
	public:
	WorkVerticalspringupward()
	{
		b2Body* ground = NULL;
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		// Create the wall surrounding the spring
		b2BodyDef bd2;
		bd2.type = b2_staticBody;
		bd2.angularDamping = 0.1f;

		bd2.position.Set(-0.05f, 5.5f);
		b2Body* leftwall = m_world->CreateBody(&bd2);

		b2PolygonShape shape2;
		shape2.SetAsBox(0.5f, 5.5f);
		leftwall->CreateFixture(&shape2, 5.0f);

		b2BodyDef bd3;
		bd3.type = b2_staticBody;
		bd3.angularDamping = 0.1f;

		bd3.position.Set(2.05f, 5.5f);
		b2Body* rightwall = m_world->CreateBody(&bd3);

		b2PolygonShape shape3;
		shape3.SetAsBox(0.5f, 5.5f);
		rightwall->CreateFixture(&shape3, 5.0f);

		// Create a static body as the box for the spring
		b2BodyDef bd1;
```

```
bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.1f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.0f;
boxFixtureDef.density = 1.5f; // this will affect the spring
    mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(1.0f, 11.0f); // the box will be
    located in (1,11.0)

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
    ;
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
    m_minLength
m_maxLength = 12.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);
```

```
        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        // Create the falling block as the movable object
        b2PolygonShape blockShape;
        blockShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef blockFixtureDef;
        blockFixtureDef.restitution = 0.0f; // bounciness
        blockFixtureDef.density = 0.25f; // this will affect the
            falling block mass
        blockFixtureDef.friction = 0.1f;
        blockFixtureDef.shape = &blockShape;

        b2BodyDef blockBodyDef;
        blockBodyDef.type = b2_dynamicBody;
        blockBodyDef.position.Set(1.0f, 12.0f);

        m_block = m_world->CreateBody(&blockBodyDef);
        b2Fixture *blockFixture = m_block->CreateFixture(&
            blockFixtureDef);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_block;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_W:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, 10000.0f), true
                );
            break;
            case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, -9500.0f),
                true);
            break;
            case GLFW_KEY_T:
```

```
                m_time = 0.0f;
                break;
            }
        }
        void UpdateUI() override
        {
            ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
            ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
            ImGui::Begin("Joint Controls", nullptr,
                ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

            if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                f"))
            {
                m_length = m_joint->SetLength(m_length);
            }

            if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                ))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            ImGui::End();
        }
        void Step(Settings& settings) override
        {
            b2MassData massData = m_box->GetMassData();
            b2MassData massData2 = m_block->GetMassData();
            b2Vec2 position = m_box->GetPosition();
```

```
b2Vec2 velocity = m_box->GetLinearVelocity();
b2Vec2 velocity2 = m_block->GetLinearVelocity();
m_time += 1.0f / 60.0f; // assuming we are using frequency of
     60 Hertz
float m = massData.mass;
float g = 9.8f;
float y = 11.0f;
// y = the position at which when we place the mass it would
    not move / equlibrium position
// y = y position of the ceiling - m_minLength - initial y
    position of the mass
float k = 250.0f;
float y_eq = 11.0f;

g_debugDraw.DrawString(5, m_textLine, "Press W to apply force
     10,000 N upward / S to apply force 9,500 N downward");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
    );
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Falling Block Mass =
    %4.2f", massData2.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Falling Block velocity
     = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring position =
    (%4.1f, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring velocity =
    (%4.1f, %4.1f)", velocity.x, velocity.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring Mass = %4.2f",
    massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
     = %4.1f", k);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
    for the spring, y = %4.1f", y_eq);
m_textLine += m_textIncrement;
// Print the result in every time step then plot it into
    graph with either gnuplot or anything

printf("%4.2f %4.2f\n", velocity2.y, position.y);
```

```
                Test::Step(settings);
        }
        static Test* Create()
        {
                return new WorkVerticalspringupward;
        }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Vertical
    Spring Upward", WorkVerticalspringupward::Create);
```

**C++ Code 69:** *tests/work_verticalspringupward.cpp "Block Dropped onto Relaxed Spring Box2D"*

Some explanations for the codes:

- The important part of this simulation is to set the height where we drop the block onto the spring since it will affect the speed before it hits the spring. With mass of 250 $g$, the height to drop the block is very near with the spring, not too high.

```
            // Create the falling block as the movable object
            b2PolygonShape blockShape;
            blockShape.SetAsBox(0.5f, 0.5f);

            b2FixtureDef blockFixtureDef;
            blockFixtureDef.restitution = 0.0f; // bounciness
            blockFixtureDef.density = 0.25f; // this will affect the
                falling block mass
            blockFixtureDef.friction = 0.1f;
            blockFixtureDef.shape = &blockShape;

            b2BodyDef blockBodyDef;
            blockBodyDef.type = b2_dynamicBody;
            blockBodyDef.position.Set(1.0f, 12.0f);

            m_block = m_world->CreateBody(&blockBodyDef);
            b2Fixture *blockFixture = m_block->CreateFixture(&
                blockFixtureDef);
```

```
-1.80 10.86
-1.96 10.84
-2.12 10.82
-2.29 10.80
-2.45 10.78
-2.61 10.76
-2.78 10.74
-2.94 10.72
-3.10 10.70
-3.27 10.68
-3.43 10.67
-3.59 10.65
-3.76 10.64
-0.70 10.63
-0.60 10.62
-0.50 10.61
-0.40 10.60
-0.29 10.60
-0.19 10.60
-0.09 10.59
```

**Figure 9.15:** *The velocity of the falling block is around* 3.5 $m/s$ *just before it hits the spring, the negative sign means it is going toward negative* $y$ *axis.*

**Figure 9.16:** *The simulation of a* 250 $g$ *block dropped onto a relaxed vertical spring that has a spring constant* $k =$ 250 $N/m$ *(the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_verticalspringupward.cpp****).*

## XV. Simulation for Power: Boxes on Conveyor Belt with Box2D

Boxes are transported from one location to another in a warehouse by means of a conveyor belt that moves with a constant speed of 0.5 $m/s$. At a certain location the conveyor belt moves for 2.0 $m$ up an incline that makes an angle of $10^0$ with the horizontal, then for 2.0 $m$ horizontally, and finally for 2.0 $m$ down an incline that makes an angle of $10^0$ with the horizontal. Assume that a 2.0 $kg$ box rides on the belt without slipping. At what rate is the force of the conveyor belt doing work on the box as the box moves

(a) Up the $10^0$ incline

(b) Horizontally

(c) Down the $10^0$ incline

**Solution:**

**Figure 9.17:** *The illustration and body diagram for transported boxes of Nutella chocolate jam from one location to another in a warehouse.*

(a) Remember that the angle $\theta$ is between the force $\vec{F}$ and the direction of travel, thus the force $\vec{F}_N$ of the incline is a combination of normal and friction force, which is serving to "cancel" the tendency of the box to fall downward (due to its 19.6 $N$ weight). Thus, $\vec{F}_N = mg$ upward. In this part, the angle $\theta$ between the belt / the direction of movement and $\vec{F}_N$ is $90^0 - \theta = 80^0$.

$$\begin{aligned} P &= Fv \ \cos\ (90^0 - \theta) \\ &= mgv \ \cos\ (90^0 - \theta) \\ &= (2)(9.8)(0.5) \ \cos\ 80^0 \\ &= 1.7 \end{aligned}$$

The power at the incline is 1.7 $W$, positive power means that the force is supplying energy to the box to move toward the conveyor belt. Since inclination will require more energy than just a horizontal conveyor belt then it is logical indeed.

(b) Now the angle between the belt and $\vec{F}_N$ is $90^0$, so

$$\begin{aligned} P &= Fv \ \cos\ 90^0 \\ &= mgv \ \cos\ 90^0 \\ &= (2)(9.8)(0.5) \ \cos\ 90^0 \\ &= 0 \end{aligned}$$

Since it is a horizontal conveyor belt, thus no energy is supplied or removed to move the box here, so that the power here is 0 $W$.

(c) Now since the box is moving downward we will have

$$\begin{aligned} P &= Fv\ \cos\ (90^0 + \theta) \\ &= mgv\ \cos\ ((90^0 + \theta)) \\ &= (2)(9.8)(0.5)\ \cos\ 100^0 \\ &= -1.7 \end{aligned}$$

Remember that negative power means the force is removing energy, thus we need to remove energy while moving the to the box on the conveyor belt here, to make the movement stable not too fast. The energy removed is 1.7 $W$

```
#define DEGTORAD 0.0174532925199432957f
#include "test.h"

class ConveyorBelt : public Test
{
	public:

	ConveyorBelt()
	{
		// Ground
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-20.0f, 0.0f), b2Vec2(108.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}

		// First Horizontal Platform
		{
			b2BodyDef bd;
			bd.position.Set(-9.0f, 4.6f);
			b2Body* body = m_world->CreateBody(&bd);

			b2PolygonShape shape;
			shape.SetAsBox(10.0f, 0.5f);

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.friction = 0.1f;
			m_platform = body->CreateFixture(&fd);
		}
		// Second Horizontal Platform
		{
			b2BodyDef bd;
```

```
        bd.position.Set(33.650578f, 8.5f);
        b2Body* body = m_world->CreateBody(&bd);

        b2PolygonShape shape;
        shape.SetAsBox(10.0f, 0.5f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.friction = 0.1f;
        m_platform4 = body->CreateFixture(&fd);
    }
    // Third Horizontal Platform
    {
        b2BodyDef bd;
        bd.position.Set(75.650578f, 4.6f);
        b2Body* body = m_world->CreateBody(&bd);

        b2PolygonShape shape;
        shape.SetAsBox(10.0f, 0.5f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.friction = 0.1f;
        m_platform5 = body->CreateFixture(&fd);
    }

    // The first incline
    {
        b2ChainShape chainShape;
        b2Vec2 vertices[] = {b2Vec2(1,5), b2Vec2(23.650578,5),
             b2Vec2(23.650578,8.9939)};
        chainShape.CreateLoop(vertices, 3);

        b2FixtureDef groundFixtureDef;
        groundFixtureDef.density = 0;
        groundFixtureDef.friction = 0.1f;
        groundFixtureDef.shape = &chainShape;

        b2BodyDef groundBodyDef;
        groundBodyDef.type = b2_staticBody;

        b2Body *groundBody = m_world->CreateBody(&
            groundBodyDef);
        m_platform2 = groundBody->CreateFixture(&
            groundFixtureDef);
    }
    // The decline triangle
    {
```

```
                b2ChainShape chainShape;
                b2Vec2 vertices[] = {b2Vec2(43.6,5), b2Vec2
                    (66.250578,5), b2Vec2(43.6,8.9939)};
                chainShape.CreateLoop(vertices, 3);

                b2FixtureDef groundFixtureDef;
                groundFixtureDef.density = 0;
                groundFixtureDef.friction = 0.1f;
                groundFixtureDef.shape = &chainShape;

                b2BodyDef groundBodyDef;
                groundBodyDef.type = b2_staticBody;

                b2Body *groundBody = m_world->CreateBody(&
                    groundBodyDef);
                m_platform3 = groundBody->CreateFixture(&
                    groundFixtureDef);
        }
        // Boxes
        for (int32 i = 0; i < 5; ++i)
        {
                b2BodyDef bd;
                bd.type = b2_dynamicBody;
                bd.position.Set(-10.0f + 2.0f * i, 7.0f);
                body = m_world->CreateBody(&bd);

                b2PolygonShape shape;
                shape.SetAsBox(0.5f, 0.5f);
                b2FixtureDef boxFixtureDef1;
                boxFixtureDef1.restitution = 0.0f;
                boxFixtureDef1.density = 2.0f; // this will affect the
                     mass
                boxFixtureDef1.friction = 0.5f;
                boxFixtureDef1.shape = &shape;
                b2Fixture *boxFixture1 = body->CreateFixture(&
                    boxFixtureDef1);

        }
        m_time = 0.0f;
}
b2Body* body;
b2Fixture* m_platform;
b2Fixture* m_platform2;
b2Fixture* m_platform3;
b2Fixture* m_platform4;
b2Fixture* m_platform5;
float m_time;
bool m_fixed_camera;
```

```
        void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
            override
        {
               Test::PreSolve(contact, oldManifold);

               b2Fixture* fixtureA = contact->GetFixtureA();
               b2Fixture* fixtureB = contact->GetFixtureB();

               if (fixtureA == m_platform)
               {
                      contact->SetTangentSpeed(5.0f);
               }

               if (fixtureB == m_platform)
               {
                      contact->SetTangentSpeed(-5.0f);
               }

               if (fixtureA == m_platform2)
               {
                      contact->SetTangentSpeed(5.0f);
               }
               if (fixtureB == m_platform2)
               {
                      contact->SetTangentSpeed(-5.0f);
               }

               if (fixtureA == m_platform3)
               {
                      contact->SetTangentSpeed(5.0f);
               }
               if (fixtureB == m_platform3)
               {
                      contact->SetTangentSpeed(-5.0f);
               }

               if (fixtureA == m_platform4)
               {
                      contact->SetTangentSpeed(5.0f);
               }
               if (fixtureB == m_platform4)
               {
                      contact->SetTangentSpeed(-5.0f);
               }

               if (fixtureA == m_platform5)
               {
```

```
                        contact->SetTangentSpeed(0.0f);
                }
                if (fixtureB == m_platform5)
                {
                        contact->SetTangentSpeed(0.0f);
                }

        }
        void Keyboard(int key) override
        {
                switch (key)
                {
                        case GLFW_KEY_C:
                        m_fixed_camera = !m_fixed_camera;
                        if(m_fixed_camera)
                        {
                                g_camera.m_center = b2Vec2(3.0f, 10.0f);
                                g_camera.m_zoom = 1.7f; // zoom out camera
                        }
                        break;
                }
        }
        void Step(Settings& settings) override
        {
                Test::Step(settings);
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz
                b2MassData massData1 = body->GetMassData();
                b2Vec2 position1 = body->GetPosition();
                b2Vec2 velocity1 = body->GetLinearVelocity();
                float m = massData1.mass;
                float g = 9.8f;
                float v = 0.5;
                float P1 = m*g*cos(80*DEGTORAD)*v;
                float P2 = m*g*cos(90*DEGTORAD)*v;
                float P3 = m*g*cos(100*DEGTORAD)*v;

                g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
                    /tracking");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
                     %4.1f)", position1.x, position1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
                     %4.1f)", velocity1.x, velocity1.y);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %4.2f",
                    massData1.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Power at First
                    Inclination = %4.2f", P1);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Power at Horizontal
                    Platform = %4.2f", P2);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Power at Downward
                    Inclination = %4.2f", P3);
                m_textLine += m_textIncrement;

                if(!m_fixed_camera)
                {
                       g_camera.m_center = body->GetPosition();
                       g_camera.m_zoom = 1.7f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new ConveyorBelt;
        }


    };

    static int testIndex = RegisterTest("Kinetic Energy and Work", "Conveyor
        Belt", ConveyorBelt::Create);
```

**C++ Code 70:** *tests/work_conveyorbelt.cpp "Work and Conveyor Belt Box2D"*

Some explanations for the codes:

- To be able to determine the adjacent and opposite length of the triangle we can use some trigonometric here, with known $\theta = 10^0$ and the hypothenuse length of the triangle is 2 $m$

$$\cos(\theta) = \frac{adjacent}{hypothenuse}$$
$$\cos(10^0) = \frac{adjacent}{2}$$
$$adjacent = 1.9696$$

$$\sin(\theta) = \frac{opposite}{hypothenuse}$$
$$\sin(10^0) = \frac{opposite}{2}$$
$$opposite = 0.347296$$

After we put the calculated opposite and adjacent, the triangle in Box2D becomes really small since the unit here is 1 meter, thus we will adjust and enlarge the triangle, the $\theta$ will still be the same, but the hypothenuse won't be of size 2, it needs larger size so we can view it better in the simulation. We decided to use the hypothenuse of 23,

$$\cos(\theta) = \frac{adjacent}{hypothenuse}$$
$$\cos(10^0) = \frac{adjacent}{23}$$
$$adjacent = 22.650578$$

$$\sin(\theta) = \frac{opposite}{hypothenuse}$$
$$\sin(10^0) = \frac{opposite}{23}$$
$$opposite = 3.9939$$

```
// First Horizontal Platform
{
        b2BodyDef bd;
        bd.position.Set(-9.0f, 4.6f);
        b2Body* body = m_world->CreateBody(&bd);

        b2PolygonShape shape;
        shape.SetAsBox(10.0f, 0.5f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.friction = 0.1f;
        m_platform = body->CreateFixture(&fd);
}
// Second Horizontal Platform
{
        b2BodyDef bd;
        bd.position.Set(33.650578f, 8.5f);
        b2Body* body = m_world->CreateBody(&bd);

        b2PolygonShape shape;
        shape.SetAsBox(10.0f, 0.5f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.friction = 0.1f;
        m_platform4 = body->CreateFixture(&fd);
}
// Third Horizontal Platform
{
        b2BodyDef bd;
        bd.position.Set(75.650578f, 4.6f);
```

```
            b2Body* body = m_world->CreateBody(&bd);

            b2PolygonShape shape;
            shape.SetAsBox(10.0f, 0.5f);

            b2FixtureDef fd;
            fd.shape = &shape;
            fd.friction = 0.1f;
            m_platform5 = body->CreateFixture(&fd);
        }
        // The first incline
        {
            b2ChainShape chainShape;
            b2Vec2 vertices[] = {b2Vec2(1,5), b2Vec2(23.650578,5),
                b2Vec2(23.650578,8.9939)};
            chainShape.CreateLoop(vertices, 3);

            b2FixtureDef groundFixtureDef;
            groundFixtureDef.density = 0;
            groundFixtureDef.friction = 0.1f;
            groundFixtureDef.shape = &chainShape;

            b2BodyDef groundBodyDef;
            groundBodyDef.type = b2_staticBody;

            b2Body *groundBody = m_world->CreateBody(&groundBodyDef)
                ;
            m_platform2 = groundBody->CreateFixture(&
                groundFixtureDef);
        }
        // The decline triangle
        {
            b2ChainShape chainShape;
            b2Vec2 vertices[] = {b2Vec2(43.6,5), b2Vec2(66.250578,5)
                , b2Vec2(43.6,8.9939)};
            chainShape.CreateLoop(vertices, 3);

            b2FixtureDef groundFixtureDef;
            groundFixtureDef.density = 0;
            groundFixtureDef.friction = 0.1f;
            groundFixtureDef.shape = &chainShape;

            b2BodyDef groundBodyDef;
            groundBodyDef.type = b2_staticBody;

            b2Body *groundBody = m_world->CreateBody(&groundBodyDef)
                ;
            m_platform3 = groundBody->CreateFixture(&
```

```
                groundFixtureDef);
        }
```

- The **void PreSolve()** is used to determine the speed of the conveyor belt at each platform

```
        void PreSolve(b2Contact* contact, const b2Manifold* oldManifold
            ) override
        {
            Test::PreSolve(contact, oldManifold);

            b2Fixture* fixtureA = contact->GetFixtureA();
            b2Fixture* fixtureB = contact->GetFixtureB();

            if (fixtureA == m_platform)
            {
                contact->SetTangentSpeed(5.0f);
            }

            if (fixtureB == m_platform)
            {
                contact->SetTangentSpeed(-5.0f);
            }

            if (fixtureA == m_platform2)
            {
                contact->SetTangentSpeed(5.0f);
            }
            if (fixtureB == m_platform2)
            {
                contact->SetTangentSpeed(-5.0f);
            }

            if (fixtureA == m_platform3)
            {
                contact->SetTangentSpeed(5.0f);
            }
            if (fixtureB == m_platform3)
            {
                contact->SetTangentSpeed(-5.0f);
            }

            if (fixtureA == m_platform4)
            {
                contact->SetTangentSpeed(5.0f);
            }
            if (fixtureB == m_platform4)
            {
                contact->SetTangentSpeed(-5.0f);
            }
```

```
                if (fixtureA == m_platform5)
                {
                        contact->SetTangentSpeed(0.0f);
                }
                if (fixtureB == m_platform5)
                {
                        contact->SetTangentSpeed(0.0f);
                }

        }
```

**Figure 9.18:** *The simulation of a 2 kg boxes of Nutella chocolate jam that are being transported to another location in a warehouse (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_conveyorbelt.cpp****).*

# Chapter 10

# DFSimulatorC++ IV: Potential Energy and Conservation of Energy

*"Je vais et je vient, entre tes reins.." - **Je t'aime Moi Non Plus song***

One general type of energy is potential energy, $U$. Technically, potential energy is energy that can be associated with the configuration (arrangement) of a system of objects that exert forces on one another.

## I. Potential Energy

[DF*] A force is a conservative force if the net work it does on a particle moving around any closed path, from an initial point and then back to that point, is zero. Equivalently, a force is conservative if the net work it does on a particle moving between two points does not depend on the path taken by the particle. The gravitational force and the spring force are conservative forces; the kinetic frictional force and drag force are nonconservative force.

[DF*] Let us send a block sliding across a floor that is not frictionless. During the sliding, a kinetic frictional force from the floor slows the block by transferring energy from its kinetic energy to a type of energy called thermal energy (which has to do with the random motions of atoms and molecule). This energy transfer cannot be reversed. Therefore, thermal energy is not a potential energy.

[DF*] Potential energy is energy that is associated with the configuration of a system in which a conservative force acts. When the conservative force does work $W$ on a particle within the system, the change $\triangle U$ in the potential energy of the system is

$$\triangle U = -W \tag{10.1}$$

If the particle moves from point $x_i$ to point $x_f$, the change in the potential energy of the system is

$$\triangle U = -\int_{x_i}^{x_f} F(x)\, dx \tag{10.2}$$

[DF*] When we throw a tomato upward, the work $W_g$ done on the tomato by the gravitational force is negative because the force transfers energy from the kinetic energy of the tomato. This energy is transferred by the gravitational force to the gravitational potential energy of the tomato-Earth system. The tomato slows, stops, and then begins to fall back down

because of the gravitational force. During the fall, the transfer is reversed: The work $W_g$ done on the tomato by the gravitational force is now positive. That force transfers energy from the gravitational potential energy of the tomato-Earth system to the kinetic energy of the tomato.

For either rise or fall, the change $\triangle U$ in gravitational potential energy defined as being equal to the negative of the work done on the tomato by the gravitational force.

[DF*] In a block-spring system, if we abruptly shove the block to send it moving rightward, the spring force acts leftward and thus does negative work on the block, transferring energy from the kinetic energy of the block to the elastic potential energy of the spring-block system. The block slows and eventually stops, and then begins to move leftward because of the spring force is still leftward. The transfer of energy is then reversed, it is from potential energy of the spring-block system to kinetic energy of the block.

[DF*] The potential energy associated with a system consisting of Earth and a nearby particle is gravitational potential energy. If the particle moves from height $y_i$ to height $y_f$, the change in the gravitational potential energy of the particle-Earth system is

$$\triangle U = mg(y_f - y_i) = mg\triangle y \tag{10.3}$$

[DF*] If the reference point of the particle is set as $y_i = 0$ and the corresponding gravitational potential energy of the system is set as $U_i = 0$, then the gravitational potential energy $U$ when the particle is at any height $y$ is

$$U(y) = mgy \tag{10.4}$$

[DF*] Elastic potential energy is the energy associated with the state of compression or extension of an elastic object. For a spring that exerts a spring force $F = -kx$ when its free and has displacement $x$, the elastic potential energy is

$$U(x) = \frac{1}{2}kx^2 \tag{10.5}$$

[DF*] The reference configuration has the spring at its relaxed length, at which $x = 0$ and $U = 0$.

[DF*] **Example:**

A ball with mass $m = 0.341\ kg$ attached to the end of a thin rod with length $L = 0.452\ m$ and negligible mass. The other end of the rod is pivoted so that the ball can move in a vertical circle. The rod is held horizontally as shown and then given enough of a downward push to cause the ball to swing down and around and just reach the vertically up position, with zero speed there.

(a) How much work is done on the ball by the gravitational force from the initial point to the lowest point?

(b) How much work is done on the ball by the gravitational force from the initial point to the highest point?

(c) How much work is done on the ball by the gravitational force from the initial point to the point on the right level with the initial point?

(d) If the gravitational potential energy of the ball-Earth system is taken to be zero at the initial point, what is the change in the gravitational potential energy when the ball reaches the lowest point?

(e) If the gravitational potential energy of the ball-Earth system is taken to be zero at the initial point, what is the change in the gravitational potential energy when the ball reaches the highest point?

(f) If the gravitational potential energy of the ball-Earth system is taken to be zero at the initial point, what is the change in the gravitational potential energy when the ball reaches the the point on the right level with the initial point?

(g) Suppose the rod were pushed harder so that the ball passed through the highest point with a nonzero speed. Would $\triangle U_g$ from the lowest point to the highest point then be greater, less than, or the same as it was when the ball stopped at the highest point?

**Figure 10.1:** *A ball with mass $m = 0.341$ kg attached to the end of a thin rod, held horizontally so the ball can move in a vertical circle.*

**Solution:**

(a) The only force that does work on the ball is the force of gravity; the force of the rod is perpendicular to the path of the ball and so does no work. In going from its initial position to the lowest point on its path, the ball moves vertically through a distance equal to the length $L$ of the rod, so the work done by the force of gravity is

$$\begin{aligned} W &= mgL \\ &= (0.341)(9.8)(0.452) \\ &= 1.51 \end{aligned}$$

the work is 1.51 $J$.

(b) In going from its initial position to the highest point on its path, the ball moves vertically through a distance equal to $L$, but this time the displacement is upward, opposite the direction of the force of gravity. The work done by the force of gravity is

$$\begin{aligned} W &= -mgL \\ &= -(0.341)(9.8)(0.452) \\ &= -1.51 \end{aligned}$$

the work done is $-1.51$ $J$.

(c) The final position of the ball is at the same height as its initial position. The displacement is horizontal, perpendicular to the force of gravity. The force of gravity does no work during this displacement.

(d) The force of gravity is conservative. The change in the gravitational potential energy of the ball-Earth system is the negative of the work done by gravity

$$\begin{aligned}\triangle U &= -mgL \\ &= -(0.341)(9.8)(0.452) \\ &= -1.51\end{aligned}$$

as the ball foes to the lowest point, the gravitational potential energy of the ball-Earth system is $-1.51\ J$.

(e) Continuing this line of reasoning, we find

$$\begin{aligned}\triangle U &= +mgL \\ &= (0.341)(9.8)(0.452) \\ &= 1.51\end{aligned}$$

as the ball foes to the highest point, the gravitational potential energy of the ball-Earth system is $1.51\ J$.

(f) Continuing this line of reasoning, we have $\triangle U = 0$ as it goes to the point at the same height.

(g) The change in the gravitational potential energy depends only on the initial and final positions of the ball, not on its speed anywhere. The change in the potential energy is the same since the initial and final positions are the same.

[DF*] **Example:**
A thin rod of length $L = 2.0\ m$ and negligible mass, that can pivot about one end to rotate in a vertical circle. A ball of mass $5\ kg$ is attached to the other end. The rod is pulled aside to angle $\theta_0 = 30^0$ and released with initial velocity $\vec{v}_0 = 0$.

(a) As the ball descends to its lowest point, how much does the gravitational force do on it?

(b) What is the change in the gravitational potential energy of the ball-Earth system?

(c) If the gravitational potential energy is taken to be zero at the lowest point, what is its value just as the ball is released?

(d) Do the magnitudes of the answers to (a) through (c) increase, decrease or remain the same if angle $\theta_0$ is increased?

**Solution:**

(a) The main challenge is to obtain the height of the ball (relative to the lowest point of the swing), when a pendulum is pulled aside at angle $\theta$, the vertical component of the rod that has been pulled will be $L\ \cos\ \theta$, thus we will get the height as

$$h = L - L\ \cos\ \theta$$

with angle $\theta$ measured from vertical. The vertical component of the displacement vector

**Figure 10.2:** *The illustration of how to calculate the height of the pendulum bob that is pulled aside at angle $\theta$.*

is downward with magnitude $h$, so we obtain

$$\begin{aligned} W_g &= \vec{F}_g \cdot \vec{d} \\ &= mgh \\ &= mgL(1 - \cos\ \theta) \\ &= (5)(9.8)(2)(1 - \cos\ 30^0) \\ &= 13.1 \end{aligned}$$

The work that the gravitational force do on it is 13.1 $J$.

(b)

$$\begin{aligned} \triangle U &= -W_g \\ &= -mgL(1 - \cos\ \theta) \\ &= -13.1 \end{aligned}$$

The change in the gravitaitonal potential energy of the ball-Earth system is $-13.1$ $J$.

(c) With $y = h$, it will yields

$$\begin{aligned} U &= mgL(1 - \cos\ \theta) \\ &= 13.1 \end{aligned}$$

(d) As the angle increases, we see that the height $h$ increases. If the angle increases then $\cos\ \theta$ decreases, thus $1 - \cos\ \theta$ increases, so the answers to parts (a) and (c) increase, and the absolute value of the answer to part (b) also increases.

## II. Simulation for Potential Energy: Slide Along the Frictionless Loop-the-Loop with Box2D

Taken from problem number 6 on chapter 8, from the book of [6].

A small block of mass $m = 0.032\ kg$ can slide along the frictionless loop-the-loop, with loop radius $R = 12\ cm$. The block is released from rest at point $P$, at height $h = 5.0R$ above the bottom of the loop. How muc does the gravitational force do on the block as the block travels from point $P$ to

(a) Point $Q$

(b) The top of the loop?

(c) If the gravitational potential energy of the block-Earth system is taken to be zero at the bottom of the loop, what is that potential energy when the block is at point $P$?

(d) If the gravitational potential energy of the block-Earth system is taken to be zero at the bottom of the loop, what is that potential energy when the block is at point $Q$?

(e) If the gravitational potential energy of the block-Earth system is taken to be zero at the bottom of the loop, what is that potential energy when the block is at the top of the loop?

(f) If, instead of merely being released, the block is given some initial speed downward along the track, do the answers to (a) through (e), are they increase, decrease, or remain the same?

**Figure 10.3:** *A small block of mass $m = 0.032\ kg$ that is released from rest at point $P$ into the frictionless loop-the-loop.*

**Solution:**

(a) The displacement between the initial point and $Q$ has a vertical component of $h - R$ downward (same direction as $\vec{F}_g$), so (with $h = 5R$) we obtain

$$\begin{aligned} W_g &= \vec{F}_g \cdot \vec{d} \\ &= 4mgR \\ &= 4(0.032)(9.8)(0.12) \\ &= 0.15 \end{aligned}$$

the gravitational force does 0.15 $J$ of work.

(b) The displacement between the initial point and the top of the loop has a vertical component of $h - 2R$ downward (same direction as $\vec{F}_g$), so (with $h = 5R$) we obtain

$$\begin{aligned} W_g &= \vec{F}_g \cdot \vec{d} \\ &= 3mgR \\ &= 3(0.032)(9.8)(0.12) \\ &= 0.11 \end{aligned}$$

the gravitational force does 0.11 $J$ of work.

(c) With $y = h = 5R$, at $P$ we find

$$\begin{aligned} U &= 5mgR \\ &= 5(0.032)(9.8)(0.12) \\ &= 0.19 \end{aligned}$$

the potential energy when the block at $P$ is 0.19 $J$.

(d) With $y = R$, at $Q$ we will have

$$\begin{aligned} U &= mgR \\ &= (0.032)(9.8)(0.12) \\ &= 0.038 \end{aligned}$$

the potential energy when the block at $Q$ is 0.038 $J$.

(e) With $y = 2R$, at the top of the loop, we find

$$\begin{aligned} U &= 2mgR \\ &= 2(0.032)(9.8)(0.12) \\ &= 0.075 \end{aligned}$$

the potential energy when the block at $Q$ is 0.075 $J$.

(f) If the block is given initial velocity when released, the potential energy won't change, since the formula does not depend on $v_i$, the initial velocity.

```
#include "test.h"

class LooptheLoop: public Test
{
        public:
        LooptheLoop()
        {
                m_speed = 50.0f;

                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);
```

```
            b2EdgeShape shape;

            b2FixtureDef fd;
            fd.shape = &shape;
            fd.density = 0.0f;
            fd.friction = 0.6f;

            shape.SetTwoSided(b2Vec2(-20.0f, 0.0f), b2Vec2(20.0f,
                0.0f));
            ground->CreateFixture(&fd);

        }
        // Create the down line
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;

            b2FixtureDef fd;
            fd.shape = &shape;
            fd.density = 0.0f;
            fd.friction = 0.0f;

            shape.SetTwoSided(b2Vec2(20.0f, 0.0f), b2Vec2(20.0f,
                -60.0f));
            ground->CreateFixture(&fd);

        }
        //
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;

            b2FixtureDef fd;
            fd.shape = &shape;
            fd.density = 0.0f;
            fd.friction = 0.0f;

            // hs0[11] will define the parameter t for each of the
                 vertices that create the circle
            // Use parametric equation to get circle shape of
                radius 24
            // (cos t, sin t)
            float hs0[11] = {b2_pi, 1.1f*b2_pi, 1.2f*b2_pi, 1.3f*
```

```
                    b2_pi, 1.4f*b2_pi, 1.5f*b2_pi, 1.6f*b2_pi, 1.7f*
                    b2_pi, 1.8f*b2_pi, 1.9f*b2_pi};
            float yc0 = 0.99f*b2_pi;
            for (int32 i = 0; i < 11; ++i)
            {
                float y0 = hs0[i];
                shape.SetTwoSided(b2Vec2(44+24*cos(yc0),
                    -59+24*sin(yc0)), b2Vec2(44+24*cos(yc0 +
                    0.1f*b2_pi), -59+24*sin(yc0 + 0.1f*b2_pi))
                    );
                ground->CreateFixture(&fd);
                yc0 = y0;
            }

            // Use parametric equation to get circle shape of
                radius 12
            // (cos t, sin t)
            float hs2[12] = {0.0f, 0.1f*b2_pi, 0.2f*b2_pi, 0.3f*
                b2_pi, 0.4f*b2_pi, 0.5f*b2_pi, 0.6f*b2_pi, 0.7f*
                b2_pi, 0.8f*b2_pi, 0.9f*b2_pi, b2_pi};
            float yc1 = 0.0f;
            for (int32 i = 0; i < 12; ++i)
            {
                float y2 = hs2[i];
                shape.SetTwoSided(b2Vec2(56+12*cos(yc1),
                    -60+12*sin(yc1)), b2Vec2(56+12*cos(yc1 +
                    0.1f*b2_pi), -60+12*sin(yc1 + 0.1f*b2_pi))
                    );
                ground->CreateFixture(&fd);
                yc1 = y2;
            }
        }
        // Boxes
        {
            b2PolygonShape boxShape1;
            boxShape1.SetAsBox(0.5f, 0.5f);

            b2FixtureDef boxFixtureDef1;
            boxFixtureDef1.restitution = 0.0f;
            boxFixtureDef1.density = 0.5f/8.0f; // this will
                affect the left elevator mass
            boxFixtureDef1.friction = 0.0f; //frictionless
            boxFixtureDef1.shape = &boxShape1;

            b2BodyDef boxBodyDef1;
            boxBodyDef1.type = b2_dynamicBody;

            boxBodyDef1.position.Set(20.5f, 0.5f);
```

```
            m_box = m_world->CreateBody(&boxBodyDef1);
            b2Fixture *boxFixture1 = m_box->CreateFixture(&
                boxFixtureDef1);
            //m_box->SetLinearVelocity(b2Vec2(0.0f, -1000.0f));

        }

        // Car
        {
            b2PolygonShape chassis;
            b2Vec2 vertices[8];
            vertices[0].Set(-1.5f, -0.5f);
            vertices[1].Set(1.5f, -0.5f);
            vertices[2].Set(1.5f, 0.0f);
            vertices[3].Set(0.0f, 0.9f);
            vertices[4].Set(-1.15f, 0.9f);
            vertices[5].Set(-1.5f, 0.2f);
            chassis.Set(vertices, 6);

            b2CircleShape circle;
            circle.m_radius = 0.4f;

            b2BodyDef bd;
            bd.type = b2_dynamicBody;
            bd.position.Set(20.0f, 1.0f);
            m_car = m_world->CreateBody(&bd);
            m_car->CreateFixture(&chassis, 1.0f);

            b2FixtureDef fd;
            fd.shape = &circle;
            fd.density = 1.0f;
            fd.friction = 0.9f;

            bd.position.Set(19.0f, 0.35f);
            m_wheel1 = m_world->CreateBody(&bd);
            m_wheel1->CreateFixture(&fd);

            bd.position.Set(21.0f, 0.4f);
            m_wheel2 = m_world->CreateBody(&bd);
            m_wheel2->CreateFixture(&fd);

            b2WheelJointDef jd;
            b2Vec2 axis(0.0f, 1.0f);

            float mass1 = m_wheel1->GetMass();
            float mass2 = m_wheel2->GetMass();

            float hertz = 4.0f;
```

```
                        float dampingRatio = 0.7f;
                        float omega = 2.0f * b2_pi * hertz;

                        jd.Initialize(m_car, m_wheel1, m_wheel1->GetPosition()
                            , axis);
                        jd.motorSpeed = 0.0f;
                        jd.maxMotorTorque = 20.0f;
                        jd.enableMotor = true;
                        jd.stiffness = mass1 * omega * omega;
                        jd.damping = 2.0f * mass1 * dampingRatio * omega;
                        jd.lowerTranslation = -0.25f;
                        jd.upperTranslation = 0.25f;
                        jd.enableLimit = true;
                        m_spring1 = (b2WheelJoint*)m_world->CreateJoint(&jd);

                        jd.Initialize(m_car, m_wheel2, m_wheel2->GetPosition()
                            , axis);
                        jd.motorSpeed = 0.0f;
                        jd.maxMotorTorque = 10.0f;
                        jd.enableMotor = false;
                        jd.stiffness = mass2 * omega * omega;
                        jd.damping = 2.0f * mass2 * dampingRatio * omega;
                        jd.lowerTranslation = -0.25f;
                        jd.upperTranslation = 0.25f;
                        jd.enableLimit = true;
                        m_spring2 = (b2WheelJoint*)m_world->CreateJoint(&jd);
                }
                m_time = 0.0f;
        }
        bool m_fixed_camera;
        float m_time;
        b2Body* m_box;
        b2Body* m_car;
        b2Body* m_wheel1;
        b2Body* m_wheel2;

        float m_speed;
        b2WheelJoint* m_spring1;
        b2WheelJoint* m_spring2;

        void Keyboard(int key) override
        {
                switch (key)
                {
                        case GLFW_KEY_F:
                        m_box->SetLinearVelocity(b2Vec2(5000.0f, 0.0f));
                        break;
                        case GLFW_KEY_A:
```

```
                m_spring1->SetMotorSpeed(m_speed);
                break;
                case GLFW_KEY_S:
                m_spring1->SetMotorSpeed(0.0f);
                break;
                case GLFW_KEY_D:
                m_spring1->SetMotorSpeed(-m_speed);
                break;
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                        g_camera.m_center = b2Vec2(2.0f, -10.0f);
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
        }
}

void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f;
        b2MassData massData1 = m_car->GetMassData();
        b2MassData massData2 = m_box->GetMassData();
        b2Vec2 position1 = m_car->GetPosition();
        b2Vec2 velocity1 = m_car->GetLinearVelocity();
        b2Vec2 position2 = m_box->GetPosition();
        b2Vec2 velocity2 = m_box->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
            /tracking");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Keys: left = a, brake
            = s, right = d, hz down = q, hz up = e");
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car Mass = %4.2f",
            massData1.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car position = (%4.1f,
             %4.1f)", position1.x, position1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car velocity = (%4.1f,
             %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %4.2f",
                    massData2.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
                     %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
                     %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;


                g_camera.m_center.x = m_car->GetPosition().x;
                Test::Step(settings);
                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_box->GetPosition();
                       g_camera.m_zoom = 3.3f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new LooptheLoop;
        }


    };

    static int testIndex = RegisterTest("Potential Energy and Conservation of
        Energy", "Loop-the-Loop", LooptheLoop::Create);
```

**C++ Code 71:** *tests/potentialenergy_looptheloop.cpp "Potential Energy Frictionless Loop-the-Loop Box2D"*

The simulation also contains a car, it is to compare how a car and a block of mass $m$ will move when released into frictionless loop-the-loop, you can comment the **m_car->CreateFixture(&chassis, 1.0f);**, **m_wheel1->CreateFixture(&fd);**, **m_wheel2->CreateFixture(&fd);**, **m_spring1 = (b2WheelJoint*)m_world->CreateJoint(&jd);**, and **m_spring2 = (b2WheelJoint*)m_world->CreateJoint(&jd);** to remove the car, the wheels, and the spring joining the wheel and the car from the simulation.

Some explanations for the codes:

- To create a perfect circle shape by connecting two lines, we can refer to parametric equation of a circle, that is
$$(x, y) = (\cos\ t, \sin\ t)$$
with $t$ is the parameter that determines the angle of the circle in radian / $\pi$ terms.
For the first circle, **float hs0[11]** contain all the values for the parameter of $t$, starting from $\pi$ and stop at $1.8\pi$, with radius of 24.
Now if you have **shape.SetTwoSided(b2Vec2(44+24*cos(yc0), -59+24*sin(yc0)), b2Vec2(44+24*cos(yc0**

**+ 0.1f*b2_pi), -59+24*sin(yc0 + 0.1f*b2_pi)));** it is a command to create line between two points in a circle, with difference of $\triangle t = 0.1\pi$, we will write the parametric mathematical equation of this circle as

$$(x, y) = (44 + 24\cos(t), -59 + 24\sin(t))$$

thus the circle will have the center at $(44, -59)$ and radius of 24. The code to make the second circle with radius of 12 can be easily comprehended after you comprehend this one.

```
float hs0[11] = {b2_pi, 1.1f*b2_pi, 1.2f*b2_pi, 1.3f*b2_pi, 1.4
    f*b2_pi, 1.5f*b2_pi, 1.6f*b2_pi, 1.7f*b2_pi, 1.8f*b2_pi,
    1.9f*b2_pi};
float yc0 = 0.99f*b2_pi;
for (int32 i = 0; i < 11; ++i)
{
       float y0 = hs0[i];
       shape.SetTwoSided(b2Vec2(44+24*cos(yc0), -59+24*sin(yc0)
           ), b2Vec2(44+24*cos(yc0 + 0.1f*b2_pi), -59+24*sin(
           yc0 + 0.1f*b2_pi)));
       ground->CreateFixture(&fd);
       yc0 = y0;
}

// Use parametric equation to get circle shape of radius 12
// (cos t, sin t)
float hs2[12] = {0.0f, 0.1f*b2_pi, 0.2f*b2_pi, 0.3f*b2_pi, 0.4f
    *b2_pi, 0.5f*b2_pi, 0.6f*b2_pi, 0.7f*b2_pi, 0.8f*b2_pi, 0.9
    f*b2_pi, b2_pi};
float yc1 = 0.0f;
for (int32 i = 0; i < 12; ++i)
{
       float y2 = hs2[i];
       shape.SetTwoSided(b2Vec2(56+12*cos(yc1), -60+12*sin(yc1)
           ), b2Vec2(56+12*cos(yc1 + 0.1f*b2_pi), -60+12*sin(
           yc1 + 0.1f*b2_pi)));
       ground->CreateFixture(&fd);
       yc1 = y2;
}
```

- This code to create a car can be obtained from **Car** example at the Box2D version 2.4.1 official source code. Basically, to create a 2D car that can run with 2 wheels, back and front, we will first define the shape of the car' body, then the two wheels should be located $x + 1$ for front wheel and $x - 1$ for back wheel, with $x$ is the car body position, both the wheels will be combined into 1 with **b2WheelJoint**, we can set the damping value, mass, or the angular velocity of the wheel as well.

```
b2PolygonShape chassis;
b2Vec2 vertices[8];
vertices[0].Set(-1.5f, -0.5f);
vertices[1].Set(1.5f, -0.5f);
```

```
        vertices[2].Set(1.5f, 0.0f);
        vertices[3].Set(0.0f, 0.9f);
        vertices[4].Set(-1.15f, 0.9f);
        vertices[5].Set(-1.5f, 0.2f);
        chassis.Set(vertices, 6);

        b2CircleShape circle;
        circle.m_radius = 0.4f;

        b2BodyDef bd;
        bd.type = b2_dynamicBody;
        bd.position.Set(20.0f, 1.0f);
        m_car = m_world->CreateBody(&bd);
        m_car->CreateFixture(&chassis, 1.0f);

        b2FixtureDef fd;
        fd.shape = &circle;
        fd.density = 1.0f;
        fd.friction = 0.9f;

        bd.position.Set(19.0f, 0.35f);
        m_wheel1 = m_world->CreateBody(&bd);
        m_wheel1->CreateFixture(&fd);

        bd.position.Set(21.0f, 0.4f);
        m_wheel2 = m_world->CreateBody(&bd);
        m_wheel2->CreateFixture(&fd);

        b2WheelJointDef jd;
        b2Vec2 axis(0.0f, 1.0f);

        float mass1 = m_wheel1->GetMass();
        float mass2 = m_wheel2->GetMass();

        float hertz = 4.0f;
        float dampingRatio = 0.7f;
        float omega = 2.0f * b2_pi * hertz;

        jd.Initialize(m_car, m_wheel1, m_wheel1->GetPosition(), axis);
        jd.motorSpeed = 0.0f;
        jd.maxMotorTorque = 20.0f;
        jd.enableMotor = true;
        jd.stiffness = mass1 * omega * omega;
        jd.damping = 2.0f * mass1 * dampingRatio * omega;
        jd.lowerTranslation = -0.25f;
        jd.upperTranslation = 0.25f;
        jd.enableLimit = true;
        m_spring1 = (b2WheelJoint*)m_world->CreateJoint(&jd);
```

```
jd.Initialize(m_car, m_wheel2, m_wheel2->GetPosition(), axis);
jd.motorSpeed = 0.0f;
jd.maxMotorTorque = 10.0f;
jd.enableMotor = false;
jd.stiffness = mass2 * omega * omega;
jd.damping = 2.0f * mass2 * dampingRatio * omega;
jd.lowerTranslation = -0.25f;
jd.upperTranslation = 0.25f;
jd.enableLimit = true;
m_spring2 = (b2WheelJoint*)m_world->CreateJoint(&jd);
```

**Figure 10.4:** *The simulation of a block and a car dropped and then slide along the frictionless loop-the-loop (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/potentialenergy_looptheloop.cpp****).*

## III. Conservation of Mechanical Energy

[DF*] The mechanical energy $E_{mec}$ of a system is the sum of its kinetic energy $K$ and potential energy $U$:

$$E_{mec} = K + U \tag{10.6}$$

[DF*] When a conservative force does work $W$ on an object within the system, that force transfers energy between kinetic energy $K$ of the object and potential energy $U$ of the system. The change $\triangle K$ in kinetic energy is

$$\triangle K = W$$

and the change $\triangle U$ in potential energy is

$$\triangle U = -W$$

Combining both energies, we find that

$$\triangle K = -\triangle U$$

We can rewrite as

$$K_2 - K_1 = -(U_2 - U_1)$$

[DF*] An isolated system is one in which no external force causes energy changes. If only conservative forces do work within an isolated system, then the echanical energy $E_{mec}$ of the system cannot change. This principle of conservation of mechanical energy is written as

$$K_2 + U_2 = K_1 + U_1 \tag{10.7}$$

in which the subscript refer to different instants during an energy tranfer process. This conservation principle can also be written as

$$\triangle E_{mec} = \triangle K + \triangle U = 0 \tag{10.8}$$

[DF*] Pendulum swings can be an example in which the principle of conservation of mechanical energy can be applied. As a pendulum swings, the energy of the pendulum-Earth system is transferred back and forth between kinetic energy $K$ and gravitational potential energy $U$, with the sum $K + U$ being constant.

[DF*] The huge advantage of using the conservation of energy instead of Newton's laws of motion is that we can jump from the initial state to the final state without considering all the intermediate motion.

**Figure 10.5:** *One full cycle of pendulum motion is shown. During the cycle the values of the pendulum-Earth system vary as the bob rises and falls, but the mechanical energy $E_{mec}$ of the system remains constant. The energy $E_{mec}$ can be described as continuously shifting between the kinetic and potential forms. If the swinging involved a frictional force at the point where the pendulum is attached to the ceiling or a drag force due to the air, then $E_{mec}$ would not be conserved, and eventually the pendulum would stop.*

## IV. Simulation for Conservation of Mechanical Energy: Marble Fired Upward Using a Spring with Box2D

Taken from problem no 13 chapter 8 of [6].

A 5 $g$ marble is fired vertically upward using a spring gun. The spring must be compressed 8 $cm$ if the marble is to just reach a target 20 $m$ above the marble's position on the compressed spring.

(a) What is the change $\triangle U_g$ in the gravitational potential energy of the marble-Earth system during the 20 $m$ ascent?

(b) What is the change $\triangle U_s$ in the elastic potential energy of the spring during its launch of the marble?

(c) What is the spring constant of the spring?

**Solution:**

(a) We take the reference point for gravitational potential energy at the position of the marble when the spring is compressed.
The gravitational potential energy when the marble is at the top of its motion is $U_g = mgh$,

where $h = 20\ m$ is the height of the highest point. Thus,

$$\begin{aligned} U_g &= (0.005)(9.8)(20) \\ &= 0.98 \end{aligned}$$

the gravitaional potential energy is $0.98\ J$.

(b) Since the kinetic energy is zero at the release point and at the highest point, then conservation of mechanical energy implies $\triangle U_g + \triangle U_s = 0$, where $\triangle U_s$ is the change in the spring's elastic potential energy. Therefore

$$\triangle U_s = -\triangle U_g = -0.98$$

the change in the spring's elastic potential energy is $-0.98\ J$.

(c) We take the spring potential energy to be zero when the spring is relaxed. Then, our result in the previous part implies that its initial potential energy is $U_s = 0.98\ J$. This must be $\frac{1}{2}kx^2$, where $k$ is the spring constant and $x$ is the initial compression. Consequently,

$$\begin{aligned} k &= \frac{2U_s}{x^2} \\ &= \frac{2(0.98)}{(0.08)^2} \\ &= 310 \end{aligned}$$

the spring constant is $310\ N/m$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class MarbleVerticalspringupward : public Test
{
        public:
        MarbleVerticalspringupward()
        {
                b2Body* ground = NULL;
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                            0.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                // Create the wall surrounding the spring
```

```
b2BodyDef bd2;
bd2.type = b2_staticBody;
bd2.angularDamping = 0.1f;

bd2.position.Set(-0.05f, 5.5f);
b2Body* leftwall = m_world->CreateBody(&bd2);

b2PolygonShape shape2;
shape2.SetAsBox(0.5f, 5.5f);
leftwall->CreateFixture(&shape2, 5.0f);

b2BodyDef bd3;
bd3.type = b2_staticBody;
bd3.angularDamping = 0.1f;

bd3.position.Set(2.05f, 5.5f);
b2Body* rightwall = m_world->CreateBody(&bd3);

b2PolygonShape shape3;
shape3.SetAsBox(0.5f, 5.5f);
rightwall->CreateFixture(&shape3, 5.0f);

// Create a static body as the box for the spring
b2BodyDef bd1;
bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.1f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.0f;
boxFixtureDef.density = 1.5f; // this will affect the spring
    mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(1.0f, 11.0f); // the box will be
```

```
    located in (1,11.0)

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
    ;
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
    m_minLength
m_maxLength = 12.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create the marble ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 0.5f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(1.0f, 12.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);
```

```
        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_ball;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_W:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, 1000.0f), true)
                ;
            break;
            case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, -800.0f), true
                );
            break;
            case GLFW_KEY_T:
            m_time = 0.0f;
            break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
            ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
            f"))
        {
            m_length = m_joint->SetLength(m_length);
        }

        if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
            ))
        {
            float stiffness;
            float damping;
```

```
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            ImGui::End();
        }
        void Step(Settings& settings) override
        {
            b2MassData massData = m_box->GetMassData();
            b2MassData massData2 = m_ball->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2Vec2 position2 = m_ball->GetPosition();
            b2Vec2 velocity2 = m_ball->GetLinearVelocity();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
                 60 Hertz
            float m = massData.mass;
            float g = 9.8f;
            float y = 11.0f;
            // y = the position at which when we place the mass it would
                not move / equlibrium position
            // y = y position of the ceiling - m_minLength - initial y
                position of the mass
            float k = 310.0f;
            float y_eq = 11.0f;

            g_debugDraw.DrawString(5, m_textLine, "Press W to apply force
                 1,000 N upward / S to apply force 1,000 N downward");
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
                );
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
```

```
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Marble Mass = %4.2f",
                    massData2.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Marble position =
                    (%4.1f, %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Marble velocity =
                    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Spring position =
                    (%4.1f, %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Spring velocity =
                    (%4.1f, %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Spring Mass = %4.2f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
                     = %4.1f", k);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
                    for the spring, y = %4.1f", y_eq);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f\n", velocity2.y, position2.y,
                    position2.y-11.0f);

                Test::Step(settings);
            }
            static Test* Create()
            {
                return new MarbleVerticalspringupward;
            }

    };

    static int testIndex = RegisterTest("Potential Energy and Conservation of
        Energy", "Vertical Spring and Marble", MarbleVerticalspringupward::
        Create);
```

**C++ Code 72:** *tests/potentialenergy_verticalspringupward.cpp "Marble Fired Upward with Spring Box2D"*

**Figure 10.6:** *The simulation of a marble being fired vertically upward by a spring gun (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/potentialenergy_verticalspringupward.cpp****).*

```
-1.14 31.57 20.57
-1.30 31.54 20.54
-1.47 31.52 20.52
-1.63 31.49 20.49
-1.79 31.46 20.46
-1.96 31.43 20.43
-2.12 31.39 20.39
-2.28 31.36 20.36
-2.45 31.32 20.32
-2.61 31.27 20.27
-2.77 31.23 20.23
-2.94 31.18 20.18
-3.10 31.13 20.13
-3.26 31.07 20.07
-3.43 31.01 20.01
-3.59 30.95 19.95
-3.75 30.89 19.89
-3.92 30.83 19.83
-4.08 30.76 19.76
-4.24 30.69 19.69
-4.41 30.61 19.61
-4.57 30.54 19.54
-4.73 30.46 19.46
-4.90 30.38 19.38
-5.06 30.29 19.29
-5.22 30.21 19.21
-5.39 30.12 19.12
-5.55 30.02 19.02
-5.71 29.93 18.93
-5.88 29.83 18.83
-6.04 29.73 18.73
-6.20 29.63 18.63
-6.37 29.52 18.52
-6.53 29.41 18.41
-6.69 29.30 18.30
-6.86 29.19 18.19
```

**Figure 10.7:** *The data printed on the xterm while running the simulation, the first column represents the marble velocity toward y axis, the second column represents the position of the marble, and the third column represents the distance of the marble toward its' initial position befored being fired up.*

## V. Simulation for Conservation of Mechanical Energy: Runaway Truck with Failed Brakes with Box2D

Taken from problem number 15 chapter 8 of [6].

A runaway truck with failed brakes is moving downgrade at 130 $km/h$ just before the driver steers the truck up a frictionless emergency escape ramp with an inclination of $\theta = 15^0$. The truck's mass is $1.2 \times 10^4$.

(a) What minimum length $L$ must the ramp have if the truck is to stop (momentarily) along it? (Assume the truck is a particle, and justify that assumption).

(b) Does the minimum length $L$ increase, decrease, or remain the same if the truck's mass is decreased?

(c) Does the minimum length $L$ increase, decrease, or remain the same if the truck's speed is decreased?

**Solution:**

(a) We neglect any work done by friction. We work with SI units, so the speed is converted

$$v = 130\ km/h = 130\left(\frac{1\ km \times 1000\ m/km}{1\ h \times 3600\ s/h}\right) = 36.1$$

the speed is 36.1 $m/s$.
We use

$$K_f + U_f = K_i + U_i$$

with $U_i = 0,\ U_f = mgh$ and $K_f = 0$. Since $K_i = \frac{1}{2}mv^2$, where $v$ is the initial speed of the truck, we obtain

$$\begin{aligned} K_f + U_f &= K_i + U_i \\ 0 + U_f &= K_i + 0 \\ mgh &= \frac{1}{2}mv^2 \\ h &= \frac{v^2}{2g} \\ &= \frac{(36.1)^2}{2(9.8)} \\ &= 66.5 \end{aligned}$$

If $L$ is the length of the ramp, then

$$L\ \sin\ 15^0 = 66.5$$

so that

$$L = \frac{66.5}{\sin\ 15^0} = 257$$

Therefore, the ramp must be about 257 $m$ long if friction is negligible.

(b) The answer do not depend on the mass of the truck. They remain the same if the mass is reduced.

(c) If the speed is decreased, $h$ and $L$ both decrease (note that $h$ is proportional (has linear relation) to the square of the speed and that $L$ is proportional to $h$).

```
#include "test.h"
#define DEGTORAD 0.0174532925199432957f

class RunawayTruck: public Test
{
	public:
	RunawayTruck()
	{
		m_speed = 50.0f;

		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.density = 0.0f;
			fd.friction = 0.6f;

			shape.SetTwoSided(b2Vec2(-20.0f, 0.0f), b2Vec2(20.0f,
				0.0f));
			ground->CreateFixture(&fd);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.density = 0.0f;
			fd.friction = 0.6f;

			float L = 257.0f;
			float costheta = cosf(15*DEGTORAD);
			float sintheta = sinf(15*DEGTORAD);
```

```
        shape.SetTwoSided(b2Vec2(-150-costheta*L,-95+
            sintheta*L), b2Vec2(-150.0f,-95.0f));
        ground->CreateFixture(&fd);

    }
    // Create the left triangle, theta = 15, L = 30
    float L = 257.0f;
    float costheta = cosf(15*DEGTORAD);
    float sintheta = sinf(15*DEGTORAD);
    b2ChainShape chainShape2;
    b2Vec2 vertices2[] = {b2Vec2(-150-costheta*L,-95), b2Vec2
        (-150-costheta*L,-95+sintheta*L), b2Vec2(-150,-95)};
    chainShape2.CreateLoop(vertices2, 3);

    b2FixtureDef groundFixtureDef2;
    groundFixtureDef2.density = 0;
    groundFixtureDef2.shape = &chainShape2;

    b2BodyDef groundBodyDef2;
    groundBodyDef2.type = b2_staticBody;

    b2Body *groundBody2 = m_world->CreateBody(&groundBodyDef2);
    b2Fixture *groundBodyFixture2 = groundBody2->CreateFixture(&
        groundFixtureDef2);

    // Create the right triangle
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-197,-120), b2Vec2(-20,-120),
        b2Vec2(-20,0)};
    chainShape.CreateLoop(vertices, 3);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    // Car
    {
        b2PolygonShape chassis;
        b2Vec2 vertices[8];
        vertices[0].Set(-4.5f, -0.5f);
        vertices[1].Set(4.5f, -0.5f);
```

```
vertices[2].Set(4.5f, 0.5f);
vertices[3].Set(0.0f, 0.5f);
vertices[4].Set(-4.05f, 0.2f);
vertices[5].Set(-4.6f, 0.0f);
chassis.Set(vertices, 6);

b2CircleShape circle;
circle.m_radius = 0.4f;

b2BodyDef bd;
bd.type = b2_dynamicBody;
bd.position.Set(-23.0f, 1.0f);
m_car = m_world->CreateBody(&bd);
m_car->CreateFixture(&chassis, 10.0f);

b2FixtureDef fd;
fd.shape = &circle;
fd.density = 1.0f;
fd.friction = 0.0f;

bd.position.Set(-26.0f, 0.35f);
m_wheel1 = m_world->CreateBody(&bd);
m_wheel1->CreateFixture(&fd);

bd.position.Set(-24.0f, 0.4f);
m_wheel2 = m_world->CreateBody(&bd);
m_wheel2->CreateFixture(&fd);

bd.position.Set(-20.0f, 0.4f);
m_wheel3 = m_world->CreateBody(&bd);
m_wheel3->CreateFixture(&fd);

b2WheelJointDef jd;
b2Vec2 axis(0.0f, 1.0f);

float mass1 = m_wheel1->GetMass();
float mass2 = m_wheel2->GetMass();
float mass3 = m_wheel3->GetMass();

float hertz = 4.0f;
float dampingRatio = 0.7f;
float omega = 2.0f * b2_pi * hertz;

jd.Initialize(m_car, m_wheel1, m_wheel1->GetPosition()
    , axis);
jd.motorSpeed = 0.0f;
jd.maxMotorTorque = 200.0f;
jd.enableMotor = true;
```

```
                        jd.stiffness = mass1 * omega * omega;
                        jd.damping = 2.0f * mass1 * dampingRatio * omega;
                        jd.lowerTranslation = -0.25f;
                        jd.upperTranslation = 0.25f;
                        jd.enableLimit = true;
                        m_spring1 = (b2WheelJoint*)m_world->CreateJoint(&jd);

                        jd.Initialize(m_car, m_wheel2, m_wheel2->GetPosition()
                            , axis);
                        jd.motorSpeed = 0.0f;
                        jd.maxMotorTorque = 100.0f;
                        jd.enableMotor = false;
                        jd.stiffness = mass2 * omega * omega;
                        jd.damping = 2.0f * mass2 * dampingRatio * omega;
                        jd.lowerTranslation = -0.25f;
                        jd.upperTranslation = 0.25f;
                        jd.enableLimit = true;
                        m_spring2 = (b2WheelJoint*)m_world->CreateJoint(&jd);

                        jd.Initialize(m_car, m_wheel3, m_wheel3->GetPosition()
                            , axis);
                        jd.motorSpeed = 0.0f;
                        jd.maxMotorTorque = 100.0f;
                        jd.enableMotor = false;
                        jd.stiffness = mass2 * omega * omega;
                        jd.damping = 2.0f * mass2 * dampingRatio * omega;
                        jd.lowerTranslation = -0.25f;
                        jd.upperTranslation = 0.25f;
                        jd.enableLimit = true;
                        m_spring3 = (b2WheelJoint*)m_world->CreateJoint(&jd);
                        //m_car->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));

                }
                m_time = 0.0f;
        }
        bool m_fixed_camera;
        float m_time;
        b2Body* m_box;
        b2Body* m_car;
        b2Body* m_wheel1;
        b2Body* m_wheel2;
        b2Body* m_wheel3;

        float m_speed;
        b2WheelJoint* m_spring1;
        b2WheelJoint* m_spring2;
        b2WheelJoint* m_spring3;
```

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_A:
                m_spring1->SetMotorSpeed(m_speed);
                break;
                case GLFW_KEY_S:
                m_spring1->SetMotorSpeed(0.0f);
                break;
                case GLFW_KEY_D:
                m_spring1->SetMotorSpeed(-m_speed);
                break;
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                        g_camera.m_center = b2Vec2(2.0f, -10.0f);
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
        }
}

void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f;
        b2MassData massData1 = m_car->GetMassData();
        b2Vec2 position1 = m_car->GetPosition();
        b2Vec2 velocity1 = m_car->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
            /tracking");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Keys: left = a, brake
            = s, right = d, hz down = q, hz up = e");
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car Mass = %4.2f",
            massData1.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car position = (%4.1f,
             %4.1f)", position1.x, position1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Car velocity = (%4.1f,
```

```
                    %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;

                g_camera.m_center.x = m_car->GetPosition().x;
                Test::Step(settings);
                if(!m_fixed_camera)
                {
                        g_camera.m_center = m_car->GetPosition();
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new RunawayTruck;
        }


};

static int testIndex = RegisterTest("Potential Energy and Conservation of
    Energy", "Runaway Truck", RunawayTruck::Create);
```

**C++ Code 73:** *tests/potentialenergy_runawaytruck.cpp "Runaway Truck with Failed Brakes Box2D"*

## VI. Simulation for Conservation of Mechanical Energy: Spring Attached to a Breadbox on Frictionless Incline with Box2D

Taken from problem number 30 chapter 8 of [6].

A 2 $kg$ breadbox on a frictionless incline of angle $\theta = 40^0$ is connected, by a cord that runs over a pulley, to a light spring of spring constant $k = 120\ N/m$. The box is released from rest when the spring is unstretched. Assume that the pulley is massless and frictionless

(a) What is the speed of the box when it has moved 10 $cm$ down the incline?

(b) How far down the incline from its point of release. does the box slide before momentarily stopping, and what are the magnitude?

(c) What is the direction (up or down the incline) of the box's acceleration at the instant the box momentarily stops?

**Solution:**

(a) We take the original height of the box to be the $y = 0$ reference level and observe that, in general, the height of the box (when the box has moved a distance $d$ downhill) is

$$y = -d\ \sin\ 40^0$$

**Figure 10.8:** *The simulation for a runaway truck with failed brakes then the trucks went up to escape ramp (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/potentialenergy_runawaytruck.cpp****).*

**Figure 10.9:** *A breadbox on a frictionless incline of angle $\theta = 40^0$ is connected by pulley to a spring of constant $k = 120\ N/m$.*

Using the conservation of energy, we have

$$\begin{aligned}
K_i + U_i &= K + U \\
0 + 0 &= \frac{1}{2}mv^2 + mgy + \frac{1}{2}kd^2 \\
0 &= \frac{1}{2}2v^2 + 2(9.8)(-0.1\ \sin\ 40^0) + \frac{1}{2}(120)(0.1)^2 \\
v^2 &= 1.25986 - 0.6 \\
v &= \sqrt{0.65986} \\
v &= 0.8123
\end{aligned}$$

Therefore, with $d = 0.10\ m$, we obtain $v = 0.81\ m/s$.

(b) We look for value of $d \neq 0$ such that $K = 0$

$$\begin{aligned}
K_i + U_i &= K + U \\
0 + 0 &= 0 + mgy + \frac{1}{2}kd^2 \\
0 &= mg(-d\ \sin\ 40^0) + \frac{1}{2}kd^2 \\
mgd\ \sin\ 40^0 &= \frac{1}{2}kd^2 \\
2(9.8)d\ \sin\ 40^0 &= \frac{1}{2}(120)d^2 \\
12.5986 &= 60d \\
d &= 0.209977
\end{aligned}$$

Thus, we obtain $d - 0.21\ m$ before the breadbox momentarily stopping.

(c) The uphill force is caused by the spring (Hooke's law) and has magnitude $kd = 25.2\ N$, we will fix the positive magnitude for up the incline direction, and negative magnitude for force moving down the incline. The downhill force is the component of gravity

$$mg\ \sin\ 40^0 = 12.6$$

Thus, the net force on the box is

$$\begin{aligned}
\sum F &= kd - mg\ \sin\ 40^0 \\
&= 25.2 - 12.6 \\
&= 12.6
\end{aligned}$$

positive means it is moving uphill, with

$$\begin{aligned}
a &= \frac{F}{m} \\
&= \frac{12.6}{2} \\
&= 6.3
\end{aligned}$$

the magnitude of the box's acceleration is $6.3\ m/s^2$.

(d) Since the magnitude is positive, then the direction for the acceleration is up the incline.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>
#define DEGTORAD 0.0174532925199432957f

class SpringIncline : public Test
{
	public:
	SpringIncline()
	{
		b2Body* ground = NULL;
		{// Create the pulley
			b2BodyDef bdp;
			ground = m_world->CreateBody(&bdp);

			b2CircleShape circle;
			circle.m_radius = 0.5f;

			circle.m_p.Set(12.0f, 0.5f); // circle with center of
				(12,0.5)
			ground->CreateFixture(&circle, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(12.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(12.0f, 0.0f), b2Vec2(12.0f
				,-20.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);
```

```
        float L = 30.0f;
        float costheta = cosf(40*DEGTORAD);
        float sintheta = sinf(40*DEGTORAD);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(35-costheta*L,-19+sintheta*L
            ), b2Vec2(35,-19));
        ground->CreateFixture(&shape, 0.0f);
    }

    // Create the left triangle, theta = 40, L = 30
    float L = 30.0f;
    float costheta = cosf(40*DEGTORAD);
    float sintheta = sinf(40*DEGTORAD);
    b2ChainShape chainShape2;
    b2Vec2 vertices2[] = {b2Vec2(35-costheta*L,-19), b2Vec2(35-
        costheta*L,-19+sintheta*L), b2Vec2(35,-19)};
    chainShape2.CreateLoop(vertices2, 3);

    b2FixtureDef groundFixtureDef2;
    groundFixtureDef2.density = 0;
    groundFixtureDef2.shape = &chainShape2;

    b2BodyDef groundBodyDef2;
    groundBodyDef2.type = b2_staticBody;

    b2Body *groundBody2 = m_world->CreateBody(&groundBodyDef2);
    b2Fixture *groundBodyFixture2 = groundBody2->CreateFixture(&
        groundFixtureDef2);


    // Create a static body as the box for the spring
    b2BodyDef bd1;
    bd1.type = b2_staticBody;
    bd1.angularDamping = 0.1f;

    bd1.position.Set(1.0f, 0.5f);
    b2Body* springbox = m_world->CreateBody(&bd1);

    b2PolygonShape shape1;
    shape1.SetAsBox(0.5f,5.5f);
    springbox->CreateFixture(&shape1, 5.0f);


    // Create the left box connected to the spring as the movable
         object
    b2PolygonShape leftboxShape;
    leftboxShape.SetAsBox(0.5f, 0.5f);
```

```
b2FixtureDef leftboxFixtureDef;
leftboxFixtureDef.restitution = 0.75f;
leftboxFixtureDef.density = 0.1f; // this will affect the box
     mass
leftboxFixtureDef.friction = 0.1f;
leftboxFixtureDef.shape = &leftboxShape;

b2BodyDef leftboxBodyDef;
leftboxBodyDef.type = b2_dynamicBody;
leftboxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&leftboxBodyDef);
b2Fixture *leftboxFixture = m_box->CreateFixture(&
    leftboxFixtureDef);
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(0.5f, 0.5f); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 2.0f; // this will affect the box
    mass,
boxFixtureDef2.friction = 0.0f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(15.0f, -1.0f);

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    leftboxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
```

```
        m_length = jd.length;
        m_minLength = 2.0f;
        m_maxLength = 10.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
            m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        // Create the Pulley
        b2PulleyJointDef pulleyDef;
        b2Vec2 anchor1(5.5f, 0.5f); // the position of the end string
             of the left cord is (5.5,0.5) connecting to the left box
        b2Vec2 anchor2(15.0f, -1.0f); // the position of the end
            string of the right cord is (15.0f,-1) connecting to the
             right hanging box
        b2Vec2 groundAnchor1(11.5f, 0.5f ); // the string of the cord
             is tightened at (11.5, 0.5)
        b2Vec2 groundAnchor2(12.5f, 0.5f); // the string of the cord
            is tightened at (12.5, 0.5)
        // the last float is the ratio
        pulleyDef.Initialize(m_box, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_boxr;
    b2DistanceJoint* m_joint;
    b2PulleyJoint* m_joint1;
    bool m_fixed_camera;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
            //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
            m_box->ApplyForceToCenter(b2Vec2(-800.0f, 0.0f), true
```

```
                        );
                    break;
                    case GLFW_KEY_S:
                    //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f));
                    m_box->ApplyForceToCenter(b2Vec2(-500.0f, 0.0f), true
                        );
                    break;
                    case GLFW_KEY_D:
                    m_box->ApplyForceToCenter(b2Vec2(500.0f, 0.0f), true);
                    break;
                    case GLFW_KEY_F:
                    m_box->ApplyForceToCenter(b2Vec2(800.0f, 0.0f), true);
                    break;
                    case GLFW_KEY_G:
                    m_box->ApplyForceToCenter(b2Vec2(1200.0f, 0.0f), true)
                        ;
                    break;
                    case GLFW_KEY_C:
                    m_fixed_camera = !m_fixed_camera;
                    if(m_fixed_camera)
                    {
                            g_camera.m_center = b2Vec2(2.0f, -10.0f);
                            g_camera.m_zoom = 1.3f; // zoom out camera
                    }
                    break;

            }
    }
    void UpdateUI() override
    {
            ImGui::SetNextWindowPos(ImVec2(10.0f, 300.0f));
            ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
            ImGui::Begin("Joint Controls", nullptr,
                ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

            if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                f"))
            {
                    m_length = m_joint->SetLength(m_length);
            }

            if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                ))
            {
                    float stiffness;
                    float damping;
                    b2LinearStiffness(stiffness, damping, m_hertz,
                        m_dampingRatio, m_joint->GetBodyA(), m_joint->
```

```
                        GetBodyB());
                    m_joint->SetStiffness(stiffness);
                    m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                    float stiffness;
                    float damping;
                    b2LinearStiffness(stiffness, damping, m_hertz,
                        m_dampingRatio, m_joint->GetBodyA(), m_joint->
                        GetBodyB());
                    m_joint->SetStiffness(stiffness);
                    m_joint->SetDamping(damping);
            }

            ImGui::End();
    }
    void Step(Settings& settings) override
    {
            b2MassData massData = m_box->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2MassData massDatar = m_boxr->GetMassData();
            b2Vec2 positionr = m_boxr->GetPosition();
            b2Vec2 velocityr = m_boxr->GetLinearVelocity();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
                 60 Hertz

            g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
                apply different force to the box");
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                f", m_time);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Left Mass position =
                (%4.1f, %4.1f)", position.x, position.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Left Mass velocity =
                (%4.1f, %4.1f)", velocity.x, velocity.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Right Mass position =
                (%4.1f, %4.1f)", positionr.x, positionr.y);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Right Mass velocity =
                (%4.1f, %4.1f)", velocityr.x, velocityr.y);
            m_textLine += m_textIncrement;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Left Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Mass = %.6f",
                    massDatar.mass);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                g_camera.m_center.x = m_box->GetPosition().x;
                Test::Step(settings);
                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_box->GetPosition();
                       g_camera.m_zoom = 1.3f; // zoom out camera
                }
                float positionnew = (positionr.x + positionr.y);
                printf("%4.2f %4.2f %4.2f\n", positionr.x, positionr.y,
                    positionnew);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new SpringIncline;
        }

    };

    static int testIndex = RegisterTest("Potential Energy and Conservation of
        Energy", "Spring Incline", SpringIncline::Create);
```

**C++ Code 74:** *tests/potentialenergy_springincline.cpp "Spring and Breadbox on an Incline Box2D"*

Some explanations for the codes:

- Even if the example does not have mass after the spring, in Box2D to use **DistanceJointDef** we need to define two bodies first before connect it to the **PulleyJointDef**, thus we create a mass to connect the spring and the pulley on top of the ground.

```
            b2DistanceJointDef jd;
            jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
                leftboxBodyDef.position);
```

**Figure 10.10:** *The simulation of a 2 kg breadbox on a frictionless incline of angle $40^0$ connected to a pulley and then to a spring (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/potentialenergy_springincline.cpp****).*

## VII. Simulation for Conservation of Mechanical Energy: Marble Spring Gun with Box2D

Taken from problem number 36 chapter 8 of [6].

Two children are playing a game in which they try to hit a small box on the floor with a marble fired from a spring-loaded gun that is mounted on a table. The target box is horizontal distance $D = 2.2\ m$ from the edge of the table; Bobby compresses the spring 1.1 $cm$, but the center of the marble falls 27 $cm$ short of the center of the box. How far should Rhoda compress the spring to score a direct hit? Assume that neither the spring nor the ball encounters friction in the gun.

**Solution:**
The distance the marble travels is determined by its initial speed (and the methods of motion in two dimensions), and the initial speed is determined (using energy conservation) by the original compression of the spring. We denote $h$ as the height of the table, and $x$ as the horizontal distance to the point where the marble lands.

Then

$$x = v_0 t$$

$$h = \frac{1}{2} g t^2$$

since the vertical component of the marble's "launch velocity" is zero. From these we find

$$x = v_0\sqrt{\frac{2h}{g}}$$

We note from this that the distance to the landing point is directly proportional to the initial speed. We denote $v_{01}$ be the initial speed of the first shot and

$$D_1 = 2.2 - 0.27 = 1.93$$

1.93 $m$ is the horizontal distance to its landing point; similarly, $v_{02}$ is the initial speed of the second shot and $D = 2.2$ $m$ is the horizontal distance to its landing spot. Then

$$\begin{aligned}\frac{v_{02}}{v_{01}} &= \frac{D}{D_1}\\ v_{02} &= \frac{D}{D_1}v_{01}\end{aligned}$$

When the spring is compressed an amount $l$, the elastic potential energy is $\frac{1}{2}kl^2$. When the marble leaves the spring its kinetic energy is

$$\frac{1}{2}mv_0^2$$

Mechanical energy is conserved:

$$\frac{1}{2}mv_0^2 = \frac{1}{2}kl^2$$

and we see that the initial speed of the marble is directly proportional to the original compression of the spring.

If $l_1$ is the compression for the first shot and $l_2$ is the compression for the second, then

$$v_{02} = \left(\frac{l_2}{l_1}\right)v_{01}$$

Relating this to the previous result, we obtain

$$\begin{aligned}l_2 &= \frac{D}{D_1}l_1\\ &= \left(\frac{2.2}{1.93}\right)(1.1)\\ &= 1.25\end{aligned}$$

Thus, Rhoda has to compress the spring 1.25 $cm$ to score a direct hit.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>
#define DEGTORAD 0.0174532925199432957f

class MarbleSpringgun : public Test
{
```

```
public:
MarbleSpringgun()
{
        b2Body* ground = NULL;

        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(-46.0f, 0.5f), b2Vec2(12.0f,
                    0.5f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(12.0f, 0.0f), b2Vec2(12.0f
                    ,-20.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(12.0f, -20.0f), b2Vec2(46.0f
                    ,-20.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(37.0f, -19.0f), b2Vec2(37.0f
                    ,-20.0f));
                ground->CreateFixture(&shape, 0.0f);
        }
        {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;
                shape.SetTwoSided(b2Vec2(35.0f, -19.0f), b2Vec2(35.0f
                    ,-20.0f));
```

```
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        float L = 30.0f;
        float costheta = cosf(40*DEGTORAD);
        float sintheta = sinf(40*DEGTORAD);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(35-costheta*L,-19+sintheta*L
            ), b2Vec2(35,-19));
        ground->CreateFixture(&shape, 0.0f);
    }

    // Create the left triangle, theta = 40, L = 30
    float L = 30.0f;
    float costheta = cosf(40*DEGTORAD);
    float sintheta = sinf(40*DEGTORAD);
    b2ChainShape chainShape2;
    b2Vec2 vertices2[] = {b2Vec2(35-costheta*L,-19), b2Vec2(35-
        costheta*L,-19+sintheta*L), b2Vec2(35,-19)};
    chainShape2.CreateLoop(vertices2, 3);

    b2FixtureDef groundFixtureDef2;
    groundFixtureDef2.density = 0;
    groundFixtureDef2.shape = &chainShape2;

    b2BodyDef groundBodyDef2;
    groundBodyDef2.type = b2_staticBody;

    b2Body *groundBody2 = m_world->CreateBody(&groundBodyDef2);
    b2Fixture *groundBodyFixture2 = groundBody2->CreateFixture(&
        groundFixtureDef2);

    // Create a static body as the box for the spring
    b2BodyDef bd1;
    bd1.type = b2_staticBody;
    bd1.angularDamping = 0.1f;

    bd1.position.Set(1.0f, 1.0f);
    b2Body* springbox = m_world->CreateBody(&bd1);

    b2PolygonShape shape1;
    shape1.SetAsBox(0.5f,5.5f);
    springbox->CreateFixture(&shape1, 5.0f);
```

```
// Create the left box connected to the spring as the movable
     object
b2PolygonShape leftboxShape;
leftboxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef leftboxFixtureDef;
leftboxFixtureDef.restitution = 0.75f;
leftboxFixtureDef.density = 0.1f; // this will affect the box
     mass
leftboxFixtureDef.friction = 0.1f;
leftboxFixtureDef.shape = &leftboxShape;

b2BodyDef leftboxBodyDef;
leftboxBodyDef.type = b2_dynamicBody;
leftboxBodyDef.position.Set(7.0f, 1.0f);

m_box = m_world->CreateBody(&leftboxBodyDef);
leftboxFixture = m_box->CreateFixture(&leftboxFixtureDef);
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward

// Create the marble ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 0.5f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(15.0f, -1.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
ballFixture = m_ball->CreateFixture(&ballFixtureDef);

// Create the marble ball that will be spawned near the
    spring

ballShape2.m_p.SetZero();
ballShape2.m_radius = 0.5f;

ballFixtureDef2.restitution = 0.75f;
```

```
ballFixtureDef2.density = 0.5f; // this will affect the ball
    mass
ballFixtureDef2.friction = 0.1f;
ballFixtureDef2.shape = &ballShape2;

ballBodyDef2.type = b2_dynamicBody;
ballBodyDef2.position.Set(10.0f, 1.0f);

m_ball2 = m_world->CreateBody(&ballBodyDef2);

// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 1.0f),
    leftboxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 10.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(5.5f, 1.0f); // the position of the end string
     of the left cord is (5.5,1.0) connecting to the left box
b2Vec2 anchor2(15.0f, -1.0f); // the position of the end
    string of the right cord is (15.0f,-1) connecting to the
     right hanging box
b2Vec2 groundAnchor1(11.5f, 1.0f ); // the string of the cord
     is tightened at (11.5, 1.0)
b2Vec2 groundAnchor2(12.5f, 1.0f); // the string of the cord
    is tightened at (12.5, 1.0)
// the last float is the ratio
pulleyDef.Initialize(m_box, m_ball, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.0f);

m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

m_time = 0.0f;
```

```
        }
        b2Body* m_box;
        b2Body* m_ball;
        b2Body* m_ball2;

        b2CircleShape ballShape2;
        b2FixtureDef ballFixtureDef2;
        b2BodyDef ballBodyDef2;

        b2DistanceJoint* m_joint;
        b2Fixture* leftboxFixture;
        b2Fixture* ballFixture;
        b2Fixture* ballFixture2;
        b2PulleyJoint* m_joint1;
        bool m_fixed_camera;
        float m_length;
        float m_time;
        float m_minLength;
        float m_maxLength;
        float m_hertz;
        float m_dampingRatio;

        void Keyboard(int key) override
        {
               switch (key)
               {
                      case GLFW_KEY_A:
                      m_box->ApplyForceToCenter(b2Vec2(-150.0f, 0.0f), true
                          );
                      break;
                      case GLFW_KEY_S:
                      m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f), true
                          );
                      break;
                      case GLFW_KEY_D:
                      m_box->ApplyForceToCenter(b2Vec2(150.0f, 0.0f), true);
                      break;
                      case GLFW_KEY_F:
                      m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f), true);
                      break;
                      case GLFW_KEY_G:
                      m_box->ApplyForceToCenter(b2Vec2(500.0f, 0.0f), true);
                      break;
                      case GLFW_KEY_B:
                      m_ball2->CreateFixture(&ballFixtureDef2);
                      break;
                      case GLFW_KEY_L:
                      m_world->DestroyJoint(m_joint1); //Thanks beautiful..
```

```
                        break;
                        case GLFW_KEY_M:
                        m_ball->DestroyFixture(ballFixture); //Thanks
                            beautiful..
                        break;
                        case GLFW_KEY_C:
                        m_fixed_camera = !m_fixed_camera;
                        if(m_fixed_camera)
                        {
                               g_camera.m_center = b2Vec2(2.0f, -10.0f);
                               g_camera.m_zoom = 1.3f; // zoom out camera
                        }
                        break;
                }
        }
        void UpdateUI() override
        {
                ImGui::SetNextWindowPos(ImVec2(10.0f, 300.0f));
                ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
                ImGui::Begin("Joint Controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

                if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                    f"))
                {
                        m_length = m_joint->SetLength(m_length);
                }

                if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                    ))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                    , 2.0f, "%.1f"))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
```

```
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
        }

        ImGui::End();
    }
    void Step(Settings& settings) override
    {
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity = m_box->GetLinearVelocity();
        b2MassData massDatar = m_ball->GetMassData();
        b2Vec2 positionr = m_ball->GetPosition();
        b2Vec2 velocityr = m_ball->GetLinearVelocity();
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz

        g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
            apply different force to the box");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press L to
            disconnected the string between the spring and the ball")
            ;
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press M to delete the
            first ball");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press B to create a
            new ball near the spring");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Spring position =
            (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Spring velocity =
            (%4.1f, %4.1f)", velocity.x, velocity.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Marble position =
            (%4.1f, %4.1f)", positionr.x, positionr.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Marble velocity =
            (%4.1f, %4.1f)", velocityr.x, velocityr.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Spring Mass = %.6f",
            massData.mass);
        m_textLine += m_textIncrement;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Marble Mass = %.6f",
                    massDatar.mass);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                g_camera.m_center.x = m_box->GetPosition().x;
                Test::Step(settings);
                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_box->GetPosition();
                       g_camera.m_zoom = 1.3f; // zoom out camera
                }
                float positionnew = (positionr.x + positionr.y);
                printf("%4.2f %4.2f %4.2f\n", positionr.x, positionr.y,
                    positionnew);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new MarbleSpringgun;
        }

    };

    static int testIndex = RegisterTest("Potential Energy and Conservation of
        Energy", "Marble Spring Gun", MarbleSpringgun::Create);
```

**C++ Code 75:** *tests/potentialenergy_marblespringgun.cpp "Marble Spring Gun Box2D"*

We can adjust the compressed force to make the spring compressed for certain distances so the ball hit by the spring can get into the small box on the floor hole-in-one.

Some explanations for the codes:

- To spawn a new ball, delete the string connecting the ball and the spring, we have to do some tricky techniques in our Box2D coding.

  First, always remember it is useful to declare the variable globally, for **b2Body, b2Fixture, b2PulleyJoint** (so we can do the delete of the pulley joint string later on, or deleting the fixture that create the ball), then in order to spawn a new ball we also need to declare **b2CircleShape, b2FixtureDef, b2BodyDef** globally.

```
            b2Body* m_box;
            b2Body* m_ball;
            b2Body* m_ball2;

            b2CircleShape ballShape2;
            b2FixtureDef ballFixtureDef2;
```

```
b2BodyDef ballBodyDef2;

b2DistanceJoint* m_joint;
b2Fixture* leftboxFixture;
b2Fixture* ballFixture;
b2Fixture* ballFixture2;
b2PulleyJoint* m_joint1;
```

thus, at the keyboard press events you can delete fixtures that already created or even spawn a new fixture.

```
case GLFW_KEY_B:
m_ball2->CreateFixture(&ballFixtureDef2);
break;
case GLFW_KEY_L:
m_world->DestroyJoint(m_joint1);
break;
case GLFW_KEY_M:
m_ball->DestroyFixture(ballFixture);
break;
```

**Figure 10.11:** *The simulation of a marble being hit by a compressed spring then hit a small box on the floor (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/potentialenergy_marblespringgun.cpp****).*

## VIII. Reading a Potential Energy Curve

[DF*] Suppose that a particle is constrained to move along an $x$ axis while the conservative force does work on it. We want to plot the potential energy $U(x)$ that is associated with that force and the work that it does, and then we want to consider how we can relate the plot back to the force and to the kinetic energy of the particle.

[DF*] If we know the potential energy function $U(x)$ for a system in which a one-dimensional force $F(x)$ acts on a particle, we can find the force as

$$F(x) = -\frac{dU(x)}{dx} \tag{10.9}$$

[DF*] If $U(x)$ is given on a graph, then at any value of $x$, the force $F(x)$ is the engative of the slope of the curve there and the kinetic energy of the particle is given by

$$K(x) = E_{mec} - U(x) \tag{10.10}$$

where $E_{mec}$ is the mechanical energy of the system.

[DF*] A turning point is a point $x$ at which the particle reverses its motion ($K = 0$). A place where $K = 0$ (because $U = E$) and the particle changes direction.

[DF*] The particle is in equilibrium at points where the slope of the $U(x)$ curve is zero ($F(x) = 0$).

[DF*] **Example:**

The figure below shows a plot of potential energy $U$ versus position $x$ of a 0.9 $kg$ particle that can travel only along an $x$ axis. (Nonconservative forces are not involved)

Three values are $U_A = 15\ J$, $U_B = 35\ J$, and $U_C = 45\ J$. The particle is released at $x = 4.5\ m$ with an initial speed of 7 $m/s$, headed in the negative $x$ direction.

(a) If the particle can reach $x = 1\ m$, what is its speed there, and if it cannot, what is its turning point?

(b) What is the magnitude of the force on the particle as it begins to move to the left of $x = 4\ m$?

(c) What is the direction of the force on the particle as it begins to move to the left of $x = 4\ m$?

(d) Suppose, instead, the particle is headed in the positive $x$ direction when it is released at $x = 4.5\ m$ at speed 7 $m/s$. If the particle can reach $x = 7\ m$, what is its speed there, and if it cannot, what is its turning point?

**Figure 10.12:** *The plot of potential energy U versus position x of a* 0.9 *kg particle.*

(e) What is the magnitude of the force on the particle as it begins to move to the right of $x = 5\ m$?

(f) What is the direction of the force on the particle as it begins to move to the right of $x = 5\ m$?

**Solution:**

(a) From the figure, we see that at $x = 4.5\ m$, the potential energy is $U_1 = 15\ J$. If the speed is $v = 7\ m/s$, then the kinetic energy is

$$\begin{aligned} K_1 &= \frac{1}{2}mv^2 \\ &= \frac{1}{2}(0.90)(7) \\ &= 22 \end{aligned}$$

Thus

$$E_1 = U_1 + K_1 = 15 + 22 = 37$$

The total energy is 37 $J$.

Now, at $x = 1\ m$, the potential energy is $U_2 = 35\ J$. By energy conservation, we have

$$\begin{aligned} K_1 + U_1 &= K_2 + U_2 \\ 37 &= K_2 + 35 \\ K_2 &= 2 \end{aligned}$$

thus, $K_2 = 2\ J$. This means that the particle can reach there with a corresponding speed

$$\begin{aligned} v_2 &= \sqrt{\frac{2K_2}{m}} \\ &= \sqrt{\frac{2(2)}{0.9}} \\ &= 2.1 \end{aligned}$$

the speed there is 2.1 $m/s$.

(b) The force acting on the particle is related to the potential energy by the negative of the slope:

$$F_x = -\frac{\triangle U}{\triangle x}$$

From the figure we have

$$\begin{aligned} F_x &= -\frac{\triangle U}{\triangle x} \\ &= -\frac{35-15}{2-4} \\ &= 10 \end{aligned}$$

(c) Since the magnitude $F_x = 10\ J > 0$, the force points in the $+x$ direction.

(d) At $x = 7\ m$, the potential energy is $U_3 = 45\ J$, which exceeds the initial total energy $E_1$. Thus, the particle can never reach there. At the turning point, the kinetic energy is zero. Between $x = 5\ m$ and $x = 6\ m$, the potential energy is given by

$$U(x) = 15 + 30(x-5), \quad 5 \le x \le 6$$

Thus, the turning point is found by solving

$$\begin{aligned} U(x) &= U_1 + U_3 - U_1(x-5) \\ E_1 &= 15 + (45-15)(x-5) \\ 37 &= 15 + 30(x-5) \\ x &= 5.7333 \end{aligned}$$

the turning point is at $x = 5.7\ m$, since the particle can never reach $x = 7\ m$.

(e) At $x = 5\ m$, the force acting on the particle is

$$\begin{aligned} F_x &= -\frac{\triangle U}{\triangle x} \\ &= -\frac{(45-15)}{(6-5)} \\ &= -30 \end{aligned}$$

The magnitude is $|F_x| = 30\ N$.

(f) The fact that $F_x < 0$ indicated that the force points in the $-x$ direction.

[DF*] **Example:**
The potential energy of a diatomic molecule (a two-atom system like $H_2$ or $O_2$) is given by

$$U = \frac{A}{r^{12}} - \frac{B}{r^6}$$

where $r$ is the separation of the two atoms of the molecule and $A$ and $B$ are positive constants. This potential energy is associated with the force that binds the two atoms together.

(a) Find the equilibrium separation, the distance between the atoms at which the force on each atom is zero.

(b) Is the force repulsive (the atoms are pushed apart) or attractive (they are pulled together) if their separation is smaller?

(c) Is the force repulsive (the atoms are pushed apart) or attractive (they are pulled together) if their separation is larger than the equilibrium separation?

**Solution:**

(a) The force at the equilibrium position $r = r_{eq}$ is

$$\begin{aligned} F &= 0 \\ -\frac{dU}{dr}|_{r=r_{eq}} &= 0 \\ -\frac{12A}{r_{eq}^{13}} + \frac{6B}{r_{eq}^{7}} &= 0 \end{aligned}$$

which lead to the result

$$r_{eq} = \left(\frac{2A}{B}\right)^{1/6} = 1.12\left(\frac{A}{B}\right)^{1/6}$$

(b) This defines a minimum in the potential energy curve (can be verified either by a graph or by taking another derivative and verifying that it concave upward at this point), which means that for values of $r$ slightly smaller than $r_{eq}$ the slope of the curve is negative (so the force is positive, repulsive).

(c) And for values of $r$ slightly larger than $r_{eq}$ the slope of the curve must be positive (so the force is negative, attractive).

[DF*] A single conservative force $F(x)$ acts on a 1 $kg$ particle that moves along an $x$ axis. The potential energy $U(x)$ associated with $F(x)$ is given by

$$U(x) = -4x\, e^{-x/4}$$

where $x$ is in meters. At $x = 5\ m$ the particle has a kinetic energy of 2 $J$.

(a) What is the mechanical energy of the system?

(b) Make a plot of $U(x)$ as a function of $x$ for $0 \leq x \leq 10$, and on the same graph draw the line that represents the mechanical energy of the system.

(c) Use part (b) to determine the least value of $x$ the particle can reach.

(d) Use part (b) to determine the greatest value of $x$ the particle can reach.

(e) Use part (b) to determine the maximum kinetic energy of the particle.

(f) Use part (b) to determine the value of $x$ which (e) occurs.

(g) Determine an expression in newtons and meters for $F(x)$ as a function of $x$.

(h) For what (finite) value of $x$ does $F(x) = 0$?

**Solution:**

(a) The potential energy at $x = 5\ m$ is

$$U(5) = -4(5)\, e^{-(5)/4} = -5.7$$

Thus

$$E_{mec} = K(x) + U(5) = 2 - 5.7 = -3.7$$

the mechanical energy at $x = 5\ m$ is $-3.7\ J$.

**Figure 10.13:** *The plot of $U(x)$ for $0 \leq x \leq 10$ and the energy $E_{mec}$ (the horizontal line) (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch10-Potential Energy/Problem41-2D Plot U(x)/main.cpp**).*

(b)

(c) The problem asks for a graphical determination of the turning points, which are the points on the curve corresponding to the total energy computed in part (a). The result for the smallest turning point (determined, by more careful means) is $x = 1.3\ m$.

(d) The result for the largest turning point is $x = 9.1\ m$.

(e) Since $K(x) = E_{mec} - U(x)$, then maximizing $K(x)$ involves finding the minimum of $U(x)$. A graphical determination suggests that this occurs at $x = 4.0\ m$, which plugs into the expression

$$\begin{aligned} K(x) &= E_{mec} - U(x) \\ &= -3.7 - (-4xe^{-x/4}) \\ &= -3.7 - (-4(4)e^{-(4)/4}) \\ &= -3.7 + 5.886 \\ K(x) &= 2.18 \end{aligned}$$

The maximum kinetic energy of the particle is $2.18\ J$. Alternatively, one can measure from the graph from the minimum of the $U$ curve up to the level representing the total energy $E$ and thereby obtain an estimate of $K$ at that point.

(f) As mentioned in the previous part, the minimum of the $U$ curve occurs at $x = 4\ m$.

(g) The force follows from the potential energy

$$\begin{aligned} F &= \frac{dU}{dx} \\ &= (4 - x)e^{-x/4} \end{aligned}$$

(h) This revisits the considerations of parts (d) and (e), since we are returning to the minimum of $U(x)$ but now with the advantage of having the analytic result of part (g). We see that the location that produces $F = 0$ is exactly $x = 4\ m$.

## IX. Work Done On A System By An External Force

[DF*] Work $W$ is energy transferred to or from a system by means of an external force acting on the system.

Positive work $+W$ done on an arbitrary system means a transfer of energy to the system. Negative work $-W$ means a transfer of energy from the system.

[DF*] When more than one force acts on a system, their net work is the transferred energy.

[DF*] When friction is not involved, the work done on the system and the change $\triangle E_{mec}$ in the mechanical energy of the system are equal:

$$W = \triangle E_{mec} = \triangle K + \triangle U \tag{10.11}$$

[DF*] When a constant horizontal force $\vec{F}$ pulls a block along an $x$ axis and through a displacement of magnitude $d$, a kinetic frictional force $\vec{f_k}$ from the floor acts on the block and opposes the motion. The block has velocity $\vec{v}_0$ at the start of a displacement $\vec{d}$ and velocity $\vec{v}$ at the end of the displacement.

Positive work $+W$ is done on the block-floor system by force $\vec{F}$, resulting in a change $\triangle E_{mec}$ in the block's mechanical energy and a change $\triangle E_{th}$ in the thermal energy of the block and floor.

Now, we can write the law for components along the $x$ axis as

$$\begin{aligned} F_{net,x} &= ma_x \\ F - f_k &= ma \end{aligned}$$

Because the forces are constant, the acceleration $\vec{a}$ is also constant. Thus,

$$v^2 = v_0^2 + 2ad$$

Solving this equation for $a$, then substituting into the equation $F - f_k = ma$, we will get

$$\begin{aligned} \frac{F - f_k}{m} &= \frac{v^2 - v_0^2}{2d} \\ 2d\frac{F - f_k}{m} &= v^2 - v_0^2 \\ Fd - f_k d &= \frac{1}{2}mv^2 - \frac{1}{2}mv_0^2 \\ Fd &= \frac{1}{2}mv^2 - \frac{1}{2}mv_0^2 + f_k d \\ Fd &= \triangle K + f_k d \end{aligned}$$

because $\frac{1}{2}mv^2 - \frac{1}{2}mv_0^2 = \triangle K$ for the block.

In a more general situation, in which the block is moving up a ramp, there can be a change in potential energy as well. To include such possible change, we generalize the formula above by writing

$$Fd = \triangle E_{mec} + f_k d \tag{10.12}$$

$Fd$ is the work $W$ done by the external force $\vec{F}$ (the energy transferred by the force).

[DF*] When a kinetic frictional force acts within the system, then the thermal energy $E_{th}$ of the system changes. (This energy is associated with the random motion of atoms and molecules in the system). The work done on the system is then

$$W = \triangle E_{mec} + \triangle E_{th} \tag{10.13}$$

[DF*] The change $\triangle E_{th}$ is related to the magnitude $f_k$ of the frictional force and the magnitude $d$ of the displacement caused by the external force by

$$\triangle E_{th} = f_k d \tag{10.14}$$

By experiment we find that the block pulled by a force and the portion of the floor along which it slides will become warmer as the block slides on the floor.

[DF*] **Example:**
A rottweiler drags its bed box across a floor by applying a horizontal force of 8 $N$. The kinetic frictional force acting on the box has magnitude of 5 $N$. As the box is dragged through 0.7 $m$ along the way what are

(a) The work done by the rottweiler's applied force ?

(b) The increase in thermal energy of the bed and floor?

**Solution:**

(a)

$$W_{applied} = Fd = 8(0.7) = 5.6$$

the work done by the rottweiler's applied force is 5.6 $J$.

(b)

$$\triangle E_{th} = f_k d = 5(0.7) = 3.5$$

the thermal energy generated is 3.5 $J$.

[DF*] **Example:**
A rope is used to pull a 3.57 $kg$ block at constant speed 4.06 $m$ along a horizontal floor. The force on the block from the rope is 7.68 $N$ and directed $15^0$ above the horizontal. What are

(a) The work done by the rope's force?

(b) The increase in thermal energy of the block-floor system?

(c) The coefficient of kinetic friction between the block and floor?

**Solution:**

(a)

$$\begin{aligned} W &= Fd \ \cos \ \theta \\ &= (7.68)(4.06) \cos \ 15^0 \\ &= 30.1 \end{aligned}$$

the work done on the block by the force in the rope is 30.1 $J$.

(b) Using $f_k = F \ \cos \ 15^0 = 7.42$ for the magnitude of the kinetic friction force,

$$\begin{aligned} \triangle E_{th} &= f_k d \\ &= (7.42)(4.06) \\ &= 30.1 \end{aligned}$$

the increase in thermal energy is 30.1 $J$.

[DF*] We can use Newton's second law of motion to obtain the frictional and normal forces, then use

$$\mu_k = \frac{f}{F_N}$$

to obtain the coefficient of friction. Place the $x$ axis along the path of the block and the $y$ axis normal to the floor. The $x$ and the $y$ component of Newton's second law are

**Figure 10.14:** *The free-body diagram for the system where a rope is used to pull a* 3.57 *kg block at constant speed*4.06 *m along a horizontal floor.*

$$x : F\ \cos\ \theta - f_k = 0$$
$$y : F_N + F\ \sin\ \theta - mg = 0$$

where $m$ is the mass of the block, $F$ is the force exerted by the rope, and $\theta$ is the angle between the force and the horizontal.

The first equation gives

$$\begin{aligned} f &= F\ \cos\ \theta \\ &= (7.68)\cos\ 15^0 \\ &= 7.42 \end{aligned}$$

the second equation gives

$$\begin{aligned} F_N &= mg - F\ \sin\ \theta \\ &= (3.57)(9.8) - (7.68)\sin\ 15^0 \\ &= 33 \end{aligned}$$

thus,

$$\mu_k = \frac{f}{F_N} = \frac{7.42}{33} = 0.225$$

the coefficient of kinetic friction between the floor and the block is 0.225.

[DF*] **Example:**

A collie drags its bed boz across a floor by applying a horizontal force of 8.0 $N$. The kinetic frictional force acting on the box has magnitude 5.0 $N$. As the box is dragged through 0.70 $m$ along the way, what are

(a) the work done by the collie's applied force?

(b) the increase in thermal energy of the bed and floor?

**Solution:**

(a) We will have

$$W_{applied} = (8.0)(0.7) = 5.6J$$

(b) The thermal energy generated is

$$\triangle E_{th} = f_k d = (5.0)(0.7) = 3.5J$$

## X. Conservation of Energy

[DF*] The total energy $E$ of a system (the sum of its mechanical energy and its internal energies, including thermal energy) can change only by amounts of energy that are transferred to or from the system. This experimental fact is known as the law of conservation of energy.

Energy simply can not magically appear or disappear.

[DF*] If work $W$ is done on the system, then

$$W = \triangle E = \triangle E_{mec} + \triangle E_{th} + \triangle E_{int} \tag{10.15}$$

If the system is isolated ($W = 0$), this gives

$$\triangle E_{mec} + \triangle E_{th} + \triangle E_{int} = 0 \tag{10.16}$$

and

$$E_{mec,2} = E_{mec,1} - \triangle E_{th} - \triangle E_{int} \tag{10.17}$$

where the subscripts 1 and 2 refer to two different instants.

[DF*] An engine increases the speed of a car with four-wheel drive (all four wheels are made to turn by the engine). During the acceleration, the engine causes the tires to push backward on the road surface. This push produces frictional forces $\vec{f}$ that act on each tire in the forward direction.

The net external force $\vec{F}$ from the road, which is the sum of these frictional forces, accelerates the car, increasing its kinetic energy. However, $\vec{F}$ does not transfer energy from the road to the car and so does no work on the car. Rather, the car's kinetic energy increases as a result of internal trasnfers from the energy stored in the fuel.

Another example, an initially stationary ice-skater pushes away from a railing and then slides over the ice. Her kinetic energy increases because of an external force $\vec{F}$ on her from the rail. However, that force does not transfer energy from the rail to her. Thus, the force does no work on her. Rather, her kinetic energy increases as a result of internal transfers from the biochemical energy in her muscles.

We can sometimes relate the external force $\vec{F}$ on an object to the change in the object's mechanical energy if we can simplify the situation. During the ice skater' push through distance $d$ we can simplify by assuming that the acceleration is constant, her speed changing

from $v_0 = 0$ to $v$. We assume $\vec{F}$ has constant magnitude $F$ and angle $\phi$. After the push, we can simplify the skater as being a particle and neglect the fact that the exertions of her muscles have increased the thermal energy in her muscles and changed other physiological features. Then we can apply

$$\begin{aligned} \frac{1}{2}mv^2 - \frac{1}{2}mv_0^2 &= F_x d \\ K - K_0 &= (F\ \cos\ \phi)d \\ \triangle K &= Fd\ \cos\ \phi \end{aligned} \tag{10.18}$$

or, if the situation also involves a change in the elevation of an object, we can include the change $\triangle U$ in gravitational potential energy by writing

$$\triangle U + \triangle K = Fd\ \cos\ \phi \tag{10.19}$$

The force on the right side of this equation does no work on the object but is still responsible for the changes in energy shown on the left side.

[DF*] The power due to a force is the rate at which that force transfers energy. If an amount of energy $\triangle E$ is transferred by a force in an amount of time $\triangle t$, the average power of the force is

$$P_{avg} = \frac{\triangle E}{\triangle t} \tag{10.20}$$

[DF*] The instantaneous power due to a force is

$$P = \frac{dE}{dt} \tag{10.21}$$

On a graph of energy $E$ versus time $t$, the power is the slope of the plot at any given time.

[DF*] **Example:**

A glider is shot by a spring along a water-drenched (frictionless) track that takes the glider from a horizontal section down to ground level. As the glider then moves along ground-level track, it is gradually brough to rest by friction. The total mass of the glider and its rider is $m = 200\ kg$, the initial compression of the spring is $d = 5\ m$, the spring constant $k = 3.2 \times 10^3\ N/m$, the initial height is $h = 35\ m$, and the coefficient of kinetic friction along the ground-level track is $\mu_k = 0.8$. Through what distance $L$ does the glider slide along the ground-level track until it stops?

**Solution:**

We need to examine all the forces and then determine what our system would be. Do we have an isolated system (our equation would be for the conservation of energy) or a system on which an external force does work (our equation would relate that work to the system's change in energy)?

Forces: The normal force on the glider from the track does no work on the glider because the direction of this force is always perpendicular to the direction of the glider's displacement. The gravitational force does work on the glider, and because the force is conservative we can associate a potential energy with it. As the spring pushes on the glider to get it moving, a spring force does work on it, transferring energy from the elastic potential energy of the compressed spring to kinetic energy of the glider. The spring force also pushes against a

rigid wall. Because there is friction between the glider and the ground-level track, the sliding of the glider along the track section increase their thermal energies.

System: Let's take the system to contain all the interacting bodies: glider, track, spring, Earth, and wall. Then, because all the force interactions are within the system, the system is isolated and thus its total energy cannot change. So, the equation we should use is not that of some external force doing work on the system. Rather, it is a conservation of energy.

$$E_{mec,2} = E_{mec,1} - \triangle E_{th}$$

Let subscript 1 correspond to the initial state of the glider (when it is still on the compressed spring) and the subscript 2 correspond to the final state of the glider (when it has come to rest on the ground-level). For both states, the mechanical energy of the system is the sum of any potential energy and any kinetic energy.

We have two types of potential energy: the elastic potential energy

$$U_e = \frac{1}{2}kx^2$$

associated with the compressed spring and the gravitational potential energy

$$U_g = mgy$$

associated with the glider's elevation. For the latter, let's take the ground level as the reference level. That means that the glider is initially at height $y = h$ and finally at height $y = 0$.
In the initial state, with the glider stationary and elevated and the spring compressed, the energy is

$$\begin{aligned} E_{mec,1} &= K_1 + U_{e1} + U_{g1} \\ &= 0 + \frac{1}{2}kd^2 + mgh \end{aligned}$$

In the final state, with the spring now in its relaxed state and the glider again stationary but no longer elevated, the final mechanical energy of the system is

$$\begin{aligned} E_{mec,2} &= K_2 + U_{e2} + U_{g2} \\ &= 0 + 0 + 0 \end{aligned}$$

Let's next go after the change $\triangle E_{th}$ of the thermal energy of the glider and the ground-level track. We can substitute for $\triangle E_{th}$ with $f_k L$ (the product of the frictional force magnitude and the distance of rubbing). We know that

$$f_k = \mu_k F_N$$

where $F_N$ is the normal force. Because the glider moves horizontally through the region with friction, the magnitude of $F_N$ is equal to $mg$. So, the friction from the mechanical energy amounts to

$$\triangle E_{th} = \mu_k mgL$$

Thus

$$\begin{aligned} E_{mec,2} &= E_{mec,1} - \triangle E_{th} \\ 0 &= \frac{1}{2}kd^2 + mgh - \mu_k mgL \\ L &= \frac{kd^2}{2\mu_k mg} + \frac{h}{\mu_k} \\ &= \frac{(3200)(5^2)}{2(0.8)(200)(9.8)} + \frac{35}{0.8} \\ &= 69.3 \end{aligned}$$

The distance $L$ till the glide stop at the ground-level track is 69.3 $m$. When we use the law of the conservation of energy, we canr ealte the initial and final states of the system with no consideration of the intermediate states. We did not need to consider the glider as it slides over the uneven track. If we had, instead, applied Newton's second law to the motion, we would have had to know the details of the track and would have faced a far more difficult calculation.

[DF*] **Example:**
A 75 $g$ Frisbee is thrown from a point 1.1 $m$ above the ground with a speed of 12 $m/s$. When it has reached a height of 2.1 $m$, its speed is 10.5 $m/s$. What was the reduction in $E_{mec}$ of the Frisbee-Earth system because of air drag?
**Solution:**
We use SI units so

$$m = 0.075\ kg$$

we know that $\triangle E_{th} = -\triangle E_{mec}$ for the energy "lost" in the sense of this problem. Thus,

$$\begin{aligned} \triangle E_{th} &= \frac{1}{2}m(v_i^2 - v_f^2) + mg(y_i - y_f) \\ &= \frac{1}{2}(0.075)(12^2 - 10.5^2) + (0.075)(9.8)(1.1 - 2.1) \\ &= 0.53\ J \end{aligned}$$

[DF*] **Example:**
During a rockslide, a 520 $kg$ rock slides from rest down a hillside that is 500 $m$ long and 300 $m$ high. The coefficient of kinetic friction between the rock and the hill surface is 0.25.

(a) If the gravitational potential energy $U$ of the rock-Earth system is zero at the bottom of the hill, what is the value of $U$ just before the slide?

(b) How much energy is transferred to thermal energy during the slide?

(c) What is the kinetic energy of the rock as it reaches the bottom of the hill?

(d) What is its speed then?

**Solution:**

(a) The initial potential energy is

$$U_i = mgy_i = (520)(9.8)(300) = 1.53 \times 10^6\ J$$

where $+y$ is upward and $y = 0$ at the bottom (so that $U_f = 0$).

(b) Since

$$f_k = \mu_k F_N = \mu_k mg \ \cos \ \theta$$

we have

$$\triangle E_{th} = f_k d = \mu_k mgd \ \cos \ \theta$$

Now, the hillside surface (of length $d = 500 \ m$) is treated as an hypotenuse of a $3-4-5$ triangle, so

$$\cos \ \theta = \frac{x}{d}$$

where $x = 400 \ m$. Therefore,

$$\begin{aligned}\triangle E_{th} &= \mu_k mgd\frac{x}{d} \\ &= \mu_k mgx \\ &= (0.25)(520)(9.8)(400) \\ &= 5.1 \times 10^5 \ J\end{aligned}$$

(c) We will have

$$\begin{aligned}K_f &= K_i + U_i - U_f - \triangle E_{th} \\ &= 0 + (1.53 \times 10^6) - 0 - (5.1 \times 10^5) \\ &= 1.02 \times 10^6 \ J\end{aligned}$$

(d) From $K_f = mv^2/2$, we obtain $v = 63 \ m/s$

[DF*] **Example:**

At a certain factory, 300 $kg$ crates are dropped vertically from a packing machine onto a conveyor belt moving at 1.20 $m/s$. (A motor maintains the belt's conveyor constant speed). The coefficient of kinetic friction between the belt and each crate is 0.400. After a short time, slipping between the belt and the crate ceases, and the crate then moves along with the belt. For the period of time during which the crate is being brought to rest relative to the belt, calculate, for a coordinate system at rest in the factory

(a) the kinetic energy supplied to the crate.

(b) the magnitude of the kinetic frictional force acting on the crate

(c) the energy supplied by the motor

(d) Explain why answers (a) and (c) differ.

**Solution:**

(a) Since the speed of the crate of mass $m$ increases from 0 to 1.20 $m/s$ relative to the factory ground, the kinetic energy supplied to it is

$$K = \frac{1}{2}mv^2 = \frac{1}{2}(300)(120)^2 = 216 \ J$$

(b) The magnitude of the kinetic frictional force is

$$f = \mu F_N = \mu mg = (0.400)(300)(9.8) = 1.18 \times 10^3 \ N$$

(c) Let the distance the crate moved relative to the conveyor belt before it stops slipping be $d$. Then from

$$v^2 = 2ad = 2\left(\frac{f}{m}\right)d$$

we find

$$\triangle E_{th} = fd = \frac{1}{2}mv^2 = K$$

Thus, the total energy that must be supplied by the motor is

$$W = K + \triangle E_{th} = 2K = (2)(216) = 432\ J$$

(d) The energy supplied by the motor is the work $W$ it does on the system, and must be greater than the kinetic energy gained by the crate computed in part (b). This is due to the fact that part of the energy supplied by the motor is being used to compensate for the energy dissipated $\triangle E_{th}$ while it was slipping.

[DF*] **Example:**
Each second, 1200 $m^3$ of water passes over a waterfall 100 $m$ high. Three-fourths of the kinetic energy gained by the water in falling is transferred to electrical energy by a hydroelectric generator. At what rate does the generator produce electrical energy? (The mass of 1 $m^3$ of water is 1000 $kg$).
**Solution:**
This problem deals with the concept of hydroelectric generator - kinetic energy of water can be converted into electrical energy.

By energy conservation, the change in kinetic energy of water in one second is

$$\begin{aligned}\triangle K &= -\triangle U \\ &= mgh \\ &= \rho V g h \\ &= (10^3)(1200)(9.8)(100) \\ &= 1.176 \times 10^9\ J\end{aligned}$$

only $\frac{3}{4}$ of this amount is transferred to electrical energy.

The power generation (assumed constant, so average power is the same as instantaneous power) is

$$P_{avg} = \frac{\left(\frac{3}{4}\right)\triangle K}{t} = \frac{\left(\frac{3}{4}\right)(1.176 \times 10^9)}{1} = 8.82 \times 10^8\ W$$

Hydroelectricity is the most widely used renewable energy, it accounts for almost 20% of the world's electricity supply.

[DF*] **Example:**
A massless rigid rod of length $L$ has a ball fo mass $m$ attached to one end. The other end is pivoted in such a way that the ball will move in a vertical circle. First, assume that there is no friction at the pivot. The system is launched downward from the horizontal position $A$ with initial speed $v_0$. The ball just barely reaches point $D$ and then stops.

(a) Derive an expression for $v_0$ in terms of $L, m$, and $g$.

(b) What is the tension in the rod when the ball passes through $B$?

**Figure 10.15:** *The illustration for a massless rigid rod of length L that has a ball of mass m attached to one end.*

(c) A little grit is placed on the pivot to increase the friction there. Then the ball just barely reaches $C$ when launched from $A$ with same speed as before. What is the decrease in the mechanical energy during this motion?

(d) What is the decrease in the mechanical energy by the time the ball finally comes to rest at $B$ after several oscillations?

**Solution:**
We have a ball attached to a rod that moves in a vertical circle. The total mechanical energy of the system is conserved.

Let position $A$ be the reference point for potential energy, $U_A = 0$. The total mechanical energies at $A, B$, and $C$ are:

$$\begin{aligned} E_A &= \frac{1}{2}mv_A^2 + U_A = \frac{1}{2}mv_0^2 \\ E_B &= \frac{1}{2}mv_B^2 + U_B = \frac{1}{2}mv_B^2 - mgL \\ E_D &= \frac{1}{2}mv_D^2 + U_D = mgL \end{aligned}$$

where $v_D = 0$. The problem can be analyzed by applying energy conservation:

$$E_A = E_B = E_D$$

(a) The condition $E_A = E_D$ gives

$$\begin{aligned} \frac{1}{2}mv_0^2 &= mgL \\ v_0 &= \sqrt{2gL} \end{aligned}$$

(b) To find the tension in the rod when the ball passes through $B$, we first calculate the speed at $B$. Using $E_B = E_D$, we find

$$\frac{1}{2}mv_B^2 - mgL = mgL$$

or $v_B = \sqrt{4gL}$. The direction of the centripetal acceleration is upward (at that moment),

as is the tension force. Thus, Newton's second law gives

$$\begin{aligned} T - mg &= \frac{mv_B^2}{r} \\ &= \frac{m(4gL)}{L} \\ T - mg &= 4mg \\ T &= 5mg \end{aligned}$$

(c) The difference in height between $C$ and $D$ is $L$, so the "loss" of mechanical energy (which goes into thermal energy) is $-mgL$.

(d) The difference in height between $B$ and $D$ is $2L$, so the total "loss" of mechanical energy (which all goes into thermal energy) is $-2mgL$.

An alternative way to calculate the energy loss is to note that

$$E'_B = \frac{1}{2}mv'^2_B + U_B = 0 - mgL = -mgL$$

which gives

$$\triangle E = E'_B - E_A = -mgL - mgL = -2mgL$$

[DF*] **Example:**

A 1.50 $kg$ water balloon is shot straight up with an initial speed of 3.00 $m/s$.

(a) What is the kinetic energy of the balloon just as it is launched?

(b) How much work does the gravitational force do on the balloon during the balloon's full ascent?

(c) What is the change in the gravitational potential energy of the balloon-Earth system during the full ascent?

(d) If the gravitatitonal potential energy is taken to be zero at the launch point, what is its value when the balloon reaches its maximum height?

(e) If, instead, the gravitational potential energy is taken to be zero at the maximum height, what is its value at the launch point?

(f) What is the maximum height?

**Solution:**

(a) The kinetic energy is

$$K_i = \frac{1}{2}mv_i^2 = \frac{1}{2}(1.5)(3^2) = 6.75\ J$$

(b) The work of gravity is the negative of its change in potential energy. At the highest point, all of $K_i$ has converted into $U$ (if we neglect air friction) so we conclude the work of gravity is $-6.75\ J$.

(c) And we conclude that

$$\triangle U = 6.75\ J$$

(d) The potential energy there is

$$U_f = U_i + \triangle U = 6.75\ J$$

(e) If $U_f = 0$, then

$$U_i = U_f - \triangle U = -6.75\ J$$

(f) Since $mg\triangle y = \triangle U$, we obtain

$$\triangle y = 0.459\ m$$

[DF*] **Example:**
The luxury liner Queen Elizabeth 2 has a diesel-electric power plant with a maximum power of 92 $MW$ at a cruising speed of 32.5 $knots$. What forward force is exerted on the ship at this speed? (1 $knot = 1.852\ km/h$). **Solution:**
We use $P = Fv$ to compute the force:

$$\begin{aligned} F &= \frac{P}{v} \\ &= \frac{92 \times 10^6}{(32.5)(1.852)\left(\frac{1000}{3600}\right)} \\ &= 5.5 \times 10^6\ N \end{aligned}$$

[DF*] **Example:**
The summit of Mount Everest is 8850 $m$ above sea level.

(a) How much energy would a 90 $kg$ climber expend against the gravitational force on him in climbint to the summit from sea level?

(b) How many candy bars at 1.25 $MJ$ per bar, would supply an energy equivalent to this?

**Solution:**

(a) The (internal) energy the climber must convert to gravitational potential energy is

$$\triangle U = mgh = (90)(9.80)(8850) = 7.8 \times 10^6\ J$$

(b) The number of candy bars corresponds to is

$$N = \frac{7.8 \times 10^6}{1.25 \times 10^6} = 6.2\ bars$$

# XI. Simulation for Conservation of Energy: a Block Slides to a Higher Level with Box2D

Taken from problem number 57 chapter 8 of [6].

A block slides along a track from one level to a higher level after passing through an intermediate valley. The track is frictionless until the block reaches the higher level. There is a frictional force stops the block in a distance $d$. The block's initial speed $v_0$ is 6.0 $m/s$, the height difference $h$ is 1.1 $m$, and $\mu_k = 0.60$. Find $d$.

**Solution:**

Since the valley is frictionless, the only reason for the speed being less when it reaches the higher level is the gain in potential energy $\triangle U = mgh$ where $h = 1.1\ m$. Sliding along the rough surface of the higher level, the block finally stops since its remaining kinetic energy has turned to thermal energy

$$\triangle E_{th} = f_k d = \mu_k mgd$$

where $\mu_k = 0.60$. Thus, with $W = 0$ provides us with an equation to solve for the distance $d$:

$$K_i = \triangle U + \triangle E_{th} = mgh + \mu_k d$$

where $K_i = \frac{1}{2}mv_i^2$ and $v_i = 6.0\ m/s$. Dividing by mass and rearranging, we obtain

$$d = \frac{v_i^2}{2\mu_k g} - \frac{h}{\mu_k} = 1.2\ m$$

```
#include "test.h"

class BlockSlidestoHigherLevel: public Test
{
	public:
	BlockSlidestoHigherLevel()
	{
		// Create the first horizontal track
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;

			b2FixtureDef fd;
			fd.shape = &shape;
			fd.density = 0.0f;
			fd.friction = 0.0f;

			shape.SetTwoSided(b2Vec2(-20.0f, -76.0f), b2Vec2(27.0
				f, -76.0f));
			ground->CreateFixture(&fd);
```

```
}
//
{
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 0.0f;
        fd.friction = 0.0f;

        // hs0[11] will define the parameter t for each of the
             vertices that create the circle
        // Use parametric equation to get circle shape of
            radius 24
        // (cos t, sin t)
        float hs0[7] = {1.3f*b2_pi, 1.4f*b2_pi, 1.5f*b2_pi,
            1.6f*b2_pi, 1.7f*b2_pi, 1.8f*b2_pi, 1.9f*b2_pi};
        float yc0 = 1.26f*b2_pi;
        for (int32 i = 0; i < 7; ++i)
        {
                float y0 = hs0[i];
                shape.SetTwoSided(b2Vec2(44+24*cos(yc0),
                    -59+24*sin(yc0)), b2Vec2(44+24*cos(yc0 +
                    0.1f*b2_pi), -59+24*sin(yc0 + 0.1f*b2_pi))
                    );
                ground->CreateFixture(&fd);
                yc0 = y0;
        }

}
// Create the second higher horizontal track
b2Body* ground2= NULL;
{
        b2BodyDef bd;
        ground2 = m_world->CreateBody(&bd);

        b2EdgeShape shape;

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 0.0f;
        fd.friction = 0.6f; // the friction for the higher
            level track
```

```
                        shape.SetTwoSided(b2Vec2(67.0f, -66.0f), b2Vec2(108.0f
                            , -66.0f));
                        ground2->CreateFixture(&fd);

                }
                // Boxes
                {
                        b2PolygonShape boxShape1;
                        boxShape1.SetAsBox(0.5f, 0.5f);

                        b2FixtureDef boxFixtureDef1;
                        boxFixtureDef1.restitution = 0.0f;
                        boxFixtureDef1.density = 0.5f/1.0f; // this will
                            affect the mass
                        boxFixtureDef1.friction = 0.01f; // if we set this to
                            0, then the box will keep moving, so we set this
                            at a very small decimal number
                        boxFixtureDef1.shape = &boxShape1;

                        b2BodyDef boxBodyDef1;
                        boxBodyDef1.type = b2_dynamicBody;

                        boxBodyDef1.position.Set(17.5f, -75.5f);
                        m_box = m_world->CreateBody(&boxBodyDef1);
                        b2Fixture *boxFixture1 = m_box->CreateFixture(&
                            boxFixtureDef1);

                }

                m_time = 0.0f;
        }
        bool m_fixed_camera;
        float m_time;
        b2Body* m_box;

        void Keyboard(int key) override
        {
                switch (key)
                {
                        case GLFW_KEY_D:
                        m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
                        break;
                        case GLFW_KEY_F:
                        m_box->SetLinearVelocity(b2Vec2(33.0f, 0.0f));
                        break;
                        case GLFW_KEY_C:
                        m_fixed_camera = !m_fixed_camera;
                        if(m_fixed_camera)
```

```
                {
                        g_camera.m_center = b2Vec2(2.0f, -10.0f);
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
        }
}

void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f;
        b2MassData massData2 = m_box->GetMassData();
        b2Vec2 position2 = m_box->GetPosition();
        b2Vec2 velocity2 = m_box->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
            /tracking");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Keys: push the box to
            the right with d= linear velocity 30 m/s");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Keys: push the box to
            the right with f = linear velocity 33 m/s");
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Box Mass = %4.2f",
            massData2.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
             %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
             %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;

        Test::Step(settings);
        if(!m_fixed_camera)
        {
                g_camera.m_center = m_box->GetPosition();
                g_camera.m_zoom = 3.3f; // zoom out camera
        }
}

static Test* Create()
```

```
            {
                    return new BlockSlidestoHigherLevel;
            }


        };

        static int testIndex = RegisterTest("Potential Energy and Conservation of
            Energy", "Block Slides to Higher Level", BlockSlidestoHigherLevel::
            Create);
```

**C++ Code 76:** *tests/conservationofenergy_blockslidestohigherlevel.cpp "Block Slides to Higher Level Box2D"*

**Figure 10.16:** *The simulation of block slides along a track from one level to a higher level through the valley as the connector (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/conservationofenergy_blockslidestohigherlevel.cpp****).*

## XII. Simulation for Conservation of Energy: Elevator Snaps with Box2D

Taken from problem number 63 chapter 8 of [6].

The cable of the 1800 $kg$ elevator cab snaps, when the cab is at rest at the first floor, where the cab bottom is at distance $d = 3.7\ m$ above a spring of spring constant $k = 0.15MN/m$. A safety device clamps the cab against guide rails so that a constant frictional force of 4.4 $kN$ opposes the cab's motion.

(a) Find the speed of the cab just before it hits the spring.

(b) Find the maximum distance $x$ that the spring is compressed (the frictional force still acts during this compression).

(c) Find the distance that the cab will bounce back up the shaft.

(d) Using conservation of energy, find the approximate total distance that the cab will move before coming to rest. (Assume that the frictional force on the cab is negligible when the cab is stationary.)

**Solution:**
We observe that the last line of the problem indicates that static friction is not to be considered a factor in this problem. The friction force of magnitude $f = 4400\ N$ mentioned in the problem is kinetic friction and is constant (and directed upward), and the thermal energy change associated with it is

$$\triangle E_{th} = fd$$

where $d = 3.7\ m$ in part (a) (but will be replaced by $x$, the spring compression, in part $(b)$).

(a) With $W = 0$ and the reference level for computing $U = mgy$ set at the top of the (relaxed) spring, leads to

$$\begin{aligned} U_i &= K + \triangle E_{th} \\ mgy_i &= \frac{1}{2}mv^2 + fd \\ v^2 &= 2d\left(g - \frac{f}{m}\right) \\ v &= \sqrt{2d\left(g - \frac{f}{m}\right)} \\ v &= 7.4 \end{aligned}$$

(b) We again utilize $W = 0$, now relating its kinetic energy at the moment it makes contact with the spring to the system energy at the bottom-most point. Using the same reference level for computing $U = mgy$ as we did in part (a), we end up with gravitational potential energy equal to $mg(-x)$ at that bottom-most point, where the spring (with spring constant $k = 1.5 \times 10^5 N/m$) is fully compressed.

$$K = mg - x + \frac{1}{2}kx^2 + fx$$

where $K = \frac{1}{2}mv^2 = 4.9 \times 10^4\ J$ using the speed found in part (a). Using the abbreviation

$$\xi = mg - f = 1.3 \times 10^4\ N$$

the quadratic formula yields

$$x = \frac{\xi \pm \sqrt{\xi^2 + 2kK}}{k} = 0.90$$

where we have taken the positive root.

(c) We relate the energy at the bottom-most point to that of the highest point of rebound (a distance $d'$ above the relaxed position of the spring). We assume $d' > x$. We now use the bottom-most point as the reference level for computing gravitational potential energy.

$$\frac{1}{2}kx^2 + mgd' + fd'$$

$$d' = \frac{kx^2}{2(mg+f)}$$

$$d' = 2.8$$

(d) The non-conservative force is friction, and the energy term associated with it is the one that keeps track of the total distance traveled (whereas the potential energy terms, coming as they do from conservative forces, depend on positions - but not on the paths that led to them). We assume the elevator comes to final rest at the equilibrium position of the spring, with the spring compressed an amount $d_{eq}$ given by

$$mg = kd_{eq}$$

$$d_{eq} = \frac{mg}{k}$$

$$d_{eq} = 0.12$$

In this part, we use that final-rest point as the reference level for computing gravitational potential energy, so the original $U = mgy$ becomes $mg(d_{eq} + d)$. In that final position, then, the gravitational energy is zero and the spring energy is $\frac{kd_{eq}^2}{2}$. Thus,

$$mg(d_{eq} + d) = \frac{1}{2}kd_{eq}^2 + fd_{total}$$

$$(1800)(9.8)(0.12 + 3.7) = \frac{1}{2}(1.5 \times 10^5)(0.12) + 4400d_{total}$$

$$d_{total} = 15$$

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class ElevatorSnaps : public Test
{
        public:
        ElevatorSnaps()
        {
                b2Body* ground = NULL;
```

```
m_world->SetGravity(b2Vec2(0.0f,-9.8f));
// Create the pendulum set
b2Body* b1;
{
       b2EdgeShape shape;
       shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
           0.0f));

       b2BodyDef bd;
       b1 = m_world->CreateBody(&bd);
       b1->CreateFixture(&shape, 0.0f);
}
b2Body* b2; // the hanging bar for the pendulum
{
       b2PolygonShape shape;
       shape.SetAsBox(7.25f, 0.25f);

       b2BodyDef bd;
       bd.position.Set(0.0f, 37.0f);
       b2 = m_world->CreateBody(&bd);
       b2->CreateFixture(&shape, 0.0f);
}

b2RevoluteJointDef jdelevator;
b2Vec2 anchor;

// Create the elevator
b2PolygonShape boxShape1;
boxShape1.SetAsBox(1.0f, 2.0f);

b2FixtureDef boxFixtureDef1;
boxFixtureDef1.restitution = 0.75f;
boxFixtureDef1.density = 225.0f; // this will affect the left
     elevator mass, the multiplier is (1800.0f/225.0f)
boxFixtureDef1.friction = 0.3f;
boxFixtureDef1.shape = &boxShape1;

b2BodyDef boxBodyDef1;
boxBodyDef1.type = b2_dynamicBody;
boxBodyDef1.position.Set(0.0f, 12.9f); // set the elevator
    position here
// boxBodyDef2.fixedRotation = true;
// boxBodyDef1.angularDamping = 0.2f;

m_boxelevator = m_world->CreateBody(&boxBodyDef1);
// Create the elevator
b2Fixture *boxFixture1 = m_boxelevator->CreateFixture(&
    boxFixtureDef1);
```

```
// Create the anchor and connect it to the crate
anchor.Set(0.0f, 36.0f); // x and y axis position for the
    Pendulum anchor
jdelevator.Initialize(b2, m_boxelevator, anchor);
// to create the joint / rope that connect the elevator,
// we use (b2RevoluteJoint*)m_world->CreateJoint(&jdelevator
    ); so we can destroy it later
m_jointelevator = (b2RevoluteJoint*)m_world->CreateJoint(&
    jdelevator);
// End of creating the pendulum set


{
       b2BodyDef bd;
       ground = m_world->CreateBody(&bd);

       b2EdgeShape shape;
       shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
           0.0f));
       ground->CreateFixture(&shape, 0.0f);
}
// Create the wall surrounding the spring
b2BodyDef bd2;
bd2.type = b2_staticBody;
bd2.angularDamping = 0.1f;

bd2.position.Set(-1.7f, 18.5f);
b2Body* leftwall = m_world->CreateBody(&bd2);

b2PolygonShape shape2;
shape2.SetAsBox(0.5f, 18.5f);
leftwall->CreateFixture(&shape2, 5.0f);

b2BodyDef bd3;
bd3.type = b2_staticBody;
bd3.angularDamping = 0.1f;

bd3.position.Set(1.65f, 18.5f);
b2Body* rightwall = m_world->CreateBody(&bd3);

b2PolygonShape shape3;
shape3.SetAsBox(0.5f, 18.5f);
rightwall->CreateFixture(&shape3, 5.0f);

// Create a static body as the box for the spring
b2BodyDef bd1;
bd1.type = b2_staticBody;
```

```
bd1.angularDamping = 0.1f;

bd1.position.Set(0.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape;
boxShape.SetAsBox(1.0f, 0.2f); // the dimension of the spring
     upper part

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.0f;
boxFixtureDef.density = 1.5f; // this will affect the spring
    mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(0.0f, 8.0f); // the spring end will
    be located in (0,8.0)

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
    ;
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(0.0f, 0.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
    m_minLength
m_maxLength = 8.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);
```

```
        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_boxelevator;
    b2DistanceJoint* m_joint;
    b2RevoluteJoint* m_jointelevator;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_W:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, 1000.0f), true)
                ;
            break;
            case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, -800.0f), true
                );
            break;
            case GLFW_KEY_L:
            m_world->DestroyJoint(m_jointelevator); //Thanks
                beautiful..
            break;
            case GLFW_KEY_T:
            m_time = 0.0f;
            break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 260.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Spring Joint Controls", nullptr,
            ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
            f"))
        {
```

```
                m_length = m_joint->SetLength(m_length);
            }

            if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                ))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            ImGui::End();
        }
        void Step(Settings& settings) override
        {
            b2MassData massData = m_box->GetMassData();
            b2MassData massDataelevator= m_boxelevator->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2Vec2 position_elevator = m_boxelevator->GetPosition();
            b2Vec2 velocity_elevator = m_boxelevator->GetLinearVelocity
                ();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
                 60 Hertz
            float m = massDataelevator.mass;
            float d = 3.7f;
            float g = 9.8f;
            float k = 150000.0f;
            float fk = 4400.0f;
            float y_eq = 7.8f;
            float v;
            v = sqrt(2*d*(g-(fk/m)));
```

```
g_debugDraw.DrawString(5, m_textLine, "Press L to make the
    elevator snaps");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press W to apply force
     to the spring 1,000 N upward / S to apply force 1,000 N
    downward");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
    );
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Elevator Mass = %4.2f"
    , massDataelevator.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring position =
    (%4.1f, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring velocity =
    (%4.1f, %4.1f)", velocity.x, velocity.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring Mass = %4.2f",
    massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
     = %4.1f", k);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Elevator position =
    (%4.1f, %4.1f)", position_elevator.x, position_elevator.y
    );
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Elevator velocity =
    (%4.1f, %4.1f)", velocity_elevator.x, velocity_elevator.y
    );
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The distance between
    the elevator and the spring, d = %4.1f", d);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The frictional force
    when the elevator falls, f = %4.1f", fk);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
    for the spring, y = %4.1f", y_eq);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The speed of the
    elevator before it hits the spring, v = %4.1f", v);
```

```
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f\n", velocity.y, position.y,
                    position.y);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new ElevatorSnaps;
        }

};

static int testIndex = RegisterTest("Potential Energy and Conservation of
    Energy", "Elevator Snaps", ElevatorSnaps::Create);
```

**C++ Code 77:** *tests/conservationofenergy_elevatorsnaps.cpp "Elevator Snaps Box2D"*

Some explanations for the codes:

- The trick to make the elevator snaps here is to destroy the **RevoluteJoint** that is acting as the rope and connector between the anchor and the elevator.

  We need to define **m_jointelevator** below, along with other variables. So that we can use this command **m_world->DestroyJoint(m_jointelevator);** at the **void Keyboard ()** for setting key commands with **GLFW**.

```
            b2RevoluteJointDef jdelevator;
            b2Vec2 anchor;

            ...
            ...

            // Create the anchor and connect it to the crate
            anchor.Set(0.0f, 36.0f); // x and y axis position for the
                Pendulum anchor
            jdelevator.Initialize(b2, m_boxelevator, anchor);
            m_jointelevator = (b2RevoluteJoint*)m_world—>CreateJoint(&
                jdelevator);

            ...
            ...

            b2RevoluteJoint* m_jointelevator;
```

**Figure 10.17:** *The simulation of an elevator that snaps and hit the spring at the bottom (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/conservationofenergy_elevatorsnaps.cpp****).*

## XIII. Simulation for Conservation of Energy: Particle Slides till Stop with Box2D

Taken from problem number 65 chapter 8 of [6].

A particle can slide along a track with elevated ends and a flat central part (like a cross section of a bowl). The flat part has length $L = 40\ cm$. The curved portions of the track are frictionless, but for the flat part the coefficient of kinetic friction is $\mu_k = 0.20$. The particle is released from rest at point $A$, which is at height $h = L/2$. How far from the left edge of the flat part does the particle finally stop?

**Solution:**
The initial and final kinetic energies are zero, and we set up energy conservation in the form of

$$W = \triangle E_{mec} + \triangle E_{th}$$

with $W = 0$ according to our assumptions.

Certainly, it can only come to a permanent stop somewhere in the flat part, but the question is whether this occurs during its first pass through (going rightward) or its second pass through (going leftward) or its third pass through (going rightward again), and so on.

If it occurs during its first pass through, then the thermal energy generated is

$$\triangle E_{th} = f_k d$$

where $d \leq L$ and $f_k = \mu_k mg$.

If it occu rs during its second pass through, then the total thermal energy is

$$\triangle E_{th} = \mu_k mg(L + d)$$

where we again use the symbol $d$ for how far through the level area it goes during that last pass (so $0 \leq d \leq L$).

Generalizing to the $n^{th}$ pass through, we see that

$$\triangle E_{th} = \mu_k mg[(n-1)L + d]$$

In this way, we have

$$mgh = \mu_k mg[(n-1)L + d]$$

which simplifies (when $h = L/2$ is inserted) to

$$\frac{d}{L} = 1 + \frac{1}{2\mu_k} - n$$

By substituting $\mu_k = 0.2$ we will have

$$1 + \frac{1}{2\mu_k} = 1 + \frac{1}{2(0.2)} = 3.5$$

and to fulfill the requirement $0 \leq \frac{d}{L} \leq 1$, we need to have $n = 3$.

$$\frac{d}{L} = 1 + \frac{1}{2\mu_k} - n = 1 + 2.5 - 3 = 0.5$$

So, we arrive at the conclusion that

$$\frac{d}{L} = \frac{1}{2}$$

or

$$d = \frac{1}{2}L = \frac{1}{2}(40) = 20$$

and that this occurs on its third pass through the flat region, it is going rightward for the second time.

```
#include "test.h"

class ParticleSlidesTillStop: public Test
{
        public:
        ParticleSlidesTillStop()
        {
                b2Body* ground = NULL;
                // Create the circular paths on the left and the right side
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);
```

```
                b2EdgeShape shape;

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.density = 0.0f;
                fd.friction = 0.0f;

                // hs0[11] will define the parameter t for each of the
                     vertices that create the circle
                // Use parametric equation to get circle shape of
                    radius 24
                // (cos t, sin t)
                float hs0[5] = {1.1f*b2_pi, 1.2f*b2_pi, 1.3f*b2_pi,
                    1.4f*b2_pi, 1.5f*b2_pi};
                float hs1[4] = {1.6f*b2_pi, 1.7f*b2_pi, 1.8f*b2_pi,
                    1.83f*b2_pi};
                float yc0 = 1.1f*b2_pi;
                float yc1 = 1.83f*b2_pi;
                for (int32 i = 0; i < 5; ++i)
                {
                        float y0 = hs0[i];
                        //shape.SetTwoSided(b2Vec2(23+26*cos(yc0),
                            -43+40*sin(yc0)), b2Vec2(23+26*cos(yc0 +
                            0.1f*b2_pi), -43+40*sin(yc0 + 0.1f*b2_pi))
                            );
                        shape.SetTwoSided(b2Vec2(5+24*cos(yc0), -59+24*
                            sin(yc0)), b2Vec2(5+24*cos(yc0 + 0.1f*b2_pi
                            ), -59+24*sin(yc0 + 0.1f*b2_pi)));
                        ground->CreateFixture(&fd);
                        yc0 = y0;
                }
                for (int32 i = 0; i < 4; ++i)
                {
                        float y1 = hs1[i];
                        shape.SetTwoSided(b2Vec2(43.5+24*cos(yc1),
                            -60.17+24*sin(yc1)), b2Vec2(43.5+24*cos(
                            yc1 + 0.1f*b2_pi), -60.17+24*sin(yc1 + 0.1
                            f*b2_pi)));
                        ground->CreateFixture(&fd);
                        yc1 = y1;
                }

        }
        // Create the flat central part
        b2Body* ground2= NULL;
        {
                b2BodyDef bd;
```

```
        ground2 = m_world->CreateBody(&bd);

        b2EdgeShape shape;

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 0.0f;
        fd.friction = 0.2f; // the friction

        shape.SetTwoSided(b2Vec2(5.0f, -83.0f), b2Vec2(51.0f,
            -83.0f));
        ground2->CreateFixture(&fd);

    }
    /*
    {
        // Create the particle / a ball
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 3.3f; // this will affect the
             ball mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(-17.0f, -65.5f);

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);
        m_ball->SetAngularVelocity(1.0f);
    }
    */
    // Create the particle / a box
    {
        b2PolygonShape boxShape1;
        boxShape1.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef1;
        boxFixtureDef1.restitution = 0.0f;
        boxFixtureDef1.density = 1.5f/1.0f; // this will
            affect the mass
        boxFixtureDef1.friction = 0.02f; // if we set this to
```

```
                        0, then the box will keep moving, so we set this
                        at a very small decimal number
                boxFixtureDef1.shape = &boxShape1;

                b2BodyDef boxBodyDef1;
                boxBodyDef1.type = b2_dynamicBody;

                boxBodyDef1.position.Set(-16.5f, -65.5f);
                m_box = m_world->CreateBody(&boxBodyDef1);
                b2Fixture *boxFixture1 = m_box->CreateFixture(&
                    boxFixtureDef1);

        }

        m_time = 0.0f;
    }

    bool m_fixed_camera;
    float m_time;
    b2Body* m_box;
    b2Body* m_ball;

    void Keyboard(int key) override
    {
        switch (key)
        {
                case GLFW_KEY_D:
                m_box->SetLinearVelocity(b2Vec2(3.0f, 0.0f));
                break;
                case GLFW_KEY_F:
                m_box->SetLinearVelocity(b2Vec2(6.0f, 0.0f));
                break;
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                        g_camera.m_center = b2Vec2(2.0f, -10.0f);
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
        }
    }

    void Step(Settings& settings) override
    {
        m_time += 1.0f / 60.0f;
        float k = 0.2;
```

```
                b2MassData massData2 = m_box->GetMassData();
                b2Vec2 position2 = m_box->GetPosition();
                b2Vec2 velocity2 = m_box->GetLinearVelocity();

                g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
                    /tracking");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Keys: push the box to
                    the right with d= linear velocity 3 m/s");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Keys: push the box to
                    the right with f = linear velocity 6 m/s");
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The flat part
                    coefficient of kinetic friction, k = %4.1f", k);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %4.2f",
                    massData2.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
                     %4.1f)", position2.x, position2.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
                     %4.1f)", velocity2.x, velocity2.y);
                m_textLine += m_textIncrement;

                Test::Step(settings);
                if(!m_fixed_camera)
                {
                        g_camera.m_center = m_box->GetPosition();
                        g_camera.m_zoom = 2.3f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new ParticleSlidesTillStop;
        }

};

static int testIndex = RegisterTest("Potential Energy and Conservation of
    Energy", "Particle Slides Till Stop", ParticleSlidesTillStop::Create);
```

**C++ Code 78:** *tests/conservationofenergy_particleslidestillstop.cpp "Particle Slides till Stop Box2D"*

**Figure 10.18:** *The simulation of a particle that slides along a track with elevated end on the left and right side and a flat central part (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/conservationofenergy_particleslidestillstop.cpp****).*

# Chapter 11

# DFSimulatorC++ V: Center of Mass and Linear Momentum

*"Yes bla, thank you bla.."* - ***1 week old Browni' puppies when I bring them to Puncak Bintang then put them in small gazebo to be breastfeeded by Browni***

The complicated motion of a system of objects, suc as a car or a ballerina, can be simplified if we determine a special point of the system - the center of mass of that system.

## I. Center of Mass

[DF*] If you toss a ball into the air without much spin on the ball, its motion is simple-it follows a parabolic path, and the ball can be treated as a particle.

If, instead, you flip a baseball bat into the air, its motion is more complicated. Because every part of the bat moves differently, along paths of many different shapes, you cannot represent the bat as a particle. Instead, it is a system of particles each of which follows itw own path through the air. However, the bat has one special point-the center of mass-that does move in a simple parabolic path. The other parts of the bat move around the center of mass.

[DF*] The center of mass of a system of particles is the point that moves as though

1. all of the system's mass were concentrated there
2. all external forces were applied there.

we define the center of mass (com) of a system of particles in order to predict the possible motion of the system.

[DF*] Suppose we have two particles of masses $m_1$ and $m_2$ separated by distance $d$. Both particles are located on $x$ axis, and the first particle is located at $x_1$ from the origin of the $x$ axis, while the second particle is located at the $x_2$ from the origin of the $x$ axis, with $x_2 - x_1 = d$, so the position of the center of mass is now defined as

$$x_{com} = \frac{m_1 x_1 + m_2 x_2}{m_1 + m_2} \tag{11.1}$$

We can rewrite the equation above as

$$x_{com} = \frac{m_1 x_1 + m_2 x_2}{M} \tag{11.2}$$

with $M$ as the total mass of the system ($M = m_1 + m_2$).

If we have many particles, then we can extend the equation above to a more general situation in which $n$ particles are strung out along the $x$ axis. Thus, the location of the center of mass is

$$\begin{aligned} x_{com} &= \frac{m_1 x_1 + m_2 x_2 + \cdots + m_n x_n}{M} \\ &= \frac{1}{M} \sum_{i=1}^{n} m_i x_i \end{aligned} \tag{11.3}$$

[DF*] **Three Dimensions**

If the particles are distributed in three dimensions, the center of mass must be identified by three coordinates

$$\begin{aligned} x_{com} &= \frac{1}{M} \sum_{i=1}^{n} m_i x_i \\ y_{com} &= \frac{1}{M} \sum_{i=1}^{n} m_i y_i \\ z_{com} &= \frac{1}{M} \sum_{i=1}^{n} m_i z_i \end{aligned} \tag{11.4}$$

Recall that the position of a particle at coordinates $x_i, y_i$, and $z_i$ is given by a position vector (it points from the origin to the particle):

$$\vec{r}_i = x_i \hat{i} + y_i \hat{j} + z_i \hat{k} \tag{11.5}$$

We can also define the center of mass with the language of vectors

$$\vec{r}_{com} = x_{com} \hat{i} + y_{com} \hat{j} + z_{com} \hat{k} \tag{11.6}$$

with $\hat{i}, \hat{j}$, and $\hat{k}$ are unit vectors pointing, respectively, in the positive direction of the $x, y$, and $z$ axes.

This is the conscise equation

$$\vec{r}_{com} = \frac{1}{M} \sum_{i=1}^{n} m_i \vec{r}_i \tag{11.7}$$

[DF*] **Solid Bodies**

An ordinary object, such as a baseball bat, contains so many particles (atoms) that we can best treat it as a continuous distribution of matter. The "particles" then become differential mass elements $dm$, the sums become integrals, and the coordinates of the center of mass are defined as

$$\begin{aligned} x_{com} &= \frac{1}{M} \int x \, dm \\ y_{com} &= \frac{1}{M} \int y \, dm \\ z_{com} &= \frac{1}{M} \int z \, dm \end{aligned} \tag{11.8}$$

where $M$ is now the mass of the object. We only consider uniform objects, that have uniform density, or mass per unit volume.

The density, $\rho$, is the same for any given element of an object as for the whole object

$$\rho = \frac{dm}{dV} = \frac{M}{V} \tag{11.9}$$

where $dV$ is the volume occupied by a mass element $dm$, and $V$ is the total volume of the object. Substituting $dm \left(\frac{M}{V}\right) dV$ gives

$$\begin{aligned} x_{com} &= \frac{1}{V}\int x\, dV \\ y_{com} &= \frac{1}{V}\int y\, dV \\ z_{com} &= \frac{1}{V}\int z\, dV \end{aligned} \tag{11.10}$$

[DF*] The center of mass of a uniform sphere (which has a point of symmetry) is at the center of the sphere (which is the point of symmetry). The center of mass of a uniform cone (whose axis is a line of symmetry) lies on the axis of the cone. The center of mass of a banana (which has a plane of symmetry that splits it into two equal parts) lies somewhere in the plane of symmetry.

The center of mass of an object need not lie within the object. There is no dough at the com of a doughnut, and no iron at the com of a horseshoe.

[DF*] **Example:**
In the ammonia molecule, three hydrogen ($H$) atoms form an equilateral triangle, with the center of the triangle at distance $d = 9.40 \times 10^{-11}\ m$ from each hydrogen atom. The nitrogen ($N$) atom is at the apex of a pyramid, with the three hydrogen atoms forming the base. The nitrogen-to-hydrogen atomic mass ratio is 13.9, and the nitrogen-to-hydrogen distance is $L = 10.14 \times 10^{-11}\ m$. What are the $x$ and $y$ coordinates of the molecule's center of mass?
**Solution:**

**Figure 11.1:** *The ammonia ($NH_3$) molecule.*

By symmetry the center of mass is located on the axis of symmetry of the molecule - the $y$

axis. Therefore

$$x_{com} = 0$$

To find $y_{com}$, we note that $3m_H y_{com} = m_N(y_N - y_{com})$, where $y_N$ is the distance from the nitrogen atom to the plane containing the three hydrogen atoms:

$$y_N = \sqrt{(10.14 \times 10^{-11})^2 - (9.4 \times 10^{-11})^2} = 3.803 \times 10^{-11}\ m$$

We know that $\frac{m_N}{m_H} = 13.9$. Thus,

$$\begin{aligned} y_{com} &= \frac{m_N y_N}{m_N + 3m_H} \\ &= \frac{(13.9m_H)(3.803 \times 10^{-11})}{(13.9m_H) + 3(m_H)} \\ &= 3.1279\ m \end{aligned}$$

with the ratio we do not need to know the masses of each Hydrogen and Nitrogen atoms, but if we want to be curios and seek the periodic table, when we input $m_N = 14.0067$ and $m_H = 1.000797$ we will still obtain the same result.

## II. Simulation for Center of Mass for Three Particles with Box2D

Compute the center of mass for three particles of masses $m_1 = 2.59\ kg, m_2 = 5.73\ kg, m_3 = 1.8\ kg$, you can set the position for each particle as you like.

**Solution:**

We only need these two formulas for this problem:

$$x_{com} = \frac{1}{M}\sum_{i=1}^{3} m_i x_i$$

$$y_{com} = \frac{1}{M}\sum_{i=1}^{3} m_i y_i$$

The simulation with **Box2D** is very helpful as it can keep computing the updated computation for $x_{com}$ and $y_{com}$ while we can also rearrange the position of any of the particle.

**Reminder**:
You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose, e.g. **Center of Mass/Three Particles**.

```
#include "test.h"
#include <iostream>

class CenterofMassThreeParticles: public Test
{
        public:

        CenterofMassThreeParticles()
        {
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                b2Timer timer;
                // Perimeter Ground body
                {
                        b2BodyDef bd;
                        b2Body* ground = m_world->CreateBody(&bd);

                        b2EdgeShape shapeGround;
                        shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                            (30.0f, 0.0f));
```

```
        ground->CreateFixture(&shapeGround, 0.0f);

        b2EdgeShape shapeTop;
        shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
            (30.0f, 30.0f));
        ground->CreateFixture(&shapeTop, 0.0f);

        b2EdgeShape shapeLeft;
        shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
            (-40.0f, 30.0f));
        ground->CreateFixture(&shapeLeft, 0.0f);

        b2EdgeShape shapeRight;
        shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
            (30.0f, 30.0f));
        ground->CreateFixture(&shapeRight, 0.0f);
    }

    // Create the first particle
    b2CircleShape ballShape;
    ballShape.m_p.SetZero();
    ballShape.m_radius = 0.5f;

    b2FixtureDef ballFixtureDef;
    ballFixtureDef.restitution = 0.75f; // the bounciness
    ballFixtureDef.density = 3.3f; // this will affect the ball
        mass
    ballFixtureDef.friction = 0.1f;
    ballFixtureDef.shape = &ballShape;

    b2BodyDef ballBodyDef;
    ballBodyDef.type = b2_dynamicBody;
    ballBodyDef.position.Set(-10.0f, 25.0f);
    // ballBodyDef.angularDamping = 0.2f;

    m_ball = m_world->CreateBody(&ballBodyDef);
    b2Fixture *ballFixture = m_ball->CreateFixture(&
        ballFixtureDef);

    // Create the second particle
    b2CircleShape ballShape2;
    ballShape2.m_p.SetZero();
    ballShape2.m_radius = 0.5f;

    b2FixtureDef ballFixtureDef2;
    ballFixtureDef2.restitution = 0.75f; // the bounciness
    ballFixtureDef2.density = 7.3f; // this will affect the ball
        mass
```

```
        ballFixtureDef2.friction = 0.1f;
        ballFixtureDef2.shape = &ballShape2;

        b2BodyDef ballBodyDef2;
        ballBodyDef2.type = b2_dynamicBody;
        ballBodyDef2.position.Set(0.0f, 25.0f);
        // ballBodyDef2.angularDamping = 0.2f;

        m_ball2 = m_world->CreateBody(&ballBodyDef2);
        b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
            ballFixtureDef2);

        // Create the third particle
        b2CircleShape ballShape3;
        ballShape3.m_p.SetZero();
        ballShape3.m_radius = 1.5f;

        b2FixtureDef ballFixtureDef3;
        ballFixtureDef3.restitution = 0.45f; // the bounciness
        ballFixtureDef3.density = 2.3f; // this will affect the ball
            mass
        ballFixtureDef3.friction = 0.1f;
        ballFixtureDef3.shape = &ballShape2;

        b2BodyDef ballBodyDef3;
        ballBodyDef3.type = b2_dynamicBody;
        ballBodyDef3.position.Set(20.0f, 25.0f);
        // ballBodyDef3.angularDamping = 0.2f;

        m_ball3 = m_world->CreateBody(&ballBodyDef3);
        b2Fixture *ballFixture3 = m_ball3->CreateFixture(&
            ballFixtureDef3);

        m_createTime = timer.GetMilliseconds();
    }
    b2Body* bodylbox;
    b2Body* m_ball;
    b2Body* m_ball2;
    b2Body* m_ball3;

    void Step(Settings& settings) override
    {
        b2MassData massData = m_ball->GetMassData();
        b2Vec2 position = m_ball->GetPosition();
        b2MassData massData2 = m_ball2->GetMassData();
        b2Vec2 position2 = m_ball2->GetPosition();
        b2MassData massData3 = m_ball3->GetMassData();
        b2Vec2 position3 = m_ball3->GetPosition();
```

```
float xcom = (1/(massData.mass+massData2.mass+massData3.mass)
    ) * (massData.mass*position.x + massData2.mass*position2.
    x + massData3.mass*position3.x);
float ycom = (1/(massData.mass+massData2.mass+massData3.mass)
    ) * (massData.mass*position.y + massData2.mass*position2.
    y + massData3.mass*position3.y);

g_debugDraw.DrawString(5, m_textLine, "First Particle Mass =
    %.6f", massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "First Particle
    Position, x = %.6f", position.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "First Particle
    Position, y = %.6f", position.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Second Particle Mass =
     %.6f", massData2.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Second Particle
    Position, x = %.6f", position2.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Second Particle
    Position, y = %.6f", position2.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Third Particle Mass =
    %.6f", massData3.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Third Particle
    Position, x = %.6f", position3.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Third Particle
    Position, y = %.6f", position3.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "x_{com} = %.6f", xcom)
    ;
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "y_{com} = %.6f", ycom)
    ;
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
    ", m_createTime);
m_textLine += m_textIncrement;
```

```
                printf("%4.2f %4.2f \n", xcom, ycom);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new CenterofMassThreeParticles;

        }

        float m_createTime;
};

static int testIndex = RegisterTest("Center of Mass", "Three Particles",
    CenterofMassThreeParticles::Create);
```

**C++ Code 79:** *tests/centerofmass_threeparticles.cpp "Center of Mass Three Particles Box2D"*

**Figure 11.2:** *The simulation of three particles with different mass and positions to compute their $x_{com}$ and $y_{com}$ (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_threeparticles.cpp****).*

## III. Newton's Second Law for a System of Particles

[DF*] If you roll a cue ball at a second billiard ball that is at rest, you expect that the two-ball system will continue to have some forward motion after impact. What continues to move forward is the center of mass of the two-ball system (its steady motion completely unaffected by the collision). The center of mass continues to move forward, as if the collision had never occurred.

[DF*] Imagine if we have $n$ particles of different masses. We are interested not in the individual motions of these particles but only in the motion of the center of mass of the system. The vector equation that governs the motion of the center of mass of such a system of particles is

$$\vec{F}_{net} = M\vec{a}_{com} \tag{11.11}$$

This equation is Newton's second law for the motion of the center of mass of a system of particles.

1. $\vec{F}_{net}$ is the net force of all external forces that act on the system. Forces on one part of the system from another part of the system (internal forces) are not included.
2. $M$ is the total mass of the system. We assume that no mass enters or leaves the system as it moves, so that $M$ remains constant. The system is said to be closed.
3. $\vec{a}_{com}$ is the acceleration of the center of mass of the system.

These equations involve the components of $\vec{F}_{net}$ and $\vec{a}_{com}$ along the three coordinate axes

$$\begin{aligned} F_{net,x} &= Ma_{com,x} \\ F_{net,y} &= Ma_{com,y} \\ F_{net,z} &= Ma_{com,z} \end{aligned} \tag{11.12}$$

Now we can examine the behavior of the billiard balls. When a cue ball has begun to roll, no net external force acts on the system. Thus, because $\vec{F}_{net} = 0$, it tells us that $\vec{a}_{com} = 0$ also. Because acceleration is the rate of change of velocity, we conclude that the velocity of the center of mass of the system of two balls does not change. When the two balls collide, the forces that come into play are internal forces, on one ball from the other. Such forces do not contribute to the net force $\vec{F}_{net}$, which remains zero. Thus, the center of mass of the system, which was moving forward before the collision, must continue to move forward after the collision, with the same speed and in the same direction.

[DF*] A system of $n$ particles has this important equation

$$M\vec{r}_{com} = m_1\vec{r}_1 + m_2\vec{r}_2 + m_3\vec{r}_3 + \cdots + m_n\vec{r}_n \tag{11.13}$$

in which $M$ is the system's total mass and $\vec{r}_{com}$ is the vector locating the position of the system's center of mass.

Differentiating with respect to time gives

$$M\vec{v}_{com} = m_1\vec{v}_1 + m_2\vec{v}_2 + m_3\vec{v}_3 + \cdots + m_n\vec{v}_n \tag{11.14}$$

Here $\vec{v}_i = \frac{d\vec{r}_i}{dt}$ is the velocity of the $i$th particle, and $\vec{v}_{com} = \frac{d\vec{r}_{com}}{dt}$ is the velocity of the center of mass.

Differentiating with respect to time leads to

$$M\vec{a}_{com} = m_1\vec{a}_1 + m_2\vec{a}_2 + m_3\vec{a}_3 + \cdots + m_n\vec{a}_n \tag{11.15}$$

Here $\vec{a}_i = \frac{d\vec{v}_i}{dt}$ is the acceleration of the $i$th particle, and $\vec{a}_{com} = \frac{d\vec{v}_{com}}{dt}$ is the acceleration of the center of mass. Although the center of mass is just a geometrical point, it has a position, a velocity, and an acceleration, as if it were a particle.

From Newton's second law, $m_i\vec{a}_i$ is equal to the resultant force $\vec{F}_i$ that acts on the $i$th particle. Thus

$$M\vec{a}_{com} = \vec{F}_1 + \vec{F}_2 + \vec{F}_3 + \cdots + \vec{F}_n \tag{11.16}$$

Among the forces that contribute to the right side will be forces that the particles of the system exert on each other (internal forces) and forces exerted on the particles from outside the system (external forces).
By Newton's third law, the internal forces form third-law force pairs and cancel out in the sum that appears on the right side. What remains is the vector sum of all the external forces that act on the system.

[DF*] **Example:**
If the particles in a system all move together, the com moves with them. But what happens when they move in different directions with different accelerations?
We have three particles that are initially at rest. Each experiences an external force due to bodies outside the three-particle system. The directions are indicated, and the magnitudes are $F_1 = 6.0\ N$, $F_2 = 12\ N$, and $F_3 = 14\ N$. What is the acceleration of the center of mass of the system, and in what direction does it move?
**Solution:**

**Figure 11.3:** *Three particles, initially at rest in the positions shown, are acted on by the external forces shown. The center of mass (com) of the system is marked. The position of the center of mass is marked by a black dot.*

We can treat the center of mass as if it were a real particle, with a mass equal to the system's total mass $M = 16\ kg$. We can also treat the three external forces as if they act at the center of mass.

We can now apply Newton's second law ($\vec{F}_{net} = m\vec{a}$) to the center of mass

$$\vec{F}_{net} = M\vec{a}_{com} \tag{11.17}$$

or

$$\vec{F}_1 + \vec{F}_2 + \vec{F}_3 = M\vec{a}_{com}$$

so

$$\vec{a}_{com} = \frac{\vec{F}_1 + \vec{F}_2 + \vec{F}_3}{M} \tag{11.18}$$

it tells us that the acceleration $\vec{a}_{com}$ of the center of mass is in the same direction as the net external force $\vec{F}_{net}$ on the system. Because the particles are initially at rest, the center of mass must also be at rest. As the center of mass then begins to accelerate, it must move off in the common direction of $\vec{a}_{com}$ and $\vec{F}_{net}$.

Along the $x$ axis, we have

$$\begin{aligned} a_{com,x} &= \frac{F_{1x} + F_{2x} + F_{3x}}{M} \\ &= \frac{-6 + (12)(\cos\ 45^0) + 14}{16} \\ &= 1.03\ m/s^2 \end{aligned}$$

Along the $y$ axis, we have

$$\begin{aligned} a_{com,y} &= \frac{F_{1y} + F_{2y} + F_{3y}}{M} \\ &= \frac{0 + (12)(\sin\ 45^0) + 0}{16} \\ &= 0.53\ m/s^2 \end{aligned}$$

From these components, we find that $\vec{a}_{com}$ has the magnitude

$$\begin{aligned} a_{com} &= \sqrt{(a_{com,x})^2 + (a_{com,y})^2} \\ &= 1.16\ m/s^2 \end{aligned}$$

and the angle (from the positive direction of the $x$ axis)

$$\theta = \tan^{-1}\left(\frac{a_{com,y}}{a_{com,x}}\right) = 27^{[}0]$$

[DF*] **Example:**
Ricardo, of mass 80 $kg$, and Carmelita, who is lighter, are enjoying Lake Merced at dusk in a 30 $kg$ canoe. When the canoe is at rest in the placid water, they exchange seats, which are 3.0 $m$ apart and symmetrically located with respect to the canoe's center. If the canoe moves 40 $cm$ horizontally relative to a pier post, what is Carmelita's mass?

**Solution:**

**Figure 11.4:** *The forces are now transferred to the center of mass of the system, which behaves like a particle with a mass M equal to the total mass of the system.*

We denote the mass of Ricardo as $M_R$ and that of Carmelita as $M_C$. Let the center of mass of the two-person system (assumed to be closer to Ricardo) be a distance $x$ from the middle of the canoe of length $L$ and mass $m$. Then

$$M_R\left(\frac{L}{2}-x\right)=mx+M_C\left(\frac{L}{2}+x\right)$$

Now, after they switch positions, the center of the canoe has moved a distance $2x$ from its initial position. Therefore,

$$x=\frac{0.4}{2}=0.2$$

which we substitute into the above equation to solve for $M_C$:

$$\begin{aligned}M_C&=\frac{M_R\left(\frac{L}{2}-x\right)-mx}{\frac{L}{2}+x}\\&=\frac{80\left(\frac{3}{2}-0.2\right)-(30)(0.2)}{\frac{3}{2}+0.2}\\&=58\ kg\end{aligned}$$

[DF*] **Example:**
A 4.5 $kg$ dog stands on an 18 $kg$ flatboat at distance $D=6.1\ m$ from the shore. It walks 2.4 $m$ along the boat toward shore and then stops. Assuming no friction between the boat and the water, find how far the dog is then from the shore?

**Solution:**
There is no net horizontal force on the dog-boat system, so their center of mass does not move. Therefore

$$M\triangle x_{com}=0=m_b\triangle x_b+m_d\triangle x_d$$

which implies

$$|\triangle x_b|=\frac{m_d}{m_b}|\triangle x_d|$$

Now we express the geometrical condition that relative to the boat the dog has moved a distance $d=2.4\ m$:

$$|\triangle x_b|+|\triangle x_d|=d$$

which accounts for the fact that the dog moves one way and the boat moves the other. We substitute for $|\triangle x_b|$ from above:

$$\frac{m_d}{m_b}|\triangle x_d| + |\triangle x_d| = d$$

which leads to

$$|\triangle x_d| = \frac{d}{1+\frac{m_d}{m_b}} = \frac{2.4}{1+\frac{4.5}{18}} = 1.92\ m$$

The dog is therefore 1.92 $m$ closer to the shore than initially (where it was $D = 6.1\ m$ from it). Thus, it is now $D - |\triangle x_d| = 4.2\ m$ from the shore.

[DF*] **Example:**
An 80 $kg$ man is on a ladder hanging from a balloon that has a total mass of 320 $kg$ (including the basket passenger). The balloon is initially stationary relative to the ground. If the man on the ladder begins to climb at 2.5 $m/s$ relative to the ladder.

(a) In what direction does the balloon move?

(b) At what speed does the balloon move?

(c) If the man then stops climbing, what is the speed of the balloon?

**Solution:**

(a) Since the center of mass of the man-balloon system does not move, the balloon will move downward with a certain speed $u$ relative to the ground as the man climbs up the ladder.

(b) The speed of the man relative to the ground is $v_g = v - u$. Thus, the speed of the center of mass of the system is

$$\begin{aligned} v_{com} &= \frac{mv_g - Mu}{M+m} \\ &= \frac{m(v-u) - Mu}{M+m} \\ &= 0 \end{aligned}$$

This yields

$$\begin{aligned} u &= \frac{mv}{M+m} \\ &= \frac{(80)(2.5)}{320+80} \\ &= 0.5\ m/s \end{aligned}$$

(c) Now that there is no relative motion within the system, the speed of both the balloon and the man is equal to $v_{com}$, which is zero. So the balloon will again be stationary.

[DF*] **Example:**
An unmanned space probe (of mass $m$ and speed $v$ relative to the Sun) approaches the planet Jupiter (of mass $M$ and speed $V_J$ relative to the Sun). The spacecraft rounds the planet and departs in the opposite direction. What is its speed (in kilometers per second), relative to the Sun, after this slingshot encounter, which can be analyzed as a collision? Assume $v = 10.5\ km/s$ and $V_J = 13\ km/s$ (the orbital speed of Jupiter). The mass of Jupiter is very

much greater than the mass of the spacecraft ($M \gg m$).

**Solution:**
Conservation of linear momentum gives

$$mv + MV_J = mv_f + MV_{Jf}$$

Similarly, the total kinetic energy is conserved:

$$\frac{1}{2}mv^2 + \frac{1}{2}MV_J^2 = \frac{1}{2}mv_f^2 + \frac{1}{2}MV_{Jf}^2$$

Solving for $v_f$ and $V_{Jf}$, we obtain:

$$\begin{aligned} v_{1f} &= \frac{m-M}{m+M}v + \frac{2M}{m+M}V_J \\ V_{Jf} &= \frac{2m}{m+M}v + \frac{M-m}{m+M}V_J \end{aligned}$$

Since $m \ll M$, the above expressions can be simplified to

$$\begin{aligned} v_{1f} &\approx -v + 2V_J \\ V_{Jf} &\approx V_J \end{aligned}$$

The velocity of the probe relative to the Sun is

$$\begin{aligned} v_{1f} &\approx -v + 2V_J \\ &= -(10.5) + 2(-13) \\ &= -36.5\ km/s \end{aligned}$$

The speed is $|v_{1f}| = 36.5\ km/s$.

## IV. Simulation of A Shell that Explodes into Two Fragments with Box2D

Taken from problem number 13 chapter 9 from [6].

A shell is shot with an initial velocity $\vec{v}_0$ of 20 $m/s$, at an angle of $\theta_0 = 60^0$ with the horizontal. At the top of the trajectory, the shell explodes into two fragments of equal mass. One fragment, whose speed immediately after the explosion is zero, falls vertically. How far from the gun does the other fragment land, assuming that the terrain is level and that air drag is negligible?

**Solution:**

A shell explodes into two segments at the top of its trajectory. Knowing the motion of one segment allows us to analyze the motion of the other using the momentum conservation principle.

We need to find the coordinates of the point where the shell explodes and the velocity of the fragment that does not fall straight down. The coordinate origin is at the firing point, the $+x$ axis is rightward, and the $+y$ direction is upward. The $y$ component of the velocity is given by $v = v_{0y} - gt$ and this is zero at time $t = \frac{v_{0y}}{g} = \frac{v_0}{g} \sin\ \theta_0$, where $v_0$ is the initial speed and $\theta_0$ is the firing angle. The coordinates of the highest point on the trajectory are

$$\begin{aligned} x &= v_{0x} t \\ &= v_0 t\ \cos\ \theta_0 \\ &= \frac{v_0}{g} \sin\ \theta_0\ \cos\ \theta_0 \\ &= \frac{(20)^2}{9.8} \sin\ 60^0\ \cos\ 60^0 \\ &= 17.7\ m \end{aligned}$$

and

$$\begin{aligned} x &= v_{0y} t - \frac{1}{2} g t^2 \\ &= v_0 \left( \frac{v_0}{g} \sin\ \theta_0 \right)\ \sin\ \theta_0 - \frac{1}{2} \left( \frac{v_0}{g} \sin\ \theta_0 \right)^2 g \\ &= \frac{1}{2} \frac{v_0^2}{g} \sin^2\ \theta_0 \\ &= \frac{1}{2} \frac{(20)^2}{9.8} \sin^2\ 60^0 \\ &= 15.3\ m \end{aligned}$$

Since no horizontal forces act, the horizontal component of the momentum is conserved. In addition, since one fragment has a velocity of zero after the explosion, the momentum of the other equals the momentum of the shell before the explosion. At the highest point the velocity of the shell is $v_0\ \cos\ \theta_0$, in the positive $x$ direction. Let $M$ be the mass of the shell and let $V_0$ be the

velocity of the fragment. Then

$$\begin{aligned} p_i &= p_f \\ Mv_{0x} &= \frac{MV_0}{2} \\ Mv_0 \ \cos \ \theta_0 &= \frac{MV_0}{2} \end{aligned}$$

since the mass of the fragment is $\frac{M}{2}$. This means

$$V_0 = 2v_0 \ \cos \ \theta_0 = 2(20) \cos \ 60^0 = 20 \ m/s$$

This information is used in the form of initial conditions for a projection motion problem to determine where the fragment lands.

Resetting our clock, we now analyze a projectile launched horizontally at time $t = 0$ with a speed of 20 $m/s$ from a location having coordinates $x_0 = 17.7 \ m$. $y_0 = 15.3 \ m$. Its $y$ coordinate is given by $y = y_0 - \frac{1}{2}gt^2$, and when it lands this is zero.

The time of landing is

$$t = \sqrt{\frac{2y_0}{g}}$$

and the $x$ coordinate of the landing point is

$$\begin{aligned} x &= x_0 + V_0 t \\ &= x_0 + V_0 \sqrt{\frac{2y_0}{g}} \\ &= 17.7 + 20\sqrt{\frac{2(15.3)}{9.8}} \\ &= 53 \ m \end{aligned}$$

In the absence of explosion, the shell with a mass $M$ would have landed at

$$\begin{aligned} R &= 2x_0 \\ &= 2v_{0x}t \\ &= 2v_0 \ \cos \ \theta_0 \left( \frac{v_0}{g} \sin \ \theta_0 \right) \\ &= \frac{v_0^2}{g} \sin 2\theta_0 \\ &= \frac{(20)^2}{9.8} \sin 2(60^0) \\ &= 35.3 \ m \end{aligned}$$

which is shorter than $x = 53 \ m$ found above. This makes sense because the broken fragment, having a smaller mass but greater horizontal speed, can travel much farther than the original shell.

```
#include "test.h"
```

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

#include <iostream> // for printf

class AtomicBomb : public Test
{
	public:

	enum
	{
		e_count = 7
	};

	AtomicBomb()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		// Ground body
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-999999990.0f, 0.0f), b2Vec2
				(99999990.0f, 0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
		// Create spaceship
		{
			b2BodyDef bd;
			bd.type = b2_dynamicBody;
			bd.position.Set(-30.0f, 1.5f);
			m_airplane1 = m_world->CreateBody(&bd);

			b2Vec2 vertices[3];

			vertices[0].Set(-2.0f, 0.0f);
			vertices[1].Set(0.0f, 2.0f);
			vertices[2].Set(2.0f, 0.0f);
			shape_airplane.Set(vertices, 3);

			vertices[0].Set(-2.0f, 0.0f);
			vertices[1].Set(-2.0f, -1.0f);
			vertices[2].Set(2.0f, -1.0f);
			vertices[3].Set(2.0f, 0.0f);
			shape_shell.Set(vertices, 4);

			m_pieceairplane = m_airplane1->CreateFixture(&
```

```
                        shape_airplane, 2.0f);
                m_pieceshell = m_airplane1->CreateFixture(&shape_shell
                    , 2.0f);

        }

        m_break = false;
        m_broke = false;
    }

    void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
        override
    {
        if (m_broke)
        {
            // The body already broke.
            return;
        }

        // Should the body break?
        int32 count = contact->GetManifold()->pointCount;

        float maxImpulse = 0.0f;
        for (int32 i = 0; i < count; ++i)
        {
            maxImpulse = b2Max(maxImpulse, impulse->normalImpulses
                [i]);
        }

        if (maxImpulse > 50.0f)
        {
            // Flag the body for breaking.
            m_break = true;
        }
    }

    void Break()
    {
        // Create three bodies from one.

        b2Body* body_airplane1 = m_pieceairplane->GetBody();
        b2Vec2 airplane_center = body_airplane1->GetWorldCenter();

        body_airplane1->DestroyFixture(m_pieceshell);
        m_pieceshell = NULL;

        b2BodyDef bd_airplane;
        bd_airplane.type = b2_dynamicBody;
```

```
            bd_airplane.position = body_airplane1->GetPosition();
            bd_airplane.angle = body_airplane1->GetAngle();

            b2Body* body_airplane2 = m_world->CreateBody(&bd_airplane);
            m_pieceshell = body_airplane2->CreateFixture(&shape_shell,
                1.0f);

            // Compute consistent velocities for new bodies based on
            // cached velocity.
            b2Vec2 airplane_center1 = body_airplane1->GetWorldCenter();
            b2Vec2 airplane_center2 = body_airplane2->GetWorldCenter();

            b2Vec2 airplane_velocity1 = m_airplanevelocity + b2Cross(
                m_airplaneangularVelocity, airplane_center1 -
                airplane_center);
            b2Vec2 airplane_velocity2 = m_airplanevelocity + b2Cross(
                m_airplaneangularVelocity, airplane_center2 -
                airplane_center);

            body_airplane1->SetAngularVelocity(m_airplaneangularVelocity
                );
            body_airplane1->SetLinearVelocity(airplane_velocity1);

            body_airplane2->SetAngularVelocity(m_airplaneangularVelocity
                );
            body_airplane2->SetLinearVelocity(airplane_velocity2);
            //printf("%4.2f %4.2f \n", airplane_velocity2.x,
                airplane_velocity2.y);


        }

        void Keyboard(int key) override
        {
            switch (key)
            {
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                    g_camera.m_center = b2Vec2(2.0f, 10.0f);
                    g_camera.m_zoom = 2.3f; // zoom out camera
                }
                case GLFW_KEY_D:
                m_break = true;

                break;
                case GLFW_KEY_S:
```

```
                int theta = 60;
                float v0 = 20.0f;
                m_airplane1->SetAngularVelocity(-0.5f);
                m_airplane1 ->SetLinearVelocity(b2Vec2(v0*cos(theta*
                    DEGTORAD), v0*sin(theta*DEGTORAD)));
                break;
        }
}

void Step(Settings& settings) override
{
        b2Vec2 v = m_airplane1->GetLinearVelocity();
        float angular_vel = m_airplane1->GetAngularVelocity();
        float angle = -60; // to make the movement to the right
        float airplane_vel = 20.0f;
        b2MassData massData = m_airplane1->GetMassData();
        b2Vec2 airplane_position = m_airplane1->GetPosition();
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz



        if (m_break)
        {
                Break();
                m_broke = true;
                m_break = false;
        }

        // Cache velocities to improve movement on breakage.
        if (m_broke == false)
        {
                m_airplanevelocity = m_airplane1->GetLinearVelocity();
                m_airplaneangularVelocity = m_airplane1->
                    GetAngularVelocity();
        }

        g_debugDraw.DrawString(5, m_textLine, "Press C to focus the
            camera on the airplane");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press S to flies the
            airplane");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press D to drop the
            bomb shell");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Airplane position =
                    (%4.1f, %4.1f)", airplane_position.x, airplane_position.y
                    );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                    .mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
                    (%4.1f, %4.1f)", v.x, v.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Angular velocity =
                    %4.2f", angular_vel);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Total linear velocity=
                     %.6f", v);
                m_textLine += m_textIncrement;

                if(!m_fixed_camera)
                {
                       g_camera.m_center = m_airplane1->GetPosition();
                }
                printf("%4.2f %4.2f %4.2f \n", airplane_position.x,
                    airplane_position.y, v);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new AtomicBomb;
        }

        b2Body* m_airplane1;
        b2Vec2 m_airplanevelocity;
        float m_airplaneangularVelocity;
        b2PolygonShape shape_airplane;
        b2PolygonShape shape_shell;
        b2Fixture* m_pieceshell;
        b2Fixture* m_pieceairplane;

        bool m_fixed_camera;
        float m_time;
        bool m_broke;
        bool m_break;
};

static int testIndex = RegisterTest("Center of Mass", "Atomic Bomb",
```

```
        AtomicBomb::Create);
```

**C++ Code 80:** *tests/centerofmass_atomicbomb.cpp "Center of Mass Atomic Bomb Box2D"*

Some explanations for the codes:

- This is where we can set the initial speed $v_0$ and the angle of elevation when the shell is shot $\theta_0$. We set the angular velocity of $-0.5$ to make the shell move realistically.

```
...

void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                        g_camera.m_center = b2Vec2(2.0f, 10.0f);
                        g_camera.m_zoom = 2.3f; // zoom out camera
                }
                case GLFW_KEY_D:
                        m_break = true;

                        break;
                case GLFW_KEY_S:
                        int theta = 60;
                        float v0 = 20.0f;
                        m_airplane1->SetAngularVelocity(-0.5f);
                        m_airplane1 ->SetLinearVelocity(b2Vec2(v0*
                            cos(theta*DEGTORAD), v0*sin(theta*
                            DEGTORAD)));
                        break;
        }
}

...
```

**Figure 11.5:** *The simulation of shell that is shot with an initial velocity of $\vec{v}_0 = 30\ m/s$ at an angle of $\theta_0 = 60^0$ and not exploding into two fragments (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_atomicbomb.cpp****).*

**Figure 11.6:** *The simulation of shell that is shot with an initial velocity of $\vec{v}_0 = 30\ m/s$ at an angle of $\theta_0 = 60^0$ and explodes into two fragments of equal mass (the current simulation code can be located in:* ***DFSimulator-C/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_atomicbomb.cpp****).*

## V. Linear Momentum

[DF*] The linear momentum of a particle is a vector quantity $\vec{p}$ that is defined as

$$\vec{p} = m\vec{v} \tag{11.19}$$

in which $m$ is the mass of the particle and $\vec{v}$ is its velocity. The SI unit for momentum is the kilogram-meter per second ($kg \cdot m/s$).

[DF*] Newton expressed his second law of motion in terms of momentum:
The time rate of change of the momentum of a particle is equal to the net force acting on the particle and is in the direction of that force.

In equation form this becomes:

$$\vec{F}_{net} = \frac{d\vec{p}}{dt} \tag{11.20}$$

The net external force $\vec{F}_{net}$ on a particle changes the particle's linear momentum $\vec{p}$. Conversely, the linear momentum can be changed only by a net external force. If ther eis no net external force, $\vec{p}$ cannot change.

By substituting for $\vec{p}$ we will obtain

$$\vec{F}_{net} = \frac{d\vec{p}}{dt} = \frac{d}{dt}(m\vec{v}) = m\frac{d\vec{v}}{dt} = m\vec{a}$$

Thus, the relations $\vec{F}_{net} = \frac{d\vec{p}}{dt}$ and $\vec{F}_{net} = m\vec{a}$ are equivalent expressions of Newton's second law of motion for a particle.

[DF*] Consider a system of $n$ particles, each with its own mass, velocity, and linear momentum. The particle may interact with each other, and external forces may act on them. The system as a whole has a total linear momentum $\vec{P}$, which is defined to be the vector sum of the individual particles' linear momentum. Thus,

$$\begin{aligned} \vec{P} &= \vec{p}_1 + \vec{p}_2 + \vec{p}_3 + \cdots + \vec{p}_n \\ &= m_1\vec{v}_1 + m_2\vec{v}_2 + m_3\vec{v}_3 + \cdots + m_n\vec{v}_n \end{aligned} \tag{11.21}$$

We see that

$$\vec{P} = M\vec{v}_{com} \tag{11.22}$$

which is another way to define the linear momentum of a system of particles:
The linear momentum of a system of particles is equal to the product of the total mass $M$ of the system and hte velocity of the center of mass.

[DF*] If we take the time derivative, we find

$$\frac{d\vec{P}}{dt} = M\frac{d\vec{v}_{com}}{dt} = M\vec{a}_{com} \tag{11.23}$$

We can write Newton's second law for a system of particles in the equivalent form

$$\vec{F}_{net} = \frac{d\vec{P}}{dt} \tag{11.24}$$

where $\vec{F}_{net}$ is the net external force acting on the system. This equation is the generalization of the single-particle equation $\vec{F}_{net} = \frac{d\vec{p}}{dt}$ to a system of many particles.

[DF*] **Example:**
A 0.70 $kg$ ball moving horizontally at 5.0 $m/s$ strikes a vertical wall and rebounds with speed 2.0 $m/s$. What is the magnitude of the change in its linear momentum?

**Solution:**
The magnitude of the ball's momentum change is

$$\begin{aligned}\triangle p &= m|v_i - v_f| \\ &= (0.7)|5-(-2)| \\ &= 4.9\ kg \cdot m/s\end{aligned}$$

[DF*] **Example:**
A 2100 $kg$ truck traveling north at 41 $km/h$ turns east and accelerates to 51 $km/h$.

(a) What is the change in the truck's kinetic energy?
(b) What is the magnitude of its momentum?
(c) What is the direction of the change in its momentum?

**Solution:**

(a) The change in the kinetic energy is

$$\begin{aligned}\triangle K &= \frac{1}{2}mv_f^2 - \frac{1}{2}mv_i^2 \\ &= \frac{1}{2}(2100)(51^2 - 41^2)\left(\frac{10^3}{3600}\right)^2 \\ &= 7.5 \times 10^4\ J\end{aligned}$$

with $\frac{10^3}{3600}$ is the conversion from $km$ to $m$ and $h$ to $s$, we power it by 2 because the speeds $(v_i, v_f)$ are powered by 2 too.

(b) The magnitude of the change in velocity is

$$|\triangle\vec{v}| = \sqrt{(-v_i)^2 + (v_f)^2} = \sqrt{(-41)^2 + (51)^2} = 65.4\ km/h$$

so the magnitude of the change in momentum is

$$\begin{aligned}|\triangle\vec{p}| &= m|\triangle\vec{v}| \\ &= 2100(65.4)\left(\frac{10^3}{3600}\right) \\ &= 3.8 \times 10^4\ kg \cdot m/s\end{aligned}$$

(c) The vector $\triangle\vec{p}$ points at an angle $\theta$ south of east, where

$$\theta = \tan^{-1}\left(\frac{v_i}{v_f}\right) = \tan^{-1}\left(\frac{41}{51}\right) = 39^0$$

[DF*] **Example:**
An object is tracked by a radar station and determined to have a position vector given by $\vec{r} = (3500 - 160t)\hat{i} + 2700\hat{j} + 300\hat{k}$, with $\vec{r}$ in meters and $t$ in seconds. The radar station's $x$ axis points east, its $y$ axis north, and its $z$ axis vertically up. If the object is a 250 $kg$ meteorological missile.

(a) What is its linear momentum?

(b) What is its direction of motion?

(c) What is its net force on it?

**Solution:** The velocity of the object is

$$\begin{aligned}\vec{v} &= \frac{d\vec{r}}{dt} \\ &= \frac{d}{dt}\left((3500 - 160t)\hat{i} + 2700\hat{j} + 300\hat{k}\right) \\ &= -(160\ m/s)\hat{i}\end{aligned}$$

(a) The linear momentum is

$$\begin{aligned}\vec{p} &= m\vec{v} \\ &= (250)(-160) \\ &= (-4 \times 10^4\ kg \cdot m/s)\hat{i}\end{aligned}$$

(b) The object is moving west because of the $-\hat{i}$ direction.

(c) Since the value of $\vec{p}$ does not change with time, the net force exerted on the object is zero.

## VI. Collision and Impulse

[DF*] The momentum $\vec{p}$ of any particle-like body cannot change unless a net external force changes it.

For example, we could arrange for the body to collide with a baseball bat. In such a collision (or crash), the external force on the body is brief, has large magnitude, and suddenly changes the body's momentum.

We need to consider a simple collision in which a moving particle-like body (a projectile) collides with some other body (target).

[DF*] **Single Collision**

Let the projectile be a ball and the target be a bat. The collision is brief, and the ball experiences a force that is great enough to slow, stop or even reverse its motion.

The ball experiences a force $\vec{F}(t)$ that varies during the collision and changes the linear momentum $\vec{p}$ of the ball. That change is related to the force by Newton's second law written in the form $\vec{F} = \frac{d\vec{p}}{dt}$.

By rearranging this second-law expression, we see that, in time interval $dt$, the change in the ball's momentum is

$$d\vec{p} = \vec{F}(t)\ dt \tag{11.25}$$

We can find the net change in the ball's momentum due to the collision if we integrate both sides from a time $t_i$, just before the collision to a time $t_f$ just after the collision

$$\int_{t_i}^{t_f} d\vec{p} = \int_{t_i}^{t_f} \vec{F}(t)\ dt \tag{11.26}$$

The left side of this equation gives us the change in momentum: $\vec{p}_f - \vec{p}_i = \triangle\vec{p}$. The right side, which is a measure of both the magnitude and the duration of the collision force, is called the impulse $\vec{J}$ of the collision:

$$\vec{J} = \int_{t_i}^{t_f} \vec{F}(t)\ dt \tag{11.27}$$

Thus, the change in an object's momentum is equal to the impulse on the object:

$$\triangle\vec{p} = \vec{J} \tag{11.28}$$

This expression can also be written in the vector form

$$\vec{p}_f - \vec{p}_i = \vec{J} \tag{11.29}$$

and in such component forms as

$$\triangle p_x = J_x \tag{11.30}$$

and

$$p_{fx} - p_{ix} = \int_{t_i}^{t_f} F_x\ dt \tag{11.31}$$

[DF*] If we have a function for $\vec{F}(t)$, we can evaluate $\vec{J}$ by integrating the function. If we have a plot of $\vec{F}$ versus time $t$, we can evaluate $\vec{J}$ by finding the area between the curve and the $t$ axis. In many situations we do not know how the force varies with time bit we do know the average magnitude $F_{avg}$ of the force and the duration $\triangle t(= t_f - t_i)$ of the collision. Then we can write the magnitude of the impulse as

$$J = F_{avg}\triangle t \tag{11.32}$$

The average force is plotted versus time. The area under the curve is equal to the area under the curve for the actual force $F(t)$ because both areas are equal to impulse magnitude $J$.

At any instant, Newton's third law tells us that the force on the bat has the same magnitude but the opposite direction as the force on the ball. This means that the impulse on the bat has the same magnitude but the opposite direction as the impulse on the ball.

[DF*] **Series of Collisions**

Now let's consider the force on a body when it udnergoes a series of identical, repeated collisions.

A steady stream of projectile bodies, with identical mass $m$ and linear momenta $m\vec{v}$, moves along an $x$ axis and collides with a target body that is fixed in place.

Let $n$ be the number of projectiles that collide in a time interval $\triangle t$. Because the motion is along only the $x$ axis, we can use the components of the momenta along that axis. Thus, each projectile has initial momentum $mv$ and undergoes a change $\triangle p$ in linear momentum because of the collision. The tota; change in linear momentum for $n$ projectiles during interval $\triangle t$ is along the $x$ axis and has the same magnitude of $n\triangle p$ but is in the opposite direction. We can write this relation in component form as

$$J = -n\triangle p \tag{11.33}$$

where the minus sign indicates that $J$ and $\triangle p$ have opposite directions.

[DF*] By rearranging and substituting, we find the average force $F_{avg}$ acting on the target during the collisions:

$$F_{avg} = \frac{J}{\triangle t} = -\frac{n}{\triangle t}\triangle p = -\frac{n}{\triangle t}m\triangle v \tag{11.34}$$

This equation gives us $F_{avg}$ in terms of $\frac{n}{\triangle t}$, the rate at which the projectiles collide with the target, and $\triangle v$, the change in the velocity of those projectiles.

[DF*] If the projectiles stop upon impact, then we can substitute, for $\triangle v$,

$$\triangle v = v_f - v_i = 0 - v = -v \tag{11.35}$$

where $v_i = v$ and $v_f = 0$ are the velocities before and after the collision, respectively. If, instead, the projectiles bounce (rebound) directly backward from the target with no change in spedd, then $v_f = -v$ and we can substitute

$$\triangle v = v_f - v_i = -v - v = -2v \tag{11.36}$$

In time interval $\triangle t$, an amount of mass $\triangle m = nm$ collides with the target. With this result, we can rewrite

$$F_{avg} = -\frac{\triangle m}{\triangle t}\triangle v \tag{11.37}$$

This equation gives the average force $F_{avg}$ in terms of $\frac{\triangle m}{\triangle t}$, the rate at which mass collides with the target.

[DF*] **Example:**
Until his seventies, Henri LaMothe excited audiences by belly-flopping from a height of 12 $m$ into 30 $cm$ of water. Assuming that he stops just as he reaches the bottom of the water and estimating his mass, find the magnitude of the impulse on him from the water.

**Solution:**
We estimate his mass in the neighborhood of 70 $kg$ and compute the upward force $F$ of the water from Newton's second law:

$$F - mg = ma$$

where we have chosen $+y$ upward, so that $a > 0$ (the acceleration is upward since it represents a deceleration of his downward motion through the water).

His speed when he arrives at the surface of the water is found from energy conservation:

$$v = \sqrt{2gh}$$

where $h = 12\ m$, and since the deceleration $a$ reduces the speed to zero over a distance $d = 0.30\ m$ we also obtain

$$v = \sqrt{2ad}$$

We use these observations in the following.

Equating our two expressions for $v$ leads to $a = \frac{gh}{d}$. Our force equation, then, leads to

$$F = mg + m\frac{gh}{d}$$

which yields $F \approx 2.8 \times 10^4\ kg$. Since we are not at all certain of his mass, we express this as a guessed-at-range (in $kN$) $25 < F < 30$.

Since $F \gg mg$, the impulse $\vec{J}$ due to the net force (while he is in contact with the water) is overwhelmingl caused by the upward force of the water:

$$F\ dt = \vec{J}$$

to a good approximation. Thus,

$$\begin{aligned} F\ dt &= \vec{p}_f - \vec{p}_i \\ &= 0 - m(-\sqrt{2gh}) \end{aligned}$$

the minus sign with the initial velocity is due to the fact that downward is the negative direction, which yields

$$m\sqrt{2gh} = (70)\sqrt{2(9.8)(12)} = 1.1 \times 10^3\ kg \cdot m/s$$

Expressing this as a range we estimate

$$1.0 \times 10^3\ kg \cdot m/s < \int F\ dt < 1.2 \times 10^3\ kg \cdot m/s$$

[DF*] **Example:**
In February 1955, a paratrooper fell 370 $m$ from an airplane without being able to open his chute but happened to land in snow, suffering only minor injuries. Assume that his speed at impact was 56 $m/s$ (terminal speed), that his mass (including gear) was 85 $kg$, and that the magnitude of the force on him from the snow was at the survivable limit of $1.2 \times 10^5$ $N$.

(a) What is the minimum depth of snow that would have stopped him safely ?

(b) What is the magnitude of the impulse on him from the snow?

**Solution:**
We choose $+y$ upward, which implies $a > 0$ (the acceleration is upward since it represents a deceleration of his downward motion through the snow).

(a) The maximum deceleration $a_{max}$ of the paratrooper (of mass $m$ and initial speed $v = 56$ $m/s$) is found from Newton's second law

$$F_{snow} - mg = ma_{max}$$

where we require $F_{snow} = 1.2 \times 10^5$ $N$. Using

$$v^2 = 2a_{max}d$$

we find the minimum depth of snow for the man to survive:

$$\begin{aligned} d &= \frac{v^2}{2a_{max}} \\ &= \frac{mv^2}{2(F_{snow} - mg)} \\ &= \frac{(85)(56)}{2(1.2 \times 10^5 - (85)(9.8))} \\ &\approx 1.1\ m \end{aligned}$$

(b) His short trip through the snow involves a change in momentum

$$\triangle\vec{p} = \vec{p}_f - \vec{p}_i = 0 - (85)(-56) = -4.8 \times 10^3\ kg \cdot m/s$$

or $|\triangle\vec{p}| = 4.8 \times 10^3\ kg \cdot m/s$. The negative value of the initial velocity is due to the fact that downward is the negative direction. By the impulse-momentum theorem, this equals the impulse due to the net force $F_{snow} - mg$, but since $F_{snow} \gg mg$ we can approximate this as the impulse on him just from the snow.

[DF*] **Example:**
In tae-kwon-do, a hand is slammed down onto a target at a speed of 13 $m/s$ and comes to a stop during the 5.0 $ms$ collision. Assume that during the impact the hand is independent of the arm and has a mass of 0.70 $kg$.

(a) What is the magnitude of the impulse?

(b) What is the average force on the hand from the target?

**Solution:**

(a) The magnitude of the impulse is

$$\begin{aligned} J &= |\triangle p| \\ &= m|\triangle v| \\ &= mv \\ &= (0.70)(13) \\ &= 9.1\ N \cdot s \end{aligned}$$

(b) With duration of $\triangle t = 5 \times 10^{-3}\ s$ for the collision, the average force is

$$F_{avg} = \frac{J}{\triangle t} = \frac{9.1}{5 \times 10^{-3}} \approx 1.8 \times 10^3\ N$$

[DF*] **Example:**
Suppose a gangster sprays Superman's chest with 3 $g$ bullets at the rate of 100 $bullets/min$, and the speed of each bullet is 500 $m/s$. Suppose too that the bullets rebound straight back with no change in speed. What is the magnitude of the average force on Superman's chest?
**Solution:**
We choose the positive direction in the direction of rebound so that $\vec{v}_f > 0$ and $\vec{v}_i < 0$. SInce they have the same speed $v$, we write this as

$$\begin{aligned} \vec{v}_f &= v \\ \vec{v}_i &= -v \end{aligned}$$

Therefore, the change in momentum for each bullet of mass $m$ is

$$\triangle \vec{p} = m\triangle v = 2mv$$

Consequently, the total change in momentum for the 100 bullets (each minute)

$$\triangle \vec{P} = 100\triangle \vec{p} = 200mv$$

The average force is then

$$\begin{aligned} \vec{F}_{avg} &= \frac{\triangle \vec{P}}{\triangle t} \\ &= \frac{(200)(3 \times 10^{-3})(500)}{(1)60} \\ &\approx 5\ N \end{aligned}$$

[DF*] **Example:**
Basilisk lizards can run across the top of a water surface. With each step, a lizard first slaps its foot against the water and then pushes it down into the water rapidly enough to form an air cavity around the top of the foot. To avoid having to pull the foot back up against water drag in order to complete the step, the lizard withdraws the foot before water can flow into the air cavity.

If the lizard is not to sink, the average upward impulse on the lizard during this full action of slap, downward push, and withdrawal must match the downward impulse due to the

gravitational force.

Suppose the mass of a basilisk lizard is 90.0 $g$, the mass of each foot is 3.0 $g$, the speed of a foot as it slaps the water is 1.5 $m/s$, and the time for a single step is 0.6 $s$.

(a) What is the magnitude of the impulse on the lizard during the slap? (Assume this impulse is directly upward.)

(b) During the 0.6 $s$ duration of a step, what is the downward impulse on the lizard due to the gravitational force?

(c) Which action, the slap or the push,provides the primary support for the lizard, or are they approximately equal in their support?

**Solution:**

(a) Choosing upward as the positive direction, the momentum change of the foot is

$$\begin{aligned}\triangle\vec{p} &= 0 - m_{foot}\vec{v}_i \\ &= 0 - (0.003)(-1.5) \\ &= 4.5 \times 10^3 N \cdot s\end{aligned}$$

(b) Now treating downward as the positive direction, we have

$$\begin{aligned}\vec{J} &= \vec{F}_{avg}\triangle t \\ &= m_{lizard}g\triangle t \\ &= (0.09)(9.8)(0.6) \\ &= 0.529\ N \cdot s\end{aligned}$$

(c) Push is what provides the primary support.

[DF*] **Example:**
A soccer player kicks a soccer ball of mass 0.45 $kg$ that is initially at rest. The foot of the player is in contact with the ball for $3 \times 10^{-3}$ $s$, and the force of the kick is given by

$$F(t) = [(6 \times 10^6)t - (2 \times 10^9)t^2]N$$

for $0 \leq t \leq 3 \times 10^{-3}$ $s$, where $t$ is in seconds.

(a) Find the magnitude of the impulse on the ball due to the kick.

(b) Find the magnitude ofthe average force on the ball from the player's foot during the period of contact.

(c) Find the magnitude of the maximum force on the ball from the player's foot during the period of contact.

(d) Find the magnitude of the ball's velocity immediately after it loses contact with the player's foot.

**Solution:**
We're given the force as a function of time, and asked to calculate the corresponding impulse, the average force and the maximum force.

Since the motion is one-dimensional, we work with the magnitudes of the vector quantities. The impulse $J$ due to a force $F(t)$ exerted on a body is

$$J = \int_{t_i}^{t_f} F(t)\ dt = F_{avg} \triangle t$$

where $F_{avg}$ is the average force and $\triangle t = t_f - t_i$. To find the time at which the maximum force occurs, we set the derivative of $F$ with respect to time equal to zero, and solve for $t$.

(a) We take the force to be in positive direction, at least for earlier times. Then the impulse is

$$\begin{aligned} J &= \int_{t_i}^{t_f} F(t)\ dt \\ &= \int_0^{3\times10^{-3}} F\ dt \\ &= \int_0^{3\times10^{-3}} \left[(6\times10^6)t - (2\times10^9)t^2\right]\ dt \\ &= \left[\frac{1}{2}(6\times10^6)t^2 - \frac{1}{3}(2\times10^9)t^3\right]\Big|_0^{3\times10^{-3}} \\ &= 9.0\ N\cdot s \end{aligned}$$

(b) Using $J = F_{avg}\triangle t$, we find the average force to be

$$F_{avg} = \frac{J}{\triangle t} = \frac{9}{3\times10^{-3}} = 3\times10^3\ N$$

(c) Differentiating $F(t)$ with respect to $t$ and setting it to zero, we have

$$\begin{aligned} \frac{dF}{dt} &= \frac{d}{dt}\left[(6\times10^6)t - (2\times10^9)t^2\right] \\ &= (6\times10^6) - (4\times10^9)t \end{aligned}$$

the maximum force can be computed when $\frac{dF}{dt} = 0$, thus

$$\begin{aligned} \frac{dF}{dt} &= 0 \\ (6\times10^6) - (4\times10^9)t &= 0 \\ t_{max} &= 1.5\times10^{-3}\ s \end{aligned}$$

At that time the force is

$$\begin{aligned} F_{max} &= (6\times10^6)t_{max} - (2\times10^9)t_{max}^2 \\ &= (6\times10^6)(1.5\times10^{-3}) - (2\times10^9)t^2(1.5\times10^{-3}) \\ &= 4.5\times10^3\ N \end{aligned}$$

(d) Since it starts from rest, the ball acquires momentum equal to the impulse from the kick. Let $m$ be the mass of the ball and $v$ its speed as it leaves the foot. The speed of the ball immediately after it loses contact with the player's foot is

$$v = \frac{p}{m} = \frac{J}{m} = \frac{9}{0.45} = 20\ m/s$$

**Figure 11.7:** *The force as function of time. The area under the curve is the impulse J. (the current code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch11-Center of Mass/2D Plot of F(t)/main.cpp****).*

## VII. Simulation for Race Car Wall Collision with Box2D

A race car collides with the racetrack wall. Just before the collision, he is travelling at speed $v_i = 70\ m/s$ along a straight line at $30^0$ from the wall. Just after the collision he is traveling at speed $v_f = 50\ m/s$ along a straight line at $10^0$ from the wall. His mass $m$ is 80 $kg$.

**Figure 11.8:** *The overhead view of the path taken by a race car and its driver as the car slams into the racetrack wall.*

(a) What is the impulse $\vec{J}$ on the driver due to the collision?

(b) The collision lasts for 14 $ms$. What is the magnitude of the average force on the driver during the collision?

**Solution:**
We can treat the driver as a particle-like body and thus apply the physics of this module. However, we cannot calculate $\vec{J}$ directly because we do not know anything about the force $\vec{F}(t)$ on the driver during the collision. That is, we do not have a function $\vec{F}(t)$ or a plot of it and thus cannot integrate to find $\vec{J}$. However, we can find $\vec{J}$ from the change in the driver's linear momentum $\vec{p}$ via $\vec{J} = \vec{p}_f - \vec{p}_i$.

(a) From the driver's momentum $\vec{p}_i$ before the collision, and the momentum $\vec{p}_f$ after the collision, we can write

$$\vec{J} = \vec{p}_f - \vec{p}_i = m\vec{v}_f - m\vec{v}_i = m(\vec{v}_f - \vec{v}_i) \tag{11.38}$$

We could evaluate the right side of this equation directly on a vector-capable calculator

**Figure 11.9:** *The initial momentum $\vec{p}_i$ and final momentum $\vec{p}_f$ of the driver. The collision changes the momentum.*

because we know $m$ is 80 $kg$, $\vec{v}_f$ is 50 $m/s$ at $-10^0$, and $\vec{v}_i$ is 70 $m/s$ at $30^0$. Instead, here we evaluate in component form.

$x$ **component**: Along the $x$ axis we have

$$\begin{aligned} J_x &= m(v_{fx} - v_{ix}) \\ &= (80)((50\cos(-10^0)) - (70\cos(30^0))) \\ &= -910\ kg \cdot m/s \end{aligned}$$

$y$ **component**: Along the $y$ axis we have

$$\begin{aligned} J_y &= m(v_{fy} - v_{iy}) \\ &= (80)((50\sin(-10^0)) - (70\sin(30^0))) \\ &= -3495\ kg \cdot m/s \end{aligned}$$

The impulse is then

$$\vec{J} = (-910\hat{\imath} - 3495\hat{\jmath})kg \cdot m/s$$

which means the impulse magnitude is

$$J = \sqrt{J_x^2 + J_y^2} = 3616\ kg \cdot m/s$$

The angle of $\vec{J}$ is given by

$$\theta = \tan^{-1}\left(\frac{J_y}{J_x}\right) = 75.4^0 + 180^0 = 255.4^0 = -105^0$$

Recall that the physically correct result of an inverse tangent might be the displayed answer plus $180^0$. We can tell which is correct here by drawing the components of $\vec{J}$.

**Figure 11.10:** *The impulse $\vec{J}$ on the driver during the collision. The impulse on the car is equal to the change in the momentum.*

(b) The magnitude $F_{avg}$ from $J = F_{avg}\triangle t$, is the ratio of the impulse magnitude $J$ to the duration $\triangle t$ of the collision.

We have

$$\begin{aligned} F_{avg} &= \frac{J}{\triangle t} \\ &= \frac{3616}{0.014} \\ &= 2.583 \times 10^3\ N \end{aligned}$$

Using $F = ma$ with $m = 80\ kg$, you can show that the magnitude of the driver's average acceleration during the collision is about $3.22 \times 10^3\ m/s^2 = 329g$, which is fatal ($329g$ is 329 times $g = 9.8\ m/s^2$, not gram from the SI unit).

Mechanical engineers attempt to reduce the chances of a fatality by designing and building a racetrack walls with more "give," so that a collision lasts longer. For example if the collision here lasted 10 times longer and the other data remained the same, the magnitudes of the average force and the average acceleration would be 10 times less and probably survivable.

```
#include "test.h"
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

class RacecarWallCollision: public Test
{
       public:
       RacecarWallCollision()
       {
              m_speed = 1000.0f;
              // Create the line below
              b2Body* ground = NULL;
              {
                     b2BodyDef bd;
                     ground = m_world->CreateBody(&bd);

                     b2EdgeShape shape;

                     b2FixtureDef fd;
                     fd.shape = &shape;
                     fd.density = 0.0f;
                     fd.friction = 0.6f;

                     shape.SetTwoSided(b2Vec2(-20.0f, 0.0f), b2Vec2(300.0f,
                          0.0f));
                     ground->CreateFixture(&fd);

              }

              // Create the line that is 30 degree toward the positive x
                  axis
              {
                     float L = 10.0f;
                     int theta = 30;
                     float deginrad = theta*DEGTORAD;
                     b2BodyDef bd;
                     ground = m_world->CreateBody(&bd);
```

```
                b2EdgeShape shape;

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.density = 0.0f;
                fd.friction = 0.6f;

                shape.SetTwoSided(b2Vec2(70.0f, 0.0f), b2Vec2(cosf(
                    deginrad)*L + 70.0f, sinf(deginrad)*L +0.0f));
                ground->CreateFixture(&fd);

            }
            // Create the upper line
            {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.density = 0.0f;
                fd.friction = 0.6f;

                shape.SetTwoSided(b2Vec2(-20.0f, 10.0f), b2Vec2(300.0f
                    , 10.0f));
                ground->CreateFixture(&fd);

            }
            // Create the vertical line on the right
            {
                b2BodyDef bd;
                ground = m_world->CreateBody(&bd);

                b2EdgeShape shape;

                b2FixtureDef fd;
                fd.shape = &shape;
                fd.density = 0.0f;
                fd.friction = 0.6f;

                shape.SetTwoSided(b2Vec2(300.0f, 0.0f), b2Vec2(300.0f,
                     10.0f));
                ground->CreateFixture(&fd);

            }

            // Car
```

```
{
        b2PolygonShape chassis;
        b2Vec2 vertices[8];
        vertices[0].Set(-1.5f, -0.5f);
        vertices[1].Set(1.5f, -0.5f);
        vertices[2].Set(1.5f, 0.0f);
        vertices[3].Set(0.0f, 0.9f);
        vertices[4].Set(-1.15f, 0.9f);
        vertices[5].Set(-1.5f, 0.2f);
        chassis.Set(vertices, 6);

        b2CircleShape circle;
        circle.m_radius = 0.4f;

        b2BodyDef bd;
        bd.type = b2_dynamicBody;
        bd.position.Set(25.0f, 1.0f);

        b2FixtureDef fd;
        fd.shape = &circle;
        fd.density = 77.0f;
        fd.friction = 0.9f;

        m_car = m_world->CreateBody(&bd);
        m_car->CreateFixture(&chassis, 1.0f);

        bd.position.Set(24.0f, 0.35f);
        m_wheel1 = m_world->CreateBody(&bd);
        m_wheel1->CreateFixture(&fd);

        bd.position.Set(26.0f, 0.45f);
        m_wheel2 = m_world->CreateBody(&bd);
        m_wheel2->CreateFixture(&fd);

        b2WheelJointDef jd;
        b2Vec2 axis(0.0f, 1.0f);

        float mass1 = m_wheel1->GetMass();
        float mass2 = m_wheel2->GetMass();

        float hertz = 4.0f;
        float dampingRatio = 0.1f;
        float omega = 2.0f * b2_pi * hertz;

        jd.Initialize(m_car, m_wheel1, m_wheel1->GetPosition()
            , axis);
        int theta = 30;
        float deginrad = theta*DEGTORAD;
```

```
                // minus means movement to the right / clockwise
                    rotation
                // positive means movement to the left /
                    counterclockwise rotation
                jd.motorSpeed = -100000.0f;
                jd.maxMotorTorque = 400.0f;
                jd.enableMotor = true;
                jd.stiffness = mass1 * omega * omega;
                jd.damping = 2.0f * mass1 * dampingRatio * omega;
                jd.lowerTranslation = -0.25f;
                jd.upperTranslation = 0.25f;
                jd.enableLimit = true;
                m_spring1 = (b2WheelJoint*)m_world->CreateJoint(&jd);

                jd.Initialize(m_car, m_wheel2, m_wheel2->GetPosition()
                    , axis);
                jd.motorSpeed = -100000.0f;
                jd.maxMotorTorque = 400.0f;
                jd.enableMotor = false;
                jd.stiffness = mass2 * omega * omega;
                jd.damping = 2.0f * mass2 * dampingRatio * omega;
                jd.lowerTranslation = -0.25f;
                jd.upperTranslation = 0.25f;
                jd.enableLimit = true;
                m_spring2 = (b2WheelJoint*)m_world->CreateJoint(&jd);
        }
        m_time = 0.0f;
}
bool m_fixed_camera;
float m_time;
b2Body* m_car;
b2Body* m_wheel1;
b2Body* m_wheel2;

float m_speed;
b2WheelJoint* m_spring1;
b2WheelJoint* m_spring2;
int theta = 30;
float deginrad = theta*DEGTORAD;

void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_G:
                m_car->SetLinearVelocity(b2Vec2(5100.0f, 4800.0f));
                break;
```

```
                        case GLFW_KEY_A:
                        m_spring1->SetMotorSpeed(m_speed);
                        m_spring2->SetMotorSpeed(m_speed);
                        break;
                        case GLFW_KEY_S:
                        m_spring1->SetMotorSpeed(0.0f);
                        m_spring2->SetMotorSpeed(0.0f);
                        break;
                        case GLFW_KEY_D:
                        m_spring1->SetMotorSpeed(-m_speed);
                        m_spring2->SetMotorSpeed(-m_speed);
                        break;
                        case GLFW_KEY_C:
                        m_fixed_camera = !m_fixed_camera;
                        if(m_fixed_camera)
                        {
                                g_camera.m_center = b2Vec2(2.0f, -10.0f);
                                g_camera.m_zoom = 3.3f; // zoom out camera
                        }
                        break;
                }
        }

        void Step(Settings& settings) override
        {
                m_time += 1.0f / 60.0f;
                float m = 80;
                float Jx;
                float Jy;
                float vf = 50;
                float vi = 70;
                int theta_i = 30;
                int theta_f = -10;
                Jx = m*(vf*cosf(theta_f*DEGTORAD) - vi*cosf(theta_i*DEGTORAD
                    ));
                Jy = m*(vf*sinf(theta_f*DEGTORAD) - vi*sinf(theta_i*DEGTORAD
                    ));
                float J = sqrt(pow(Jx,2) + pow(Jy,2));
                int theta;
                theta = acosf(Jx/J) * RADTODEG;

                b2MassData massData1 = m_car->GetMassData();
                b2MassData massData2 = m_wheel1->GetMassData();
                b2MassData massData3 = m_wheel2->GetMassData();
                b2Vec2 position1 = m_car->GetPosition();
                b2Vec2 velocity1 = m_car->GetLinearVelocity();

                g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
```

```
                    /tracking");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Keys: left = a, brake
                    = s, right = d, set the car to hit the wall at 30 degree
                    = g ");
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Car Mass = %4.2f",
                    massData1.mass + massData2.mass + massData3.mass );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Car position = (%4.1f,
                     %4.1f)", position1.x, position1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Car velocity = (%4.1f,
                     %4.1f)", velocity1.x, velocity1.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "(J_{x}, J_{y}) = %4.2f
                     i + %4.2f j", Jx, Jy);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "J = %4.2f ", J);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The angle of J = %4.5f
                     ", acosf(Jx/J) * RADTODEG);
                m_textLine += m_textIncrement;

                g_camera.m_center.x = m_car->GetPosition().x;
                Test::Step(settings);
                if(!m_fixed_camera)
                {
                        g_camera.m_center = m_car->GetPosition();
                        g_camera.m_zoom = 3.3f; // zoom out camera
                }
        }

        static Test* Create()
        {
                return new RacecarWallCollision;
        }


};

static int testIndex = RegisterTest("Center of Mass", "Racecar-Wall
    Collision", RacecarWallCollision::Create);
```

**C++ Code 81:** *tests/centerofmass_racecarwallcollision.cpp "Center of Mass Three Particles Box2D"*

Some explanations for the codes:

- We are using this formula to obtain the exact $\theta$:

$$\theta = \cos^{-1}\left(\frac{J_x}{J}\right)$$

  instead of the formula used from the book' solution manual that is using $\tan^{-1}$, the formula of $\cos^{-1}$ is capable of computing the exact degree directly.

```
void Step(Settings& settings) override
{
        m_time += 1.0f / 60.0f;
        float m = 80;
        float Jx;
        float Jy;
        float vf = 50;
        float vi = 70;
        int theta_i = 30;
        int theta_f = -10;
        Jx = m*(vf*cosf(theta_f*DEGTORAD) - vi*cosf(theta_i*
            DEGTORAD));
        Jy = m*(vf*sinf(theta_f*DEGTORAD) - vi*sinf(theta_i*
            DEGTORAD));
        float J = sqrt(pow(Jx,2) + pow(Jy,2));
        int theta;
        theta = acosf(Jx/J) * RADTODEG;

        ...
        ...
```

- If you want to play around with the speed before the collision you can edit for the **GLFW_KEY_G** that will send the car with a very high speed toward the wall, you can set the velocity toward the $x$ axis and toward the $y$ axis as you please.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_G:
                m_car->SetLinearVelocity(b2Vec2(5100.0f, 4800.0f)
                    ); // will make the hypothenuse to be 70
                    before hit the wall
                break;

                ...
                ...
        }
}
```

**Figure 11.11:** *The simulation of a race car that hits the wall at* $30^0$ *(the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_racecarwallcollision.cpp****).*

**Figure 11.12:** *The simulation of a race car that hits the wall at* $30^0$ *(the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_racecarwallcollision.cpp****).*

## VIII. Conservation of Linear Momentum

[DF*] Suppose that the net external force $\vec{F}_{net}$ (and thus the net impulse $\vec{J}$) acting on a system of particles is zero (the system is isolated) and that no particles leave or enter the system (the system is closed). Putting $\vec{F}_{net} = 0$ then yields

$$\frac{d\vec{P}}{dt} = 0$$

which means that

$$\vec{P} = c \tag{11.39}$$

with $c$ as a constant for closed, isolated system. In words:
**If no net external force acts on a system of particles, the total linear momentum $\vec{P}$ of the system cannot change**.

This result is called the law of conservation of linear momentum and is an extremely powerful tool in solving problems. We usually write the law as

$$\vec{P}_i = \vec{P}_f \tag{11.40}$$

for closed, isolated system. In words, this equation says that, for a closed, isolated system, the total linear momentum at some initial time $t_i$ is equal to the total linear momentum at some later time $t_f$.

Momentum should not be confused with energy.
[DF*] Depending on the forces acting on a system, linear momentum might be conserved in one or two directions but not in all directions. However

**If the component of the net external force on a closed system is zero along an axis, then the component of the linear momentum of the system along that axis cannot change.**
[DF*] How can you know if linear momentum can be conserved along, say, an $x$ axis? Check the force components along that axis. If the net of any such components is zero, then the conservation applies.

As an example, suppose that Hisoka throw his card to the Hunter' examinee at the end of the first test. During the card' flight, the only external force acting on the card is the gravitational force $\vec{F}_g$, which is directed vertically downward. Thus, the vertical component of the linear momentum of the card changes, but since no horizontal external force acts on the card, the horizontal component of the linear momentum cannot change.
[DF*] We focus on the external forces acting on a closed system. Although internal forces can change the linear momentum of portions of the system, they cannot change the total linear momentum of the entire system. For example, there are plenty of forces acting between the organs of your body, but they do not propel you across the room (thankfully).
[DF*] **Example:**
A space vehicle is traveling at 4300 $km/h$ relative to Earth when the exhausted rocket motor (mass $4m$) is disengaged and sent backward with a speed of 82 $km/h$ relative to the command module (mass $m$). What is the speed of the command module relative to Earth just after the separation?

**Solution:**

Our notation is as follows: the mass of the motor is $M$; the mass of the module is $m$; the initial speed of the system is $v_0$; the relative speed between the motor and the module is $v_r$; and, the speed of the module relative to the Earth is $v$ after the separation.

Conservation of linear momentum requires

$$(M+m)v_0 = mv + M(v - v_r)$$

Therefore,

$$\begin{aligned} v &= v_0 + \frac{Mv_r}{M+m} \\ &= 4300 + \frac{(4m)(82)}{4m+m} \\ &= 4.4 \times 10^3 \ km/h \end{aligned}$$

[DF*] **Example:**
A stationary block explodes into two pieces $L$ and $R$ that slide across a frictionless floor and then into regions with friction, where they stop. Piece $L$, with a mass of 2.0 $kg$, encounters a coefficient of kinetic friction $\mu_L = 0.40$ and slides to a stop in distance $d_L = 0.15\ m$. Piece $R$ encounters a coefficient of kinetic friction $\mu_R = 0.50$ and slides to stop in a distance $d_R = 0.25\ m$. What was the mass of the block?

**Solution:**
We can think of the sliding-until-stopping as an example of kinetic energy converting into thermal energy. This leads to $v^2 = 2\mu g d$ being true separately for each piece. Thus we can set up a ratio:

$$\left(\frac{v_L}{v_R}\right)^2 = \frac{2\mu_L g d_L}{2\mu_R g d_R} = \frac{2(0.4)(9.8)(0.15)}{2(0.5)(9.8)(0.25)} = \frac{12}{25}$$

But (by conservation of momentum) the ratio of speeds must be inversely proportional to the ratio of masses (since the initial momentum before the explosion was zero). Consequently,

$$\begin{aligned} \left(\frac{m_R}{m_L}\right)^2 &= \frac{12}{25} \\ m_R &= \sqrt{\frac{12}{25}} m_L \\ &= 1.39\ kg \end{aligned}$$

Therefore, the total mass is $m_R + m_L = 3.39\ kg$.

[DF*] **Example:**
Particle $A$ and particle $B$ are held together with a compressed spring between them. When they are released, the spring pushes them apart, adn they then fly off in opposite directions, free of the spring.

The mass $A$ is 2 times the mass of $B$, and the energy stored in the spring was 60 $J$. Assume that the spring has negligible mass and that all its stored energy is transferred to the particles. Once the transfer is complete, what are the kinetic energies of particle $A$ and particle $B$?

**Solution:**
This problem involves both mechanical energy conservation

$$U_i = K_1 + K_2$$

where $U_i = 60\ J$, and momentum conservation

$$0 = m_1\vec{v}_1 + m_2\vec{v}_2$$

where $m_2 = 2m_1$. From the second equation, we find

$$|\vec{v}_1| = 2|\vec{v}_2|$$

which in turn implies (since $v_1 = |\vec{v}_1|$ and likewise for $v_2$)

$$\begin{aligned} K_1 &= \frac{1}{2}m_1v_1^2 \\ &= \frac{1}{2}\frac{1}{2}m_2(2v_2)^2 \\ &= 2\frac{1}{2}m_2v_2^2 \\ &= 2K_2 \end{aligned}$$

We substitute $K_1 = 2K_2$ into the energy conservation relation and find

$$\begin{aligned} U_i &= 2K_2 + K_2 \\ 60 &= 3K_2 \\ K_2 &= 20\ J \end{aligned}$$

and we obtain

$$K_1 = 2K_2 = 40\ J$$

[DF*] **Example:**
Block $L$ of mass $m_L = 1\ kg$ and block $R$ of mass $m_R = 0.5\ kg$ are held in place with a compressed spring between them. When the blocks are released, the spring sends them sliding across a frictionless floor. (The spring has negligible mass and falls to the floor after the blocsk leave it.)

(a) If the spring gives block $L$ a release speed of 1.20 $m/s$ relative to the floor, how far does block $R$ travel in the next 0.8 $s$?

(b) If, instead, the spring gives block $L$ a release speed of 1.2 $m/s$ relative to the velocity that the spring gives block $R$, how far does block $R$ travel in the next 0.8 $s$?

**Solution:**

(a) Momentum conservation gives

$$\begin{aligned} m_Rv_R + m_Lv_L &= 0 \\ (0.5)v_R + (1)(-1.2) &= 0 \\ v_R &= 2.4\ m/s \end{aligned}$$

Thus,

$$\triangle x = v_Rt = (2.4)(0.8) = 1.92\ m$$

(b) Now we have

$$m_R v_R + m_L(v_R - 1.2) = 0$$

which yields

$$\begin{aligned} v_R &= \frac{(1.2)m_L}{m_L + m_R} \\ &= \frac{(1.2)(1)}{1 + 0.5} \\ &= 0.8\ m/s \end{aligned}$$

Consequently, $\triangle x = v_R t = 0.640\ m$.

## IX. Simulation for Two-dimensional Airplane Explosion with Box2D

An Air Force One airplane of mass $M$ is flying above the White House and suddenly it explodes into two big pieces. The bottom piece with mass $0.20M$, has final speed $\vec{v}_{fB} = 500.0\ m/s$.

1. What is the speed of the right piece, with mass $0.40M$?
2. What is the speed of the left piece?

**Solution:**

1. First we need to see whether linear momentum is conserved. We note that
   (a) The airplane and its pieces form a closed system.
   (b) The explosion forces are internal to that system.
   (c) No net external force acts on the system.

   Therefore the linear momentum of the system is conserved. Although the momentum of the system does not change, the momenta of the pieces certainly do.

   To get started, we superimpose an $xy$ coordinate system, with the negative direction of the $x$ axis coinciding with the direction of $\vec{v}_{fL}$. The $x$ axis is at $80^0$ with the direction of $\vec{v}_{fR}$ and $-50^0$ with the direction of $\vec{v}_{fB}$.
   Linear momentum is conserved separately along each axis. Let's use the $y$ axis and write

   $$P_{iy} = P_{fy}$$

   where subscript $i$ refers to the initial value (before the explosion), and subscript $y$ refers to the $y$ component of $\vec{P}_i$ or $\vec{P}_f$.

   The component $P_{iy}$ of the initial linear momentum is zero, because the the explosion occurs when the airplane is steady at one position for a while. Linear momentum is also conserved along the $x$ axis To get an expression for $P_{fy}$, we find the $y$ component of the final linear momentum of each piece, using the $y$-component version ($p_y = mv_y$)

   $$\begin{aligned} p_{fL,y} &= 0.40Mv_{fL,y} 0.40Mv_{fL} \sin\ 80^0 = 0 \\ p_{fR,y} &= 0.40Mv_{fR,y} = 0.40Mv_{fR} \sin\ 80^0 \\ p_{fB,y} &= -0.20Mv_{fB,y} = -0.20Mv_{fB} \sin\ 50^0 \end{aligned}$$

   we can now write

   $$P_{iy} = P_{fy} = p_{fL,y} + p_{fR,y} + p_{fB,y}$$

   Then, with $v_{fB} = 500.0\ m/s$, we have

   $$\begin{aligned} P_{fy} &= p_{fL,y} + p_{fR,y} + p_{fB,y} \\ 0 &= 0 + (0.40M)(v_{fR}) \sin\ 80^0 - 0.20M(500) \sin\ 50^0 \\ v_{fR} &= 194.46548\ m/s \end{aligned}$$

2. Linear momentum is also conserved along the $x$ axis because there is no net external force acting on the coconut and pieces along that axis. Thus we have

$$P_{ix} = P_{fx} \tag{11.41}$$

where $P_{ix} = 0$ because the explosion occurs when the airplane is steady at one position for a while. To get $P_{fx}$, we find the $x$ components of the final momenta, using the fact that the left piece must have a mass of $0.40M$ ($= M - 0.20M - 0.40M$):

$$\begin{aligned}
p_{fL,x} &= 0.40Mv_{fR,x} = 0.40Mv_{fL}\cos\ 180^0 = -0.40Mv_{fL} \\
p_{fR,x} &= 0.40Mv_{fR,x} = 0.40Mv_{fR}\cos\ 80^0 \\
p_{fB,x} &= 0.20Mv_{fB,x} = 0.20Mv_{fB}\cos\ (-50^0) = 0.20Mv_{fB}\cos\ 50^0
\end{aligned}$$

For the conservation of momentum along the $x$ axis can now be written as

$$P_{ix} = P_{fx} = p_{fL,x} + p_{fR,x} + p_{fB,x}$$

Then, with $v_{fB} = 500\ m/s$ and $v_{fR} = 194.46548\ m/s$, we have

$$\begin{aligned}
P_{fx} &= p_{fL,x} + p_{fR,x} + p_{fB,x} \\
0 &= -0.40Mv_{fL} + 0.40Mv_{fR}\cos\ 80^0 + 0.20Mv_{fB}\cos\ 50^0 \\
v_{fL} &= 194.46548\ m/s
\end{aligned}$$

```
#include "test.h"

// This is used to test sensor shapes.
class AirplaneExplosion : public Test
{
	public:

	enum
	{
		e_count = 7
	};

	AirplaneExplosion()
	{
		// Ground body
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
				0.0f));
			ground->CreateFixture(&shape, 0.0f);
		}
```

```
        // Create spaceship
        {
            b2BodyDef bd;
            bd.type = b2_dynamicBody;
            bd.position.Set(15.0f, 25.0f);
            m_ship2 = m_world->CreateBody(&bd);

            b2Vec2 vertices[3];

            vertices[0].Set(-2.0f, 0.0f);
            vertices[1].Set(1.0f, 2.0f);
            vertices[2].Set(0.0f, 4.0f);
            shape_left.Set(vertices, 3);

            vertices[0].Set(2.0f, 0.0f);
            vertices[1].Set(-1.0f, 2.0f);
            vertices[2].Set(0.0f, 4.0f);
            shape_right.Set(vertices, 3);

            vertices[0].Set(2.0f, 0.0f);
            vertices[1].Set(0.0f, 1.3f);
            vertices[2].Set(-2.0f, 0.0f);
            shape_bottom.Set(vertices, 3);

            m_pieceleft = m_ship2->CreateFixture(&shape_left, 2.0f
                );
            m_pieceright = m_ship2->CreateFixture(&shape_right,
                2.0f);
            m_piecebottom = m_ship2->CreateFixture(&shape_bottom,
                2.0f);

            //m_ship2->SetLinearVelocity(1.0f,1.0f);
            m_ship2->SetAngularVelocity(1.0f);

        }

        m_break = false;
        m_broke = false;
    }

    void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
        override
    {
        if (m_broke)
        {
            // The body already broke.
            return;
        }
```

```
        // Should the body break?
        int32 count = contact->GetManifold()->pointCount;

        float maxImpulse = 0.0f;
        for (int32 i = 0; i < count; ++i)
        {
            maxImpulse = b2Max(maxImpulse, impulse->normalImpulses
                [i]);
        }

        if (maxImpulse > 50.0f)
        {
            // Flag the body for breaking.
            m_break = true;
        }
    }

    void Break()
    {
        // Create three bodies from one.

        b2Body* body_spaceship1 = m_pieceleft->GetBody();
        b2Vec2 spaceship_center = body_spaceship1->GetWorldCenter();

        body_spaceship1->DestroyFixture(m_pieceright);
        m_pieceright = NULL;

        body_spaceship1->DestroyFixture(m_piecebottom);
        m_piecebottom = NULL;

        b2BodyDef bd_spaceship;
        bd_spaceship.type = b2_dynamicBody;
        bd_spaceship.position = body_spaceship1->GetPosition();
        bd_spaceship.angle = body_spaceship1->GetAngle();

        b2Body* body_spaceship2 = m_world->CreateBody(&bd_spaceship)
            ;
        m_pieceright = body_spaceship2->CreateFixture(&shape_right,
            1.0f);

        b2Body* body_spaceship3 = m_world->CreateBody(&bd_spaceship)
            ;
        m_piecebottom = body_spaceship3->CreateFixture(&shape_bottom
            , 1.0f);

        // Compute consistent velocities for new bodies based on
        // cached velocity.
```

```
            b2Vec2 spaceship_center1 = body_spaceship1->GetWorldCenter()
                ;
            b2Vec2 spaceship_center2 = body_spaceship2->GetWorldCenter()
                ;
            b2Vec2 spaceship_center3 = body_spaceship3->GetWorldCenter()
                ;

            b2Vec2 spaceship_velocity1 = m_spaceshipvelocity + b2Cross(
                m_spaceshipangularVelocity, spaceship_center1 -
                spaceship_center);
            b2Vec2 spaceship_velocity2 = m_spaceshipvelocity + b2Cross(
                m_spaceshipangularVelocity, spaceship_center2 -
                spaceship_center);
            b2Vec2 spaceship_velocity3 = m_spaceshipvelocity + b2Cross(
                m_spaceshipangularVelocity, spaceship_center3 -
                spaceship_center);

            body_spaceship1->SetAngularVelocity(
                m_spaceshipangularVelocity);
            body_spaceship1->SetLinearVelocity(spaceship_velocity1);

            body_spaceship2->SetAngularVelocity(
                m_spaceshipangularVelocity);
            body_spaceship2->SetLinearVelocity(spaceship_velocity2);

            body_spaceship3->SetAngularVelocity(
                m_spaceshipangularVelocity);
            body_spaceship3->SetLinearVelocity(spaceship_velocity3);
        }

        void Keyboard(int key) override
        {
            switch (key)
            {
                case GLFW_KEY_D:
                m_break = true;
                break;
            }
        }

        void Step(Settings& settings) override
        {
            b2Vec2 v = m_ship2->GetLinearVelocity();
            float angle = m_ship2->GetAngle();
            float ship_vel = 2.1510f;
            b2MassData massData = m_ship2->GetMassData();
            b2Vec2 ship_position = m_ship2->GetPosition();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
```

```
                60 Hertz

        if (m_break)
        {
            Break();
            m_broke = true;
            m_break = false;
        }

        // Cache velocities to improve movement on breakage.
        if (m_broke == false)
        {
            m_ship2->SetLinearVelocity(b2Vec2(-ship_vel*sinf(
                angle), ship_vel*cosf(angle)));

            m_spaceshipvelocity = m_ship2->GetLinearVelocity();
            m_spaceshipangularVelocity = m_ship2->
                GetAngularVelocity();
        }

        g_debugDraw.DrawString(5, m_textLine, "Press D to make the
            plane exlodes");
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Airplane position =
            (%4.1f, %4.1f)", ship_position.x, ship_position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
            .mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
            (%4.1f, %4.1f)", v.x, v.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Total linear velocity=
             %.6f", v);
        m_textLine += m_textIncrement;

        Test::Step(settings);
    }

    static Test* Create()
    {
        return new AirplaneExplosion;
    }

    b2Body* m_ship2;
```

```
        b2Vec2 m_spaceshipvelocity;
        float m_spaceshipangularVelocity;
        b2PolygonShape shape_left;
        b2PolygonShape shape_right;
        b2PolygonShape shape_bottom;
        b2Fixture* m_pieceleft;
        b2Fixture* m_pieceright;
        b2Fixture* m_piecebottom;

        float m_time;
        bool m_broke;
        bool m_break;
};

static int testIndex = RegisterTest("Center of Mass", "Airplane Explosion",
     AirplaneExplosion::Create);
```

**C++ Code 82:** *tests/centerofmass_airplaneexplosion.cpp "Center of Mass Airplane Explosion Box2D"*

Some explanations for the codes:

- We put the **m_ship2->SetLinearVelocity(b2Vec2(-ship_vel*sinf(angle), ship_vel*cosf(angle)))** inside the **if** statement while the airplane is still circling around **m_broke == false** and not yet explodes. So by the power of gravity after the airplane explodes all the pieces will fall on earth.

```
        ...

        void Step(Settings& settings) override
        {
                b2Vec2 v = m_ship2->GetLinearVelocity();
                float angle = m_ship2->GetAngle();
                float ship_vel = 2.1510f;
                b2MassData massData = m_ship2->GetMassData();
                b2Vec2 ship_position = m_ship2->GetPosition();
                m_time += 1.0f / 60.0f; // assuming we are using
                    frequency of 60 Hertz

                if (m_break)
                {
                        Break();
                        m_broke = true;
                        m_break = false;
                }

                // Cache velocities to improve movement on breakage.
                if (m_broke == false)
                {
                        m_ship2->SetLinearVelocity(b2Vec2(-ship_vel*sinf
                            (angle), ship_vel*cosf(angle)));
```

```
                m_spaceshipvelocity = m_ship2->GetLinearVelocity
                    ();
                m_spaceshipangularVelocity = m_ship2->
                    GetAngularVelocity();
        }
        ...
}

...
...
```

- We are also using a conditional if there is a big impulse coming in, or a crash on air happens, then the plane will explodes too.

```
...

void PostSolve(b2Contact* contact, const b2ContactImpulse*
    impulse) override
{
        if (m_broke)
        {
                // The body already broke.
                return;
        }

        // Should the body break?
        int32 count = contact->GetManifold()->pointCount;

        float maxImpulse = 0.0f;
        for (int32 i = 0; i < count; ++i)
        {
                maxImpulse = b2Max(maxImpulse, impulse->
                    normalImpulses[i]);
        }

        if (maxImpulse > 50.0f)
        {
                // Flag the body for breaking.
                m_break = true;
        }
}
...
```

**Figure 11.13:** *The simulation of an airplane that is circling around and then explodes into 3 big pieces (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_airplaneexplosion.cpp****).*

**Figure 11.14:** *The simulation of an airplane that is circling around and then explodes into 3 big pieces (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_airplaneexplosion.cpp****).*

**Figure 11.15:** *The simulation of an airplane that is circling around and then explodes into 3 big pieces (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_airplaneexplosion.cpp****).*

## X. Momentum and Kinetic Energy in Collisions

[DF*] The total linear momentum $\vec{P}$ of the system cannot change because there is no net external force to change it. This is a very powerful rule because it can allow us to determine the results of a collision without knowing the details of the collision (such as how much damage is done).

[DF*] If the total kinetic energy of a system of two colliding bodies happens to be unchanged by the collision, then the kinetic energy of the system is conserved (it is the same before and after the collision). Such a collision is called an elastic collision.

In everyday collisions of common bodies, such as two cars or a ball and a bat, some energy is always transferred from kinetic energy to other forms of energy, such as thermal energy or energy of sound. Thus, the kinetic energy of the system is not conserved. Such a collision is called an inelastic collision.

The inelastic collision of two bodies always involves a loss in the kinetic energy of the system. The greatest loss occurs if the bodies stick together, in which case the collision is called a completely inelastic collision. The collision of a baseball and a bat is inelastic. However, the collision of a wet putty ball and a bat is completely inelastic because the putty sticks to the bat.

[DF*] **One-Dimensional Inelastic Collision**

In one-dimensional inelastic collision, suppose in $x$ axis there are two bodies, Body 1 and Body 2, Body 1 is located on the left of Body 2 and they are moving toward positive $x$ axis.

The velocities before the collision (subscript $i$) and after the collision (subscript $f$) are indicated. The two bodies form our system, which is closed and isolated. We can write the law of conservation of linear momentum for this two-body system as
**The total momentum $\vec{P}_i$ before the collision is equal to the total momentum $\vec{P}_f$ after the collision**.

Which we can symbolize as

$$\vec{p}_{1i} + \vec{P}_{2i} = \vec{p}_{1f} + \vec{p}_{2f} \tag{11.42}$$

Because the motion is one-dimensional, we can drop the overhead arrows for vectors and use only components along the axis, indicating direction with a sign. Thus, from $p = mv$,

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f} \tag{11.43}$$

**Figure 11.16:** *Bodies 1 and 2 move along an x axis, before and after they have an inelastic collision.*

[DF*] **One-Dimensional Completely Inelastic Collision**
In this case, when we have two bodies that are having collision they will stick together, since it is a completely inelastic collision.

The body with mass $m_2$ happens to be initially at rest ($v_{2i} = 0$). We can refer to that body as the target and to the incoming body as the projectile. After the collision, the stuck-together bodies move with velocity $V$. For this situation, we will have

$$m_1 v_{1i} = (m_1 + m_2)V \tag{11.44}$$

or

$$V = \frac{m_1}{m_1 + m_2} v_{1i} \tag{11.45}$$

**Figure 11.17:** *A completely inelastic collision between two bodies. Before the collision, the body with mass $m_2$ is at rest and the body with mass $m_1$ moves directly toward it. After the collision, the stuck-together bodies move with the same velocity $\vec{V}$.*

[DF*] **Velocity of the Center of Mass**
In a closed, isolated system, the velocity $\vec{v}_{com}$ of the center of mass of the system cannot be changed by a collision because, with the system isolated, there is no net external force to change it. To get an expression for $\vec{v}_{com}$, let us return to the two-body system and one-dimensional collision ($\vec{P} = M\vec{v}_{com}$), we can relate $\vec{v}_{com}$ to the total linear momentum $\vec{P}$ of

that two-body system by writing

$$\vec{P} = M\vec{v}_{com} = (m_1 + m_2)\vec{v}_{com} \tag{11.46}$$

The total linear momentum $\vec{P}$ is conserved during the collision; so it is given by either $\vec{p}_{1i} + \vec{p}_{2i}$ or $\vec{p}_{1f} + \vec{p}_{2f}$, thus

$$\vec{P} = \vec{p}_{1i} + \vec{p}_{2i} \tag{11.47}$$

Substituting this expression for $\vec{P}$ and solving for $\vec{v}_{com}$ give us

$$\vec{v}_{com} = \frac{\vec{P}}{m_1 + m_2} = \frac{\vec{p}_{1i} + \vec{p}_{2i}}{m_1 + m_2} \tag{11.48}$$

The right side of this equation is a constant, and $\vec{v}_{com}$ has the same constant value before and after the collision.

[DF*] **Example:**

The ballistic pendulum was used to measure the speeds of bullets before electronic timing devices were developed. The version consists of a large block of wood of mass $M = 5.4\ kg$, hanging from two long cords. A bullet of mass $m = 9.5\ g$ is fired into the block, coming quickly to rest. The $block + bullet$ then swing upward, their center of mass rising a vertical distance $h = 6.3\ cm$ before the pendulum comes momentarily to rest at the end of its arc. What is the speed of the bullet just prior to the collision?

**Solution:**

We can see that the bullet's speed $v$ must determine the rise height $h$. However, we cannot use the conservation of mechanical energy to relate these two quantities because surely energy is transferred from mechanical energy to toher forms (such as thermal energy and energy to break apart the wood) as the bullet penetrates the block. Nevertheless, we can split this complicated motion into two steps that we can separately analyze:

1. The bullet-block collision.
2. The bullet-block rise, during which mechanical energy is conserved.

Because the collision within the bullet-block system is so brief, we can make two important assumptions:

1. During the collision, the gravitational force on the block and the force on the block from the cords are still balanced. Thus, during the collision, the net external impulse on the bullet-block system is zero. Therefore, the system is isolated and its total linear momentum is conserved. The total momentum before the collision is equal to the total momentum after the collision.
2. The collision is one-dimensional in the sense that the direction of the bullet and block just after the collision is in the bullet's original direction of motion.

Because the collision is one-dimensional, the block is initially at rest, and the bullet sticks in the block, we have

$$V = \frac{m}{m + M} v$$

As the bullet and block now swing up together, the mechanical energy of the bullet-block-Earth system is conserved. The mechanical energy at bottom equal to the mechanical energy at top. (This mechanical energy is not changed by the force of the cords on the block, because that force is always directed perpendicular to the block's direction of travel.)

Let's take the block's initial level as our reference level of zero gravitational potential energy. Then conservation of mechanical energy means that the system's kinetic energy at the start of the swing must equal its gravitational potential energy at the highest point of the swing. Because the speed of the bullet and block at the start of the swimng is the speed $V$ immediately after the collision, we may write the conservation as

$$\frac{1}{2}(m+M)V^2 = (m+M)gh$$

Substituting for $V$ leads to

$$\begin{aligned} v &= \frac{m+M}{m}\sqrt{2gh} \\ &= \left(\frac{0.0095+5.4}{0.0095}\right)\sqrt{(2)(9.8)(0.063)} \\ &= 630\ m/s \end{aligned}$$

The ballistic pendulum is a kind of "transformer," exchanging the high speed of a light object (the bullet) for the low-and thus more easily measurable-speed of a massive object (the block).

[DF*] **Example:**
A 3.5 $g$ bullet is fired horizontally at two blocks at rest on a frictionles table. The bullet passes through block 1 (mass 1.2 $kg$) and embeds itself in block 2 (mass 1.8 $kg$). The blocks end up with speeds $v_1 = 0.63\ m/s$ and $v_2 = 1.4\ m/s$. Neglecting the material removed from block 1 by the bullet, find the speed of the bullet as it leaves and enters block 1.

**Solution:**
In solving this problem, our $+x$ direction is to the right (so all velocities are positive-valued).

We apply momentum conservation to relate the situation just before the bullet strikes the second block to the situation where the bullet is embedded within the block.

$$\begin{aligned} (0.0035)v &= (1.8035)(1.4) \\ v &= 721\ m/s \end{aligned}$$

We apply momentum conservation to relate the situation just before the bullet strikes the first block to the instant it has passed through it (having speed $v$)

$$\begin{aligned} (0.0035)v_0 &= (1.2)(0.630) + (0.0035)(721) \\ v_0 &= 937\ m/s \end{aligned}$$

[DF*] **Example:**
A ball of mass $m = 60\ g$ is shot with speed $v_i = 22\ m/s$ into barrel of a spring gun of mass $M = 240\ g$ initially at rest on a frictionless surface. The ball sticks in the barrel a tthe point of maximum compression of the spring. Assume that the increase in thermal energy due to friction between the ball and the barrel is negligible.

(a) What is the speed of the spring gun after the ball stops in the barrel?

(b) What fraction of the kinetic energy of the ball is stored in the spring?

**Solution:**

(a) Let $v$ be the final velocity of the ball-gun system. Since the total momentum of the system is conserved

$$mv_i = (m + M)v$$

Therefore

$$\begin{aligned} v &= \frac{mv_i}{m + M} \\ &= \frac{(60)(22)}{60 + 240} \\ &= 4.4\ m/s \end{aligned}$$

(b) The initial kinetic energy is $K_i = \frac{1}{2}mv_i^2$, and the final kinetic energy is

$$\begin{aligned} K_f &= \frac{1}{2}(m + M)v_f^2 \\ &= \frac{1}{2}\frac{m^2 v_i^2}{m + M} \end{aligned}$$

[DF*] **Example:**

Block 2 (mass 1 $kg$) is at rest on a frictionless surface and touching the end of an unstretched spring of spring constant 200 $N/m$. The other end of the spring is fixed to the wall. Block 1 (mass 2 $kg$), traveling at speed $v_1 = 4\ m/s$, collides with block 2, and the two blocks stick together. When the blocks momentarily stop, by what distance is the spring compressed?

**Solution:**
We think of this as having two parts: the first is the collision itself, where the blocks "join" so quickly that the 1.0 $kg$ block has not had time to move through any distance yet, and then the subsequent motion of the 3.0 $kg$ system as it compresses the spring to the maximum amount $x_m$. The first part involves momentum conservation (with $+x$ rightward):

$$\begin{aligned} m_1 v_1 &= (m_1 + m_2)v \\ (2)(4) &= 3\vec{v} \\ \vec{v} &= 2.7\ m/s \end{aligned}$$

The second part involves mechanical energy conservation:

$$\begin{aligned} \frac{1}{2}mv^2 &= \frac{1}{2}kx_m^2 \\ \frac{1}{2}(3)(2.7) &= \frac{1}{2}(200)x_m^2 \\ x_m &= 0.33\ m \end{aligned}$$

## XI. Simulation for Bullet Shot Upward with Box2D

A 10 $g$ bullet moving directly upward at 1000 $m/s$ strikes and passes through the venter of mass of a 5 $kg$ block initially at rest. The bullet emerges from the block moving directly upward at 400 $m/s$. To what maximum height does the block then rise above its initial position?

**Solution:**

We think of this as having two parts: the first is the collision itself - where the bullet passes through the block so quickly that the block has not had time to move through any distance yet - and then the subsequent "leap" of the block into the air (up to height $h$ measured from its initial position). The first part involves momentum conservation (with $+y$ upward):

$$\begin{aligned} p_i &= p_f \\ m_2\vec{v}_i &= m_1\vec{v}_{f1} + m_2\vec{v}_{f2} \\ (0.01)(1000) &= (5)(\vec{v}_{f1}) + (0.01)(400) \\ \vec{v}_{f1} &= 1.2\ m/s \end{aligned}$$

The second part involves either the free-fall equations (since we are ignoring air friction) or simple energy conservation. Choosing the latter approach, we have

$$\begin{aligned} \frac{1}{2}m_1 v_{f1}^2 &= m_1 g h \\ \frac{1}{2}(5)(1.2) &= (5)(9.8)h \\ h &= 0.073\ m \end{aligned}$$

the box does not rise too high, because not only it is heavier than the bullet, if you ever watch action movie, a bullet from a handgun cannot send people fly away, two bullets from shotgun that is heavier is capable to fly zombies or people away, but not for a small bullet of 10 $g$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class BulletShotUpward : public Test
{
	public:
	BulletShotUpward()
	{
		b2Body* ground = NULL;
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
```

```
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f
                , 46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2(4.5f,
                12.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(5.5f, 12.0f), b2Vec2(46.0f,
                12.0f));
            ground->CreateFixture(&shape, 0.0f);
        }

        // Create the bullet
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.3f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.0f;
```

```
        ballFixtureDef.density = 0.37f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(5.0f, 0.5f);
        // ballBodyDef.angularDamping = 0.2f;

        m_ball = m_world->CreateBody(&ballBodyDef);
        ballFixture = m_ball->CreateFixture(&ballFixtureDef);

        // Create the box as the movable object with friction 0
        b2PolygonShape boxShape;
        boxShape.SetAsBox(1.5f, 1.0f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 5.0f/6.0f; // this will affect the
            box mass
        boxFixtureDef.friction = 0.0f;
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(5.0f, 12.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
            ;


        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_ball;

    b2Fixture* ballFixture;
    float m_bulletvelocity;
    float m_time;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_S:
            m_ball->SetLinearVelocity(b2Vec2(0.0f, 1000.0f));
```

```
                        break;
                        case GLFW_KEY_A:
                        m_ball->SetLinearVelocity(b2Vec2(0.0f, 500.0f));
                        break;
                }
        }
        void UpdateUI() override
        {
                ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
                ImGui::SetNextWindowSize(ImVec2(420.0f, 150.0f));
                ImGui::Begin("Bullet velocity controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

                if (ImGui::SliderFloat("Bullet", &m_bulletvelocity, 0.0f,
                    1000.0f, "%.0f"))
                {
                        m_ball->SetLinearVelocity(b2Vec2(0.0f,
                            m_bulletvelocity));
                }

                ImGui::End();
        }
        void Step(Settings& settings) override
        {
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz
                b2MassData massData = m_box->GetMassData();
                b2MassData massData2 = m_ball->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                b2Vec2 velocity = m_box->GetLinearVelocity();
                b2Vec2 positionbullet = m_ball->GetPosition();
                b2Vec2 velocitybullet = m_ball->GetLinearVelocity();

                g_debugDraw.DrawString(5, m_textLine, "Press S to shoot the
                    bullet with velocity of 1000 m/s");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Press A to shoot the
                    bullet with velocity of 500 m/s");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
                     %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
                     %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Bullet position =
                    (%4.1f, %4.1f)", positionbullet.x, positionbullet.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Bullet velocity =
                    (%4.1f, %4.1f)", velocitybullet.x, velocitybullet.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Bullet Mass = %.6f",
                    massData2.mass);
                m_textLine += m_textIncrement;
                Test::Step(settings);

                g_camera.m_zoom = 1.4f; // zoom out camera

                printf("%4.2f %4.2f %4.2f %4.2f \n", position.x, position.y,
                    velocity.x, velocity.y);
        }
        static Test* Create()
        {
                return new BulletShotUpward;
        }

};

static int testIndex = RegisterTest("Center of Mass", "Bullet Shot Upward",
     BulletShotUpward::Create);
```

**C++ Code 83:** *tests/centerofmass_bulletshotupward.cpp "Center of Mass Bullet Shot Upward Box2D"*

**Figure 11.18:** *The simulation of a bullet shot upward at* 1000 $m/s$ *toward a block of* 5 $kg$*, we are using a ball as a replacement for a bullet shape (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_bulletshotupward.cpp****).*

## XII. Elastic Collisions in One Dimension

[DF*] Everyday collisions are inelastic but we can approximate some of them as being elastic; that is, we can approximate that the total kinetic energy of the colliding bodies is conserved and is not transferred to other forms of energy:
**The total kinetic energy before the collision is equal to the total kinetic energy after the collision.**

This means:
**In an elastic collision, the kinetic energy of each colliding body may change, but the total kinetic energy of the system does not change.**

[DF*] The collision of a cue ball with an object ball in a game of pool can be approximated as being an elastic collision. If the collision is head-on (the cue ball heads directly toward the object ball), the kinetic energy of the cue ball can be transferred almost entirely to the object ball. (Still, the collision transfers some of the energy to the sound you hear.)

[DF*] **Stationary Target**
A projectile body of mass $m_1$ and initial velocity $v_1$ moves toward a target body mass $m_2$ that is initially at rest ($v_{21} = 0$). Let's assume that this two-body system is closed and isolated. Then the net linear momentum of the system is conserved, and we can write the conservation as

$$m_1 v_{1i} = m_1 v_{1f} + m_2 v_{2f} \tag{11.49}$$

If the collision is also elastic, then the total kinetic energy is conserved and we can write that

conservation as

$$\frac{1}{2}m_1 v_{1i}^2 = \frac{1}{2}m_1 v_{1f}^2 + \frac{1}{2}m_2 v_{1f}^2 \tag{11.50}$$

In each of these equations, the subscript $i$ identifies the initial velocities and the subscript $f$ the final velocities of the bodies. If we know the masses of the bodies and if we also know $v_{1i}$, the initial velocity of body 1, the only unknown quantities are $v_{1f}$ and $v_{2f}$, the final velocities of two bodies. With two equations at our disposal, we should be able to find these two unknowns. To find the two unknowns we will solve these two equations

**Figure 11.19:** *Body 1 moves along an x axis before having an elastic collision with body 2, which is initially at rest. Both bodies move along that axis after the collision.*

$$m_1(v_{1i} - v_{1f}) = m_2 v_{2f} \tag{11.51}$$

and

$$m_1(v_{1i} - v_{1f})(v_{1i} + v_{1f}) = m_2 v_{2f}^2 \tag{11.52}$$

Then we obtain

$$v_{1f} = \frac{m_1 - m_2}{m_1 + m_2} v_{1i} \tag{11.53}$$

$$v_{2f} = \frac{2m_1}{m_1 + m_2} v_{1i} \tag{11.54}$$

Note that $v_{2f}$ is always positive (the initially stationary target body with mass $m_2$ always moves forward).

We see that $v_{1f}$ may be of either sign (the projectile body with mass $m_1$ moves forward if $m_1 > m_2$, but rebounds if $m_1 < m_2$).

[DF*] Let us look at a few special situations.

1. **Equal masses**
   If $m_1 = m_2$, then
   $$v_{1f} = 0, \quad v_{2f} = v_{1i}$$
   which we might call a pool player's result. It predicts that after a head-on collision of bodies with equal masses, body 1 (initially moving) stops dead in its tracks and body 2 (initially at rest) takes off with the initial speed of body 1. In head-on collisions, bodies of equal mass simply exchange velocities. This, is true even if body 2 is not initially at rest.

2. **A massive target**
   A massive target means that $m_2 \gg m_1$. For example, we might fire a golf ball at a stationary cannonball. We will have
   $$\begin{aligned} v_{1f} &\approx -v_{1i} \\ v_{2f} &\approx \left(\frac{2m_1}{m_2}\right) v_{1i} \end{aligned} \tag{11.55}$$

This tells us that body 1 (the golf ball) simply bounces back along its incoming path, its speed essentially unchanged. Initially stationary body 2 (the cannonball) moves forward at a low speed, because the quantity in parentheses is much less than unity. All this is what we should expect.

3. **A massive projectile**
   This is the opposite case; that is $m_1 \gg m_2$. This time, we fire a cannonball at a stationary golf ball. We will have

$$\begin{aligned} v_{1f} &\approx v_{1i} \\ v_{2f} &\approx 2v_{v1i} \end{aligned} \tag{11.56}$$

   The equations tell us that body 1 (the cannonball) simply keeps on going scarcely slowed by the collision. Body 2 (the golf ball) charges ahead at twice the speed of the cannonball. Why twice the speed? Recall the collision in number (2), in which the velocity of the incident light body (the golf ball) changed from $+v$ to $-v$, a velocity change of $2v$. The same change in velocity (but now from zero to $2v$) occurs in this example also.

[DF*] **Moving Target**
For an elastic collision with both moving target move toward each other, the conservation of linear momentum is written as

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f} \tag{11.57}$$

and the conservation of kinetic energy is written as

$$\frac{1}{2} m_1 v_{1i}^2 + \frac{1}{2} m_2 v_{2i}^2 = \frac{1}{2} m_1 v_{1f}^2 + \frac{1}{2} m_2 v_{2f}^2 \tag{11.58}$$

To solve these simultaneous equations for $v_{1f}$ and $v_{2f}$, we will use these equations

$$\begin{aligned} m_1(v_{1i} - v_{1f}) &= -m_2(v_{2i} - v_{2f}) \\ m_1(v_{1i} - v_{1f})(v_{1i} + v_{1f}) &= -m_2(v_{2i} - v_{2f})(v_{2i} + v_{2f}) \end{aligned} \tag{11.59}$$

We will obtain

$$\begin{aligned} v_{1f} &= \frac{m_1 - m_2}{m_1 + m_2} v_{1i} + \frac{2m_2}{m_1 + m_2} v_{2i} \\ v_{2f} &= \frac{2m_1}{m_1 + m_2} v_{1i} + \frac{m_2 - m_1}{m_1 + m_2} v_{2i} \end{aligned} \tag{11.60}$$

[DF*] **Example:**
Block 1 approaches a line of two stationary blocks with a velocity of $v_{1i} = 10\ m/s$. It collides with block 2, which then collides with block 3, which has mass $m_3 = 6.0\ kg$. After the second collision, block 2 is again stationary and block 3 has velocity $v_{3f} = 5.0\ m/s$. Assume that the collisions are elastic. What are the masses of blocks 1 and 2? What is the final velocity of $v_{1f}$ of block 1?

**Solution:**

Because we assume that the collisions are elastic, we are to conserve mechanical energy (thus energy loses to sound, heating, and oscillations of the blocks are negligible). Because no

external horizontal force acts on the blocks, we are to conserve linear momentum along the $x$ axis.

If we start with the first collision, we have too many unknowns to make any progress; we do not know the masses or the final velocities of the blocks. So, let's start with the second collision in which block 2 stops because of its collision with block 3. Applying to this collision, we have

$$v_{2f} = \frac{m_2 - m_3}{m_2 + m_3} v_{2i}$$

where $v_{2i}$ is the velocity of block 2 just before the collision and $v_{2f}$ is the velocity just afterward. Susbtituting $v_{2f} = 0$ (block 2 stops) and then $m_3 = 6.0\ kg$ gives us

$$m_2 = m_3 = 6.0\ kg$$

With similar notation changes, we can rewrite for the second collision as

$$v_{3f} = \frac{2m_2}{m_2 + m_3} v_{2i}$$

where $v_{3f}$ is the final velocity of block 3. Substituting $m_2 = m_3$ and the given $v_{3f} = 5.0\ m/s$, we find

$$v_{2i} = v_{3f} = 5.0\ m/s$$

Next, let's reconsider the first collision, but we have to be careful with the notation for block 2: its velocity $v_{2f}$ just after the first collision is the same as its velocity $v_{2i}$ just before the second collision. We have

$$\begin{aligned} v_{2f} &= \frac{2m_1}{m_1 + m_2} v_{1i} \\ 5 &= \frac{2m_1}{m_1 + 6} 10 \\ \frac{1}{2} m_1 + 3 &= 2m_1 \\ m_1 + 6 &= 4m_1 \\ m_1 &= 2.0\ kg \end{aligned}$$

Finally, applying to the first collision with this result and the given $v_{1i}$, we write

$$\begin{aligned} v_{1f} &= \frac{m_1 - m_2}{m_1 + m_2} v_{1i} \\ &= \frac{2 - 6}{2 + 6} 10 \\ &= -5.0\ m/s \end{aligned}$$

[DF*] **Example:**
Block 1 of mass $m_1$ slides along an $x$ axis on a frictionless floor at speed 4 $m/s$. Then it undergoes a one-dimensional elastic collision with stationary block 2 of mass $m_2 = 2m_1$. Next, block 2 undergoes a one-dimensional elastic collision with stationary block 3 of mass $m_3 = 2m_2$.

(a) What is the speed of block 3?

(b) Is the speed of block 3 greater than, less than, or the same as the initial values for block 1?

(c) Is the kinetic energy of block 3 greater than, less than, or the same as the initial values for block 1?

(d) Is the momentum of block 3 greater than, less than, or the same as the initial values for block 1?

**Solution:**
After the first collision we have

$$\begin{aligned} v_{1f} &= \frac{m_1 - m_2}{m_1 + m_2} v_{1i} \\ &= \frac{m_1 - 2m_1}{m_1 + 2m_1} v_{1i} \\ &= -\frac{1}{3} v_{1i} \\ v_{2f} &= \frac{2m_1}{m_1 + m_2} v_{1i} \\ &= \frac{2m_1}{m_1 + 2m_1} v_{1i} \\ &= \frac{2}{3} v_{1i} \end{aligned}$$

After the second collision, the velocities are

$$\begin{aligned} v_{2ff} &= \frac{m_2 - m_3}{m_2 + m_3} v_{2f} \\ &= \frac{-m_2}{3m_2} \frac{2}{3} v_{1i} \\ &= -\frac{2}{9} v_{1i} \\ v_{3ff} &= \frac{2m_2}{m_2 + m_3} v_{2f} \\ &= \frac{2m_2}{3m_2} \frac{2}{3} v_{1i} \\ &= \frac{4}{9} v_{1i} \end{aligned}$$

(a) Setting $v_{1i} = 4\ m/s$, we find $v_{3ff} \approx 1.78\ m/s$.

(b) We see that $v_{3ff}$ is less than $v_{1i}$.

(c) The final kinetic energy of block 3 (expressed in terms of the initial kinetic energy of block 1) is

$$K_{3ff} = \frac{1}{2} m_3 v_3^2 = \frac{1}{2}(4m_1)\left(\frac{4}{9}\right)^2 v_{1i}^2 = \frac{64}{81} K_{1i}$$

We see that this is less than $K_{1i}$.

(d) The final momentum of block 3 is

$$p_{3ff} = m_3 v_{3ff} = (4m_1)\left(\frac{16}{9}\right) v_1$$

We see that this is greater than $m_1 v_1$

**Figure 11.20:** *The simulation of a speed deamplifier with three boxes that have increasing masses from box* 1 *to box* 2 *to box* 3*. (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_speeddeamplifier.cpp****).*

[DF*] **Example:**
Block 1 of mass $m_1$ slides along an $x$ axis on a frictionless floor with a speed of $v_{1i} = 4\ m/s$. Then it undergoes a one-dimensional elastic collision with stationary block 2 of mass $m_2 = 0.5m_1$. Next, block 2 undergoes a one-dimensional elastic collision with stationary block 3 of mass $m_3 = 0.5m_2$.

(a) What is the speed of block 3?

(b) Is the speed of block 3 greater than, less than, or the same as the initial values for block 1?

(c) Is the kinetic energy of block 3 greater than, less than, or the same as the initial values for block 1?

(d) Is the momentum of block 3 greater than, less than, or the same as the initial values for block 1?

**Solution:**

[DF*] To obtain the speed for the block 2 and block 3 we will use these equations

$$\begin{aligned} v_2 &= \frac{2m_1}{m_1 + m_2} v_{1i} \\ &= \frac{2m_1}{1.5m_1}(4) \\ &= \frac{16}{9}\ m/s \end{aligned}$$

$$\begin{aligned} v_3 &= \frac{2m_2}{m_2+m_3}v_2 \\ &= \frac{2m_2}{1.5m_2}\left(\frac{16}{3}\right) \\ &= \frac{64}{9}\ m/s \end{aligned}$$

[DF*] Clearly, the speed of block 3 is greater than the (initial) speed of block 1.

[DF*] The kinetic energy of block 3 is

$$\begin{aligned} K_{3f} &= \frac{1}{2}m_3v_3^2 \\ &= \left(\frac{1}{2}\right)^3 m_1\left(\frac{16}{9}\right)^2 v_{1i}^2 \\ &= \frac{64}{81}K_{1i} \end{aligned}$$

since $v_2 = \frac{4}{3}v_{1i}$ and $v_3 = \frac{4}{3}v_{1i}$. We see the kinetic energy of block 3 is less than the (initial) $K$ of block 1. In the final situation, the initial $K$ is being shared among the three blocks (which are all in motion), so this is not a surprising conclusion.

[DF*] The momentum of block 3 is

$$\begin{aligned} p_{3f} &= m_3v_3 \\ &= \left(\frac{1}{2}\right)^2 m_1\left(\frac{16}{9}\right)v_{1i} \\ &= \frac{4}{9}p_{1i} \end{aligned}$$

and is therefore less than the initial momentum of block 1.

[DF*] **Example:**

An electron undergoes a one-dimensional elastic collision with an initially stationary hydrogen atom. What percentage of the electron's initial kinetic energy is transferred to kinetic energy of the hydrogen atom? (the mass of the hydrogen atom is 1840 times the mass of the electron.)

**Solution:**

We use $m_1$ for the mass of the electron and $m_2 = 1840m_1$ for the mass of the hydrogen atom. Thus,

$$v_{2f} = \frac{2m_1}{m_1+1840m_1}v_{1i} = \frac{2}{1841}v_{1i}$$

we compute the final kinetic energy of the hydrogen atom:

$$\begin{aligned} K_{2f} &= \frac{1}{2}m_2v_{2f}^2 \\ &= \frac{1}{2}1840m_1\left(\frac{2}{1841}v_{1i}\right)^2 \\ &= \frac{(1840)(4)}{1841^2}\frac{1}{2}m_1v_{1i}^2 \\ &= 0.00217\ \frac{1}{2}m_1v_{1i}^2 \end{aligned}$$

**Figure 11.21:** *The simulation of a speed amplifier with three boxes that have decreasing masses from box* 1 *to box* 2 *to box* 3*. (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_speeddeamplifier.cpp****).*

so we find the fraction, the percentage of electron's initial kinetic energy that is transferred to kinetic energy of the hydrogen atom is 0.217%

# XIII. Simulation for Elastic Collision of a Steel Ball Pendulum with Box2D

A steel ball of mass 0.5 $kg$ is fastened to a cord that is 70 $cm$ long and fixed at the far end. The ball is then released when the cord is horizontal. At the bottom of its path, the ball strikes a 2.5 $kg$ steel block initially at rest on a frictionless surface. The collision is elastic.

(a) Find the speed of the ball after the collision.

(b) Find the speed of the block after the collision.

**Solution:**

(a) First, we find the speed $v$ of the ball of mass $m_1$ right before the collision (just as it reaches its lowest point of swing). Mechanical energy conservation (with $h = 0.7\ m$) leads to

$$\begin{aligned} m_1 gh &= \frac{1}{2} m_1 v^2 \\ v &= \sqrt{2gh} \\ &= 3.7\ m/s \end{aligned}$$

We now treat the elastic collision using this formula

$$\begin{aligned} v_{1f} &= \frac{m_1 - m_2}{m_1 + m_2} v \\ &= \frac{0.5 - 2.5}{0.5 + 2.5}(3.7) \\ &= -2.47\ m/s \end{aligned}$$

which means the final speed of the ball is 2.47 $m/s$, the negative sign means that the ball moving toward $-x$ axis after the collision.

(b) Finally, to find the final speed o the block:

$$\begin{aligned} v_{2f} &= \frac{2m_1}{m_1 + m_2} v \\ &= \frac{2(0.5)}{0.5 + 2.5}(3.7) \\ &= 1.23\ m/s \end{aligned}$$

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class ElasticCollisionSteelBall : public Test
{
        public:

        ElasticCollisionSteelBall()
```

```
        {
                b2Body* b1;
                {
                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-9990.0f, 24.2f), b2Vec2
                            (9990.0f, 24.2f));

                        b2BodyDef bd;
                        b1 = m_world->CreateBody(&bd);
                        b1->CreateFixture(&shape, 0.0f);
                }


                b2Body* b2; // the hanging bar for the pendulum
                {
                        b2PolygonShape shape;
                        shape.SetAsBox(7.25f, 0.25f);

                        b2BodyDef bd;
                        bd.position.Set(0.0f, 37.0f);
                        b2 = m_world->CreateBody(&bd);
                        b2->CreateFixture(&shape, 0.0f);
                }

                // Create the steel block as the movable object with friction
                     0
                b2PolygonShape boxShape;
                boxShape.SetAsBox(0.5f, 0.5f);

                b2FixtureDef boxFixtureDef;
                boxFixtureDef.restitution = 0.75f;
                boxFixtureDef.density = 2.5f; // this will affect the block
                    mass
                boxFixtureDef.friction = 0.0f;
                boxFixtureDef.shape = &boxShape;

                b2BodyDef boxBodyDef;
                boxBodyDef.type = b2_dynamicBody;
                boxBodyDef.position.Set(0.0f, 24.5f);

                m_box = m_world->CreateBody(&boxBodyDef);
                b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
                    ;

                // Create the pendulum steel ball
                b2RevoluteJointDef jd;
                b2Vec2 anchor;
```

```
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
        ballShape.m_radius = 0.5f;

        b2FixtureDef ballFixtureDef;
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 0.64f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.0f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(-11.0f, 36.0f);
        // ballBodyDef.angularDamping = 0.2f;

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);

        // Create the anchor and connect it to the ball
        anchor.Set(0.0f, 36.0f); // x and y axis position for the
            Pendulum anchor
        jd.Initialize(b2, m_ball, anchor);
        //jd.collideConnected = true;
        m_world->CreateJoint(&jd); // Create the Pendulum anchor
    }
    b2Body* m_box;

    float m_time;

    void Step(Settings& settings) override
    {
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
             60 Hertz
        b2MassData massDataBlock = m_box->GetMassData();
        b2Vec2 positionBlock = m_box->GetPosition();
        b2Vec2 v_block = m_box->GetLinearVelocity();

        b2Vec2 v = m_ball->GetLinearVelocity();
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
        b2Vec2 position = m_ball->GetPosition();

        float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
            massData.I * omega * omega;
```

```
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Steel Ball position, x
                     = (%4.2f, %4.2f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Steel Ball velocity, v
                     = (%4.2f, %4.2f)", v.x, v.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Steel Ball Linear
                    velocity = %.6f", v);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Stell Ball Mass = %.6f
                    ", massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Steel Ball Kinetic
                    energy = %.6f", ke);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Steel Ball Angle (in
                    degrees) = %.6f", angle*RADTODEG);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "Steel Block position =
                     (%4.2f, %4.2f)", positionBlock.x, positionBlock.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Steel Block velocity =
                     (%4.2f, %4.2f)", v_block.x, v_block.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Stell Block Mass = %.6
                    f", massDataBlock.mass);
                m_textLine += m_textIncrement;

                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
                    position.x, position.y, angle*RADTODEG, positionBlock.x,
                    positionBlock.y, v_block.x, v_block.y);
                //std::ofstream MyFile("/root/output.txt"); ;
                //MyFile << angle << " " << position.x << " " << position.y;

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new ElasticCollisionSteelBall;
```

```
        }

        b2Body* m_ball;
    };

    static int testIndex = RegisterTest("Center of Mass", "Elastic Collision
        Steel Ball", ElasticCollisionSteelBall::Create);
```

**C++ Code 84:** *tests/centerofmass_elasticcollisionsteelball.cpp "Center of Mass Elastic Collision Steel Ball on Pendulum Box2D"*

**Figure 11.22:** *The simulation steel ball that is fastened to a cord, put at angle of $-90^0$, release to swing then hit a steel block on a frictionless surface (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_elasticcollisionsteelball.cpp****).*

**Figure 11.23:** *The simulation steel ball that is fastened to a cord, put at angle of $-90^0$, release to swing then hit a steel block on a frictionless surface (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_elasticcollisionsteelball.cpp****).*

# XIV. Simulation for Elastic Collision of Baseball and Basketball with Box2D

A small ball of mass $m$ is aligned above a larger ball of mass $M = 0.63\ kg$ (with a slight separation, as with the baseball and basketball), and the two are dropped simultaneously from a height of $h = 1.8\ m$. (Assume the radius of each ball is negligible relative to $h$.)

(a) If the larger ball rebounds elastically from the floor and then the small ball rebounds elastically from the larger ball, what value of $m$ results in the larger ball stopping when it collides with the small ball?

(b) What height does the small ball then reach?

**Solution:**

(a) We use conservation of mechanical energy to find the speed of either ball after it has fallen a distance $h$. The initial kinetic energy is zero, the initial gravitational potential energy is $Mgh$, the final kinetic energy is $\frac{1}{2}Mv^2$, and the final potential energy is zero.

Thus

$$\begin{aligned} Mgh &= \frac{1}{2}Mv^2 \\ v &= \sqrt{2gh} \end{aligned}$$

The collision of the ball of $M$ with the floor is an elastic collision of a light object with a stationary massive object. The velocity of the light object reverses direction without change in magnitude. After the collisions, the ball is traveling upward with a speed of $\sqrt{2gh}$.

The ball of mass $m$ is traveling downward with the same speed. We will find the expression for the velocity of the ball of mass $M$ after the collision:

$$\begin{aligned} v_{Mf} &= \frac{M-m}{M+m}v_{Mi} + \frac{2m}{M+m}v_{mi} \\ &= \frac{M-m}{M+m}\sqrt{2gh} - \frac{2m}{M+m}\sqrt{2gh} \\ &= \frac{M-3m}{M+m}\sqrt{2gh} \end{aligned}$$

For this to be zero, $m = \frac{M}{3}$. With $M = 0.63\ kg$, we have $m = 0.21\ kg$.

(b) We use the same equation to find the velocity of the ball of mass $m$ after the collision:

$$\begin{aligned} v_{mf} &= -\frac{m-M}{M+m}\sqrt{2gh} + \frac{2M}{M+m}\sqrt{2gh} \\ &= \frac{3M-m}{M+m}\sqrt{2gh} \\ &= \frac{9m-m}{3m+m}\sqrt{2gh} \\ &= 2\sqrt{2gh} \end{aligned}$$

We next use conservation of mechanical energy to find the height $h'$ to which the ball rises. The initial kinetic energy is $\frac{1}{2}mv_{mf}^2$, the initial potential energy is zero, the final kinetic energy is zero, and the final potential energy is $mgh'$. Thus,

$$\begin{aligned}\frac{1}{2}mv_{mf}^2 &= mgh' \\ h' &= \frac{v_{mf}^2}{2g} \\ &= 4h\end{aligned}$$

with $h = 1.8\ m$, we have $h' = 7.2\ m$.

```
#include "test.h"
#include <iostream>

class BaseballBasketball: public Test
{
	public:

	BaseballBasketball()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		b2Timer timer;
		// Perimeter Ground body
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shapeGround;
			shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
			    (30.0f, 0.0f));
			ground->CreateFixture(&shapeGround, 0.0f);

			b2EdgeShape shapeTop;
			shapeTop.SetTwoSided(b2Vec2(-40.0f, 60.0f), b2Vec2
			    (30.0f, 60.0f));
			ground->CreateFixture(&shapeTop, 0.0f);

			b2EdgeShape shapeLeft;
			shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
			    (-40.0f, 60.0f));
			ground->CreateFixture(&shapeLeft, 0.0f);

			b2EdgeShape shapeRight;
			shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
			    (30.0f, 60.0f));
			ground->CreateFixture(&shapeRight, 0.0f);
```

```
            }

            // Create the basketball
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 1.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.75f; // the bounciness
            ballFixtureDef.density = 0.09f; // this will affect the ball
                mass
            ballFixtureDef.friction = 0.1f;
            ballFixtureDef.shape = &ballShape;

            b2BodyDef ballBodyDef;
            ballBodyDef.type = b2_dynamicBody;
            ballBodyDef.position.Set(0.0f, 25.0f);
            // ballBodyDef.angularDamping = 0.2f;

            m_ball = m_world->CreateBody(&ballBodyDef);
            b2Fixture *ballFixture = m_ball->CreateFixture(&
                ballFixtureDef);

            // Create the baseball
            b2CircleShape ballShape2;
            ballShape2.m_p.SetZero();
            ballShape2.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef2;
            ballFixtureDef2.restitution = 0.75f; // the bounciness
            ballFixtureDef2.density = 0.27f; // this will affect the ball
                 mass
            ballFixtureDef2.friction = 0.1f;
            ballFixtureDef2.shape = &ballShape2;

            b2BodyDef ballBodyDef2;
            ballBodyDef2.type = b2_dynamicBody;
            ballBodyDef2.position.Set(0.0f, 30.0f);
            // ballBodyDef2.angularDamping = 0.2f;

            m_ball2 = m_world->CreateBody(&ballBodyDef2);
            b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
                ballFixtureDef2);

            m_createTime = timer.GetMilliseconds();
            m_time = 0.0f;
        }
        b2Body* bodylbox;
```

```
        b2Body* m_ball;
        b2Body* m_ball2;
        float m_time;

        void Step(Settings& settings) override
        {
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)=
                    %4.2f", m_time);
                m_textLine += m_textIncrement;

                b2MassData massData2 = m_ball2->GetMassData();
                b2Vec2 position2 = m_ball2->GetPosition();

                g_debugDraw.DrawString(5, m_textLine, "Baseball Mass = %.6f",
                     massData2.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Baseball Position =
                    (%4.2f, %4.2f)", position2.x, position2.y);
                m_textLine += m_textIncrement;

                b2MassData massData = m_ball->GetMassData();
                b2Vec2 position = m_ball->GetPosition();

                g_debugDraw.DrawString(5, m_textLine, "Basketball Mass = %.6f
                    ", massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Basketball Position =
                    (%4.2f, %4.2f)", position.x, position.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
                    ",
                m_createTime);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f \n", position2.y, position.y);

                Test::Step(settings);
        }

        static Test* Create()
        {
                return new BaseballBasketball;

        }
```

```
        float m_createTime;
    };

    static int testIndex = RegisterTest("Center of Mass", "Baseball Basketball"
        , BaseballBasketball::Create);
```

**C++ Code 85:** *tests/centerofmass_elasticcollisiontwoballs.cpp "Center of Mass Elastic Collision Baseball Basketball Box2D"*

**Figure 11.24:** *The simulation of a baseball that is aligned above a basketball, both are dropped simulataneously (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/centerofmass_elasticcollisiontwoballs.cpp****).*

## XV. Collisions in Two Dimensions

[DF*] When two bodies collide, the impulse between them determines the directions in which they then travel. In particular, when the collision is not head-on, the bodies do not end up traveling along their initial axis. For such two-dimensional collisions in a closed, isolated system, the total linear momentum must still be conserved:

$$\vec{P}_{1i} + \vec{P}_{2i} = \vec{P}_{1f} + \vec{P}_{2f} \tag{11.61}$$

If the collision is also elastic (a special case), then the total kinetic energy is also conserved:

$$K_{1i} + K_{2i} = K_{1f} + K_{2f} \tag{11.62}$$

The impulses between the bodies have sent the bodies off at angle $\theta_1$ and $\theta_2$ to the $x$

**Figure 11.25:** *An elastic collision between two bodies in which the collision is not head-on. The body with mass $m_2$ (the target) is initially at rest.*

axis, along which the projectile initially traveled. In this situation we would rewrite for components along the $x$ axis as

$$m_1 v_{1i} = m_1 v_{1f} \cos\ \theta_1 + m_2 v_{2f} \cos\ \theta_2 \tag{11.63}$$

and along the $y$ axis as

$$0 = m_1 v_{1f} \sin\ \theta_1 + m_2 v_{2f} \sin\ \theta_2 \tag{11.64}$$

We can also write for the special case of an elastic collision in terms of speeds:

$$\frac{1}{2} m_1 v_{1i}^2 = \frac{1}{2} m_1 v_{1f}^2 + \frac{1}{2} m_2 v_{2f}^2 \tag{11.65}$$

The 3 equations above contain seven variables, if we know any four of these quantities, then with three equations we can solve for the remaining three quantities.

# XVI. System with Varying Mass: A Rocket

[DF*] Previously, we have assumed that the total mass of the system remains constant. Sometimes, as in a rocket, it does not. Most of the mass of a rocket on its launching pad is fuel, all of which will eventually be burned and ejected from the nozzle of the rocket engine. We handle the variation of the mass of the rocket as the rocket accelerates by applying Newton's second law, not to the rocket alone but to the rocket and its ejected combustion products taken together. The mass of this system does not change as the rocket accelerates.

[DF*] **Finding the Acceleration**

Assume that we are at rest relative to an inertial reference frame, watching a rocket accelerate through deep space with no gravitational or atmospheric drag forces acting on it.

For this one-dimensional motion, let $M$ be the mass of the rocket and $v$ its velocity at an arbitrary time $t$. The figure shows how things stand at time interval $dt$ later. The rocket now

**Figure 11.26:** *An accelerating rocket of mass M at time t, as seen from an inertial reference frame.*

**Figure 11.27:** *The same but at time $t + dt$. The exhaust products released during interval $dt$ are shown. The ejection of mass from the rocket's rear increases the rocket's speed.*

has velocity $v + dv$ and mass $M + dM$, where the change in mass $dM$ is a negative quantity. The exhaust products released by the rocket during interval $dt$ have mass $-dM$ and velocity $U$ relative to our inertial reference frame.

[DF*] Our system consists of the rocket and the exhaust products released during interval $dt$. The system is closed and isolated, so the linear momentum of the sustem must be conserved during $dt$; that is,

$$P_i = P_f \tag{11.66}$$

where the subscript $i$ and $f$ indicate the values at the beginning and end of time interval $dt$. We will have

$$Mv = -dM\,U + (M + dM)(v + dv) \tag{11.67}$$

where the first term on the right is the linear momentum of the exhaust products released during interval $dt$ and the second term is the linear momentum of the rocket at the end on interval $dt$.

[DF*] We can simplify by using the relative speed $v_{rel}$ between the rocket and the exhaust products, which is related to the velocities relative to the frame with the velocity of rocket relative to frame equal to the velocity of rocket relative to products added by the velocity of products relative to frame.

In symbols, this means

$$(v + dv) = v_{rel} + U$$
$$U = v + dv - v_{rel}$$

Substituting this result for $U$ yields

$$-\,dM\ v_{rel} = M\ dv \tag{11.68}$$

Dividing each side by $dt$ gives us

$$-\frac{dM}{dt} v_{rel} = M\frac{dv}{dt} \tag{11.69}$$

We replace $\frac{dM}{dt}$ (the rate at which the rocket loses mass) by $-R$, where $R$ is the (positive) mass rate of fuel consumption, and we recognize that $\frac{dv}{dt}$ is the acceleration of the rocket. With these changes, we will have

$$Rv_{rel} = Ma \tag{11.70}$$

called the first rocket equation, it holds for the values at any given instant.
Note the left side has the dimensions of force ($kg \cdot m/s^2 = N$) and depends only on design characteristics of the rocket engine-namely, the rate $R$ at which it consumes fuel mass and the speed $v_{rel}$ with which that mass is ejected relative to the rocket. We call this term $Rv_{rel}$ the thrust of the rocket engine and represent it with $T$. Newton's second law emerges if we write

$$T = Ma$$

in which $a$ is the acceleration of a rocket change as it consumes its mass is $M$.
[DF*] How will the velocity of a rocket change as it consumes its fuel? We have

$$dv = -v_{rel}\frac{dM}{M}$$

integrating leads to

$$\int_{v_i}^{v_f} dv = -v_{rel}\int_{M_i}^{M_f} \frac{dM}{M}$$

in which $M_i$ is the initial mass of the rocket and $M_f$ its final mass. Evaluating the integrals then gives

$$v_f - v_i = v_{rel}\ \ln\frac{M_i}{M_f}$$

for the increase in the speed of the rocket during the change in mass from $M_i$ to $M_f$, the symbol ln means natural logarithm. We see here the advantage of multistage rockets, in which $M_f$ is reduced by discarding successive stages when their fuel is depleted. An ideal rocket would reach its destination with only its payload remaining.
[DF*] **Example:**
A rocket whose initial mass $M_i = 850\ kg$ consumes fuel at the rate $R = 2.3\ kg/s$. The speed $v_{rel}$ of the exhaust gases relative to the rocket engine is 2800 $m/s$.

(a) What thrust does the rocket engine provide?

(b) What is the initial acceleration of the rocket?

**Solution:**

(a) Thrust $T$ is equal to the product of the fuel consumption rate $R$ and the relative speed $v_{rel}$ at which exhaust gases are expelled. Here we find

$$\begin{aligned} T &= Rv_{rel} \\ &= (2.3)(2800) \\ &= 6440\ N \end{aligned}$$

(b) We can relate the thrust $T$ of a rocket to the magnitude $a$ of the resulting acceleration with $T = Ma$, where $M$ is the rocket's mass. However, $M$ decreases and $a$ increases as fuel is consumed. Because we want the initial value of $a$ here, we must use the initial value of $M_i$ of the mass. We find

$$a = \frac{T}{M_i} = \frac{6440}{850} = 7.6\ m/s^2$$

To be launched from Earth's surface, a rocket must have an initial acceleration greater than $g = 9.8\ m/s^2$. That is, it must be greater than the gravitational acceleration at the surface. Put another way, the thrust $T$ of the rocket engine must exceed the initial gravitational force on the rocket, which here has the magnitude $M_i g$, which gives us

$$T_{boundary} = M_i g = (850)(9.8) = 8330\ N$$

Because the acceleration or thrust requirement is not met (here $T = 6400\ N$), our rocket could not be launched from Earth's surface by itself; it would require another, more powerful, rocket.

[DF*] **Example:**
A 6090 $kg$ space probe moving nose-first toward Jupiter at 105 $m/s$ relative to the Sun fires its rocket engine, ejecting 80 $kg$ of exhaust at a speed of 253 $m/s$ relative to the space probe. What is the final velocity of the probe?

**Solution:**

We will use second rocket equation

$$\begin{aligned} v_f - v_i &= v_{rel} \ln\left(\frac{M_i}{M_f}\right) \\ v_f &= v_i + v_{rel} \ln\left(\frac{M_i}{M_f}\right) \\ &= 105 + 253 \ln\left(\frac{6090}{6010}\right) \\ &= 108\ m/s \end{aligned}$$

[DF*] **Example:**
A rocket that is in deep space and initially at rest relative to an inertial reference frame has a mass of $2.55 \times 10^5\ kg$, of which $1.81 \times 10^5\ kg$ is fuel. The rocket engine is then fired for 250 $s$ while fuel is consumed at the rate of 480 $kg/s$. The speed of the exhaust products relative to the rocket is 3.27 $km/s$.

(a) What is the rocket's thrust?

(b) After the 250 $s$ firing, what is the mass of the rocket?

(c) After the 250 $s$ firing, what is the speed of the rocket?

**Solution:**
As fuel is consumed, both the mass and the speed of the rocket will change.

The thrust of the rocket is given by $T = Rv_{rel}$ where $R$ is the rate of fuel consumption and $v_{rel}$ is the speed of the exhaust gas relative to the rocket. On the other hand, the mass of fuel ejected is given by $M_{fuel} = R\triangle t$, where $\triangle t$ is the time interval of the burn. Thus, the mass of the rocket after the burn is

$$M_f = M_i - M_{fuel}$$

(a) Given that $R - 480\ kg/s$ and $v_{rel} = 3.27 \times 10^3\ m/s$, we find the thrust to be

$$\begin{aligned} T &= Rv_{rel} \\ &= (480)(3.27 \times 10^3) \\ &= 1.57 \times 10^6\ N \end{aligned}$$

(b) With the mass of fuel ejected given by

$$M_{fuel} = R\triangle t = (480)(250) = 1.2 \times 10^5\ kg$$

the final mass of the rocket is

$$\begin{aligned} M_f &= M_i - M_{fuel} \\ &= (2.55 \times 10^5) - (1.2 \times 10^5) \\ &= 1.35 \times 10^5\ kg \end{aligned}$$

(c) Since the initial speed is zero, the final speed of the rocket is

$$\begin{aligned} v_f &= v_{rel} \ln\left(\frac{M_i}{M_f}\right) \\ &= (3.27 \times 10^3\ m/s) \ln\left(\frac{2.55 \times 10^5}{1.35 \times 10^5}\right) \\ &= 2.08 \times 10^3\ m/s \end{aligned}$$

Learn the speed of the rocket continues to rise as the fuel is consumed. From the first rocket equation, the thrust of the rocket is related to the acceleration by $T = Ma$. Using this equation, we find the initial acceleration to be

$$\begin{aligned} a_i &= \frac{T}{M_i} \\ &= \frac{1.57 \times 10^6}{2.55 \times 10^5} \\ &= 6.16\ m/s^2 \end{aligned}$$

[DF*] **Example:**
A 6100 $kg$ rocket is set for vertical firing from the ground. If the exhaust speed is 1200 $m/s$.

(a) How much gas must be ejected each second if the thrust is to equal the magnitude of the gravitational force on the rocket?

(b) How much gas must be ejected each second if the thrust is to give the rocket an initial upward acceleration of 21 $m/s^2$?

**Solution:**
To successfully launch a rocket from the gorund, fuel is consumed at a rate that results in a thrust big enough to overcome the gravitational force.

The thrust of the rocket is given by $T = Rv_{rel}$ where $R$ is the rate of fuel consumption and $v_{rel}$ is the speed of the exhaust gas relative to the rocket.

(a) The exhaust speed is $v_{rel} = 1200\ m/s$. For the thrust to equal the weight $Mg$ where $M = 6100\ kg$, we must have

$$\begin{aligned} T &= Rv_{rel} \\ Tv_{rel} &= Mg \\ R &= \frac{Mg}{v_{rel}} \\ &= \frac{(6100)(9.8)}{1200} \\ &= 49.8\ kg/s \end{aligned}$$

(b) With the additional effect due to gravity and requiring that $a = 21\ m/s^2$, we have

$$\begin{aligned} Rv_{rel} - Mg &= Ma \\ R &= \frac{M(g+a)}{v_{rel}} \\ &= \frac{(6100)(9.8+21)}{1200} \\ &= 156.6\ kg/s \end{aligned}$$

A greater upward acceleration requires a greater fuel consumption rate. To be launched from Earth's surface, the initial acceleration of the rocket must exceed $g = 9.8\ m/s^2$. This means that the rate $R$ must be greater than 50 $kg/s$.

[DF*] **Example:**
A 500 $kg$ module is attached to a 400 $kg$ shuttle craft, which moves at 1000 $m/s$ relative to the stationary main spaceship. Then a small explosion sends the module backward with speed 100 $m/s$ relative to the new speed of the shuttle craft. As a measured by someone on the main spaceship, by what fraction did the kinetic energy of the module and shuttle craft increase because of the explosion?
**Solution:**
Conservation of momentum leads to

$$\begin{aligned} Mv_i &= m_{module}(v_{module}) + m_{shuttle}v_{shuttle} \\ (900)(1000) &= (500)(v_{shuttle} - 100) + (400)(v_{shuttle}) \\ v_{shuttle} &= 1055.6\ m/s \end{aligned}$$

so the shuttle speed is $v_{shuttle} = 1055.6\ m/s$ and the module speed is $v_{module} = v_{shuttle} - 100 = 955.6\ m/s$. All measured in the frame of reference of the stationary main spaceship. The

fractional increase in the kinetic energy is

$$\begin{aligned}\frac{\triangle K}{K_i} &= \frac{K_f}{K_i} - 1 \\ &= \frac{\frac{1}{2}(500)(955.6)^2 + \frac{1}{2}(400)(1055.6)^2}{\frac{1}{2}(900)(1000)^2} \\ &= 2.5 \times 10^{-3}\end{aligned}$$

[DF*] **Example:**
During a lunar mission, it is necessary to increase the speed of a spacecraft by 2.2 $m/s$ when it is moving at 400 $m/s$ relative to the Moon. The speed of the exhaust products from the rocket engine is 1000 $m/s$ relative to the spacecraft. What fraction of the initial mass of the spacecraft must be burned and ejected to accomplish the speed increase?

**Solution:**
We will use the second rocket equation and simplify with $v_f - v_i = \triangle v$, and $v_{rel} = u$.

$$\begin{aligned}v_f - v_i &= v_{rel}\ \ln\left(\frac{M_i}{M_f}\right) \\ \frac{M_f}{M_i} &= e^{\frac{-\triangle v}{u}}\end{aligned}$$

If $\triangle v = 2.2\ m/s$ and $u = 1000\ m/s$, we obtain

$$\frac{M_i - M_f}{M_i} = 1 - e^{-0.0022} \approx 0.0022$$

# XVII. Simulation for with Box2D

**C++ Code 86:** *tests/centerofmass_threeparticles.cpp "Center of Mass Three Particles Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 12

# DFSimulatorC++ VI: Rotation

*"Sudo, oh honey honey.. You are my Cambridge girl.. Honey.. oh Sudo Sudo.. you are my pretty girl" -*
***Sugar song changed to Sudo Sudo by Mischkra and DS Glanzsche***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 87:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 13

# DFSimulatorC++ VII: Rolling, Torque, and Angular Momentum

*"Bam.. baram bam.. bam brem bam bem bam bam.. aya yay aya ya ya yayay.."* - ***Sine (the first Sapirian Husky Dog in the world)***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 88:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 14

# DFSimulatorC++ VIII: Equilibrium and Elasticity

*"Tantum Ergo Sacromentum, veneremur cernui, et antiquum documentum, novo cedat ritui" - **Pange Lingua***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 89:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 15

# DFSimulatorC++ IX: Gravity

*"Nous allons voir"* - ***DS Glanzsche***

We are bounded by gravity, we walk, do activity, run, eat, all affected by gravity, that's why we step on this land of this planet earth. That's why rocket is designed that way with specific machine in order to go against gravity and able to get out of this planet. It should be understood, not only with theory in Physics formula and Mathematics differential equation, but why things fall that way, and for how long till it reach the ground from certain height?

This first simulation is asked by the Elder at Valhalla Projection. I thought I was asked to do some push and pulling box simulation first. But this comes first, let see then. Our favorite phrase in French, "Nous allons voir."

## I. Mathematical Physics Formula for Gravity

We are going to remember how Newton' formula changes our lives, remember that:

$$F = ma$$

means that force that is applied to a body is proportional with the mass and acceleration that is ongoing with the body. Now for an object that is falling we will have:

$$F = mg$$

The different is that we regard $a$ as the acceleration towards horizontal axis, while $g$ is the acceleration towards the vertical axis.

Based on elementary differential equation [1], we will have:

$$\begin{aligned} F &= mg \\ m\frac{dv}{dt} &= mg \\ \text{(without air drag)} & \\ F &= mg - \gamma v \\ m\frac{dv}{dt} &= mg - \gamma v(t) \end{aligned}$$

with $v$ is the speed or velocity of the object that is falling, a variable that depends on $t$, time variable, thus we will write $v(t)$ instead of just $v$. We need to solve the differential equation above to obtain:

$$v(t),\ x(t),\ T$$

with $v(t)$ is the solution that represents the velocity of the falling object at time $t$, and $x(t)$ is the distance that the objects fall at time $t$, and time $T$ is the time required for the object to fall for certain height, say $h$ meters. We need to solve this so then we can input the numerical computation into C++ code so the animation or simulation of the falling object due to gravity with (or without) air drag is realistic.

Now, since computer is really helpful for computation in symbolic and numerical terms, I choose to use JULIA to solve the differential equation above. Thus,

$$\begin{aligned} m\frac{dv}{dt} &= mg - \gamma v(t) \\ \frac{dv}{dt} &= g - \frac{\gamma v(t)}{m} \\ v(t) &= \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma} \end{aligned}$$

Now if we substitute the value for $m = 10$, an object with mass that is 10 kg, $g = 9.8\ m/s^2$, which is the gravity on earth, and the air drag coefficient is $\gamma = 0.3$, we will have:

$$v(t) = 326.667 - 326.667e^{-0.03t}$$

If we alter the input parameters, such as different mass for the object or on different planet with different gravity, we will obtain different numerical solution. Now what we need to input to C++ code is this:

$$v(t) = \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma}$$

Then we define $m, g, \gamma$, and eventually $h$, the height from which the object falls, later on. There should be impact as well if the speed is too fast and the height is too low, like meteor hitting on this planet on Dinosaur era, but we will talk about it later on.

```
julia> include("diffeq.jl")
The differential equation:
  d
m·──(v(t)) = g·m - γ·v(t)
  dt
Initial Value problem solution:
                    -γ·t
                    ────
                     m
       g·m   g·m·e
v(t) = ───── - ──────────
        γ        γ
The solution for the differential equation with certain parameter inputs:
                                                 -0.03·t
v(t) = 326.666666666667 - 326.666666666667·e
       ─
```

**Figure 15.1:** *The symbolical computation for finding the solution of an object falling with JULIA and the SymPy package (**DFSimulatorC/Source Codes/Julia/ch5-1-gravitydiffeq.jl**).*

It is very important and necessary, as we are having a lot of air flights and air sports currently, knowing how to jump with parachute from an airplane and estimate time to land with certain speed can be of help one day.

We are going to compare the gravity / falling bodies simulation between Bullet physics engine and ReactPhysics3D.

## II. Simulation for Gravity with Bullet3, GLEW, GLFW and OpenGL

Since this is the very first simulation asked by the elder, by coincidence a book has a nice scope on this with Bullet3, thus after modifying I am going to show how to simulate 3 dynamic bodies with shape of a sphere and different masses fall from the same height. Inspired by Chapter 7 from [12], I add two other bodies. You can directly copy the files for this example or type it manually.

We can compile all examples from Bullet, but I prefer to create one like this (per project based), then compile, compiling all examples for Bullet took so long, since it already expanded. With only using the necessary library for our test simulation purpose, we won't need to compile all other examples, that in my opinion is more time saving and can help to focus more on current physics problem and its' simulation.

In a new directory, create a file by opening a terminal and type:
**vim main.cpp**

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>
#include "ShaderLoader.h"
#include "Camera.h"
#include "LightRenderer.h"

#include "MeshRenderer.h"
#include "TextureLoader.h"
#include <btBulletDynamicsCommon.h>
#include<chrono>

void initGame();
void renderScene();

Camera* camera;
LightRenderer* light;

MeshRenderer* sphere;
MeshRenderer* sphere2;
MeshRenderer* sphere3;
MeshRenderer* ground;

//physics
```

```
btDiscreteDynamicsWorld* dynamicsWorld;

void initGame() {

        // Enable the depth testing
        glEnable(GL_DEPTH_TEST);

        ShaderLoader shader;

        GLuint flatShaderProgram = shader.createProgram("/root/
            SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/
            SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
        GLuint texturedShaderProgram = shader.createProgram("/root/
            SourceCodes/CPP/Assets/Shaders/TexturedModel.vs", "/root/
            SourceCodes/CPP/Assets/Shaders/TexturedModel.fs");

        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3
            (0.0f, 8.0f, 30.0f)); //camera position at y=+8 and z=+30

        light = new LightRenderer(MeshType::kTriangle, camera);
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 3.0f, 0.0f));

        TextureLoader tLoader;

        GLuint sphereTexture = tLoader.getTextureID("/root/
            SourceCodes/CPP/Assets/Textures/globe.jpg");
        GLuint groundTexture = tLoader.getTextureID("/root/
            SourceCodes/CPP/Assets/Textures/ground.jpg");

        //init physics
        btBroadphaseInterface* broadphase = new btDbvtBroadphase();
        btDefaultCollisionConfiguration* collisionConfiguration = new
             btDefaultCollisionConfiguration();
        btCollisionDispatcher* dispatcher = new btCollisionDispatcher
            (collisionConfiguration);
        btSequentialImpulseConstraintSolver* solver = new
            btSequentialImpulseConstraintSolver();

        dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
            broadphase, solver, collisionConfiguration);
        dynamicsWorld->setGravity(btVector3(0, -0.8f, 0));

        // Sphere Rigid Body
        btCollisionShape* sphereShape = new btSphereShape(1); //
            sphere with radius 1
        btCollisionShape* sphereShape2 = new btSphereShape(1);
        btCollisionShape* sphereShape3 = new btSphereShape(1);
```

```
// Set the initial location where the spheres fall
btDefaultMotionState* sphereMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
    , btVector3(-3, 13, 0)));
btDefaultMotionState* sphereMotionState2 = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
    , btVector3(0, 13, 0)));
btDefaultMotionState* sphereMotionState3 = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
    , btVector3(3, 13, 0)));

btScalar mass = 5.0;
btScalar mass2 = 10.0;
btScalar mass3 = 15.0;
btVector3 sphereInertia(0, 0, 0);
btVector3 sphereInertia2(0, 0, 0);
btVector3 sphereInertia3(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);
sphereShape2->calculateLocalInertia(mass2, sphereInertia2);
sphereShape3->calculateLocalInertia(mass3, sphereInertia3);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(
    mass, sphereMotionState, sphereShape, sphereInertia);
btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI2(
    mass2, sphereMotionState2, sphereShape2, sphereInertia2);
btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI3(
    mass3, sphereMotionState3, sphereShape3, sphereInertia3);

btRigidBody* sphereRigidBody = new btRigidBody(
    sphereRigidBodyCI);
btRigidBody* sphereRigidBody2 = new btRigidBody(
    sphereRigidBodyCI2);
btRigidBody* sphereRigidBody3 = new btRigidBody(
    sphereRigidBodyCI3);
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);
sphereRigidBody2->setRestitution(1.0f);
sphereRigidBody2->setFriction(1.0f);
sphereRigidBody3->setRestitution(1.0f);
sphereRigidBody3->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);
dynamicsWorld->addRigidBody(sphereRigidBody2);
dynamicsWorld->addRigidBody(sphereRigidBody3);

// Sphere 1 Mesh
sphere = new MeshRenderer(MeshType::kSphere, camera,
```

```
                    sphereRigidBody);
            sphere->setProgram(texturedShaderProgram);
            sphere->setTexture(sphereTexture);
            sphere->setScale(glm::vec3(1.0f));

            //Sphere 2 Mesh
            sphere2 = new MeshRenderer(MeshType::kSphere, camera,
                sphereRigidBody2);
            sphere2->setProgram(texturedShaderProgram);
            sphere2->setTexture(sphereTexture);
            sphere2->setScale(glm::vec3(1.0f));

            // Sphere 3 Mesh
            sphere3 = new MeshRenderer(MeshType::kSphere, camera,
                sphereRigidBody3);
            sphere3->setProgram(texturedShaderProgram);
            sphere3->setTexture(sphereTexture);
            sphere3->setScale(glm::vec3(1.0f));

            // Ground Rigid body
            btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f
                , 0.5f, 4.0f));
            btDefaultMotionState* groundMotionState = new
                btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
                , btVector3(0, -2.0f, 0)));

            btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI
                (0.0f, new btDefaultMotionState(), groundShape, btVector3
                (0, 0, 0));

            btRigidBody* groundRigidBody = new btRigidBody(
                groundRigidBodyCI);

            groundRigidBody->setFriction(1.0);
            groundRigidBody->setRestitution(0.5);

            groundRigidBody->setCollisionFlags(btCollisionObject::
                CF_STATIC_OBJECT);

            dynamicsWorld->addRigidBody(groundRigidBody);

            // Ground Mesh
            ground = new MeshRenderer(MeshType::kCube, camera,
                groundRigidBody);
            ground->setProgram(texturedShaderProgram);
            ground->setTexture(groundTexture);
            ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));
        }
```

```
void renderScene(){
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glClearColor(1.0, 1.0, 0.0, 1.0);

	//light->draw();
	sphere->draw();
	sphere2->draw();
	sphere3->draw();
	ground->draw();
}

int main(int argc, char **argv)
{
	glfwInit();
	GLFWwindow* window = glfwCreateWindow(800, 600, " Bullet and
	    OpenGL ", NULL, NULL);
	glfwMakeContextCurrent(window);
	glewInit();

	initGame();
	auto previousTime = std::chrono::high_resolution_clock::now()
	    ;
	while (!glfwWindowShouldClose(window)){

		auto currentTime = std::chrono::high_resolution_clock
		    ::now();
		float dt = std::chrono::duration<float, std::chrono::
		    seconds::period>(currentTime - previousTime).
		    count();

		dynamicsWorld->stepSimulation(dt);

		renderScene();

		glfwSwapBuffers(window);
		glfwPollEvents();

		previousTime = currentTime;
	}
	glfwTerminate();

	delete camera;
	delete light;

	return 0;
}
```

**C++ Code 90:** *main.cpp "Gravity with 3 Bodies"*

Do not follow 100 %, read slowly and try to debug and tinker by yourself, if there are warnings and errors find the solutions by yourself if possible, otherwise try to ask. Another thing is do kindly adjust the path, for example at these lines:

```
GLuint flatShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/FlatModel.vs", "/root/SourceCodes/CPP/Assets/Shaders/
    FlatModel.fs");
GLuint texturedShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/TexturedModel.vs", "/root/SourceCodes/CPP/Assets/Shaders
    /TexturedModel.fs");
```

Your path shall not be the same as mine: **/root/SourceCodes/CPP/Assets/Shaders/FlatModel.fs** for **FlatModel.fs**, a fragment shader. All the shader and texture and images related are inside repository as well, you can suit yourself if you like it use that or use your own.

Some explanations for the codes:

- To add physics we use:
  **#include <btBulletDynamicsCommon.h>**

- The Bullet libraries we are using are: **BulletCollision, BulletDynamics, LinearMath**. We will need to link to this library when compiling either with CMake or manual compiling.
- To create a Physics world we use this:
  **btDiscreteDynamicsWorld* dynamicsWorld**;
- After create the Physics world, then inside the **void initGame()** we will have:

```
btBroadphaseInterface* broadphase = new btDbvtBroadphase();
btDefaultCollisionConfiguration* collisionConfiguration = new
    btDefaultCollisionConfiguration();
btCollisionDispatcher* dispatcher = new btCollisionDispatcher(
    collisionConfiguration);
btSequentialImpulseConstraintSolver* solver = new
    btSequentialImpulseConstraintSolver();
```

  Collision detection is done in two phase: broadphase (the physics engine eliminates all the objects that are unlikely to collide) and narrowphase (the actual shape of the object is used to check the likelihood of a collision). Pairs of object are created with a strong likelihood of collision. For **btCollisionDispatcher**, a pair of objects that have a strong likelihood for colliding are tested for collision using their actual shapes. For **btSequentialImpulseConstraintSolver**, you can create constraints, such as a hinge constraint or slider constraint which can restrict motion or rotation of one object about another object. The calculation is repeated a number of times to get the optimal solution.
- Still in **void initGame()**:

```
dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
    broadphase, solver, collisionConfiguration);
dynamicsWorld->setGravity(btVector3(0, -9.8f, 0));
```

  to create a new **dynamicsWorld** by passing the **dispatcher, broadphase, solver**, and **collisionConfiguration** as parameters to the **btDiscreteDynamicsWorld** function. After that we can set the parameters for our physics with gravity of $-9.8f$ in the $y$-axis.

- After finish with the Physics world, we can create rigid bodies or soft bodies, still in **void initGame()**:

```
btCollisionShape* sphereShape = new btSphereShape(1);

btDefaultMotionState* sphereMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
    btVector3(0, 15, 0)));

btScalar mass = 10.0;
btVector3 sphereInertia(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass
    , sphereMotionState, sphereShape, sphereInertia);

btRigidBody* sphereRigidBody = new btRigidBody(
    sphereRigidBodyCI);
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);
```

**new btSphereShape(1);** means create a sphere shape with radius of 1. The **btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1), btVector3(0, 15, 0)));** specifies the rotation and position of the sphere, it will be located on $x = 0, y = 15, z = 0$. To test it you may change the **btQuaternion(0, 0, 0, 1)** into **btQuaternion(0.3, 0, 0.5, 0.7)** to see how the sphere rotates while falling. After that, we set the mass to be 10, the inertia, and calculate inertia of the **sphereShape**.

Then, to create the rigid body, we first create **btRigidBodyConstructionInfo** and pass the rigid body' variables / parameters. We then set physical properties of the rigid body, such as the restitution and the friction. For restitution 0, means the rigid body will have no bounciness, while 1 means the rigid body is very bouncy, like a basketball. While for friction, 0 means a smooth rigid body, while 1 means a very rough rigid body.

Finally, we add the rigid body to the Physics world.

- In **Mesh.cpp**, we define the vertices for Triangle, Quad, Cube, and Sphere. Thus in **MeshRenderer.h** and **MeshRenderer.cpp** we are not only render the sphere, but to make it behave like a sphere that obeying the law of physics, we pass the rigid body to the sphere mesh.

  We use **btTransform** variable to get the transformation from the rigid body's **getMotionState** function and then get the **WorldTransform** variable and set it to our **btTransform** variable **t**.

  We create **btQuaternion** type to store rotation and **btvector3** to store translation values using the **getRotation** and **getOrigin** functions of the **btTransform** class.

  We create three **glm::mat4** variables, called **RotationMatrix, TranslationMatrix,** and **ScaleMatrix**. We set the values of rotation and translation using the **glm::rotate** and **glm::translation**

functions.

The rest of the files (.cpp and .h) can be obtained at the repository, all will be (don't forget the texture, vertex and fragment shaders):

- Camera.cpp
- Camera.h
- CMakeLists.txt
- LightRenderer.cpp
- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- MeshRenderer.cpp
- MeshRenderer.h
- ShaderLoader.cpp
- ShaderLoader.h
- TextureLoader.cpp
- TextureLoader.h

After having all of them then open the terminal at this working directory and type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./result**

If you want to compile manually without CMake, type:
**g++ *.cpp -o result -lBulletCollision -lBulletDynamics -lLinearMath -lGLEW -lglfw -lGL -I/usr/include/stb-master**
**./result**

This example has no air drag. If we want an air drag we need to damp the velocity when the bodies are falling. Air drag depends on the geometry shape and material of the object. On October 26th, 2023 I drop down from the same height a small dog doll and 1 Euro cent coin, they fall at the same time. But, when I do that with the Euro coin and a piece of paper, the paper still flying when the coin hits the ground. That's Physics, why that happens? Back to the geometry shape and material of the paper.

Calculate the time it hits the ground. Is mass not affecting time to hit the ground? Check the formula, etc.

# III. Simulation for Gravity with Box2D

## i. Build DianFreya Modified Box2D Testbed

We have modified all the codes for Box2D testbed by removing the original tests codes into our own codes to learn Physics and Math from beginning, with Box2D library and imgui, learning C++, Physics and Math become more fun. You are welcome to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:

**Figure 15.2:** *Linking the libraries from Bullet for this simulation, make sure that you have set the right LIBRARY_PATH for the Bullet' libraries that were installed and compiled.*

**Figure 15.3:** *Three falling bodies with shape of sphere, from left to right have masses of: 5, 10, then 15. (files for this example in folder:* ***DFSimulatorC/Source Codes/C++/DianFreya-Bullet/Gravity and 3 Bodies****).*

**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

If you are having a difficulty or error, read again chapter 5 on how to download, compile and install Box2D from raw to become library, along with copying the headers of imgui and sajson. If you read again, besides the source codes inside **../tests/** that I replaced, the **CMakeLists.txt** is being modified to include the source codes from the newly modified source codes in **../tests/**, thus when the testbed is being build it will only show the new tests for this book simulation, not the original default Box2D testbed tests.

Look for the related simulation under the **Tests** tab on the right panel, then choose **Gravitation/Gravity Check**.

```
#include "test.h"
#include <iostream>

class GravityCheck: public Test
{
	public:

	GravityCheck()
	{
		m_world->SetGravity(b2Vec2(0.0f,-9.8f));
		b2Timer timer;
		// Perimeter Ground body
		{
			b2BodyDef bd;
			b2Body* ground = m_world->CreateBody(&bd);

			b2EdgeShape shapeGround;
			shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
				(30.0f, 0.0f));
			ground->CreateFixture(&shapeGround, 0.0f);

			b2EdgeShape shapeTop;
			shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
				(30.0f, 30.0f));
			ground->CreateFixture(&shapeTop, 0.0f);

			b2EdgeShape shapeLeft;
			shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
				(-40.0f, 30.0f));
			ground->CreateFixture(&shapeLeft, 0.0f);

			b2EdgeShape shapeRight;
			shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
```

```
                    (30.0f, 30.0f));
                ground->CreateFixture(&shapeRight, 0.0f);
            }

            // Create the left ball
            b2CircleShape ballShape2;
            ballShape2.m_p.SetZero();
            ballShape2.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef2;
            ballFixtureDef2.restitution = 0.75f; // the bounciness
            ballFixtureDef2.density = 7.3f; // this will affect the ball
                mass
            ballFixtureDef2.friction = 0.1f;
            ballFixtureDef2.shape = &ballShape2;

            b2BodyDef ballBodyDef2;
            ballBodyDef2.type = b2_dynamicBody;
            ballBodyDef2.position.Set(-10.0f, 25.0f);
            // ballBodyDef2.angularDamping = 0.2f;

            m_ball2 = m_world->CreateBody(&ballBodyDef2);
            b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
                ballFixtureDef2);

            // Create the right ball
            b2CircleShape ballShape;
            ballShape.m_p.SetZero();
            ballShape.m_radius = 0.5f;

            b2FixtureDef ballFixtureDef;
            ballFixtureDef.restitution = 0.75f; // the bounciness
            ballFixtureDef.density = 3.3f; // this will affect the ball
                mass
            ballFixtureDef.friction = 0.1f;
            ballFixtureDef.shape = &ballShape;

            b2BodyDef ballBodyDef;
            ballBodyDef.type = b2_dynamicBody;
            ballBodyDef.position.Set(0.0f, 25.0f);
            // ballBodyDef.angularDamping = 0.2f;

            m_ball = m_world->CreateBody(&ballBodyDef);
            b2Fixture *ballFixture = m_ball->CreateFixture(&
                ballFixtureDef);
            m_createTime = timer.GetMilliseconds();
        }
        b2Body* m_ball;
```

```
        b2Body* m_ball2;
        void Step(Settings& settings) override
        {
            b2MassData massData2 = m_ball2->GetMassData();
            g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass = %.6f"
                , massData2.mass);
            m_textLine += m_textIncrement;

            b2Vec2 position2 = m_ball2->GetPosition();
            g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, x
                = %.6f", position2.x);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, y
                = %.6f", position2.y);
            m_textLine += m_textIncrement;

            b2MassData massData = m_ball->GetMassData();
            g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass = %.6f
                ", massData.mass);
            m_textLine += m_textIncrement;

            b2Vec2 position = m_ball->GetPosition();
            g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, x
                 = %.6f", position.x);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, y
                 = %.6f", position.y);
            m_textLine += m_textIncrement;

            g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
                ",
            m_createTime);
            m_textLine += m_textIncrement;

            printf("%4.2f %4.2f \n", position2.y, position.y);

            Test::Step(settings);
        }

        static Test* Create()
        {
            return new GravityCheck;

        }

        float m_createTime;
    };
```

```
static int testIndex = RegisterTest("Gravitation", "Gravity Check",
    GravityCheck::Create);
```

**C++ Code 91:** *tests/gravity_check.cpp "Gravity Check Box2D"*

Some explanations for the codes:

- To create the green perimeter:

```
// Perimeter Ground body
{
        b2BodyDef bd;
        b2Body* ground = m_world->CreateBody(&bd);

        b2EdgeShape shapeGround;
        shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
            (30.0f, 0.0f));
        ground->CreateFixture(&shapeGround, 0.0f);

        b2EdgeShape shapeTop;
        shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2(30.0f
            , 30.0f));
        ground->CreateFixture(&shapeTop, 0.0f);

        b2EdgeShape shapeLeft;
        shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
            (-40.0f, 30.0f));
        ground->CreateFixture(&shapeLeft, 0.0f);

        b2EdgeShape shapeRight;
        shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2(30.0f
            , 30.0f));
        ground->CreateFixture(&shapeRight, 0.0f);
}
```

- Create two balls objects inside **GravityCheck()**

```
// Create the left ball
b2CircleShape ballShape2;
ballShape2.m_p.SetZero();
ballShape2.m_radius = 0.5f;

b2FixtureDef ballFixtureDef2;
ballFixtureDef2.restitution = 0.75f; // the bounciness
ballFixtureDef2.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef2.friction = 0.1f;
ballFixtureDef2.shape = &ballShape2;

b2BodyDef ballBodyDef2;
ballBodyDef2.type = b2_dynamicBody;
```

```
ballBodyDef2.position.Set(-10.0f, 25.0f);
// ballBodyDef2.angularDamping = 0.2f;

m_ball2 = m_world->CreateBody(&ballBodyDef2);
b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
    ballFixtureDef2);

// Create the right ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f; // the bounciness
ballFixtureDef.density = 3.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
m_createTime = timer.GetMilliseconds();
```

- Declare each of the ball as Box2D body.

```
b2Body* m_ball;
b2Body* m_ball2;
```

- To show on the top left of the screen for the $x$ and $y$ position of each ball (left and right), along with its mass. The $y$ position for each ball will be printed out on the terminal / xterm

```
void Step(Settings& settings) override
{
       b2MassData massData2 = m_ball2->GetMassData();
       g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass =
           %.6f", massData2.mass);
       m_textLine += m_textIncrement;

       b2Vec2 position2 = m_ball2->GetPosition();
       g_debugDraw.DrawString(5, m_textLine, "Left Ball
           Position, x = %.6f", position2.x);
       m_textLine += m_textIncrement;
       g_debugDraw.DrawString(5, m_textLine, "Left Ball
           Position, y = %.6f", position2.y);
```

```
                m_textLine += m_textIncrement;

                b2MassData massData = m_ball->GetMassData();
                g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass =
                     %.6f", massData.mass);
                m_textLine += m_textIncrement;

                b2Vec2 position = m_ball->GetPosition();
                g_debugDraw.DrawString(5, m_textLine, "Right Ball
                    Position, x = %.6f", position.x);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Ball
                    Position, y = %.6f", position.y);
                m_textLine += m_textIncrement;

                g_debugDraw.DrawString(5, m_textLine, "create time =
                    %6.2f ms",
                m_createTime);
                m_textLine += m_textIncrement;

                printf("%4.2f %4.2f \n", position2.y, position.y);

                Test::Step(settings);
            }
```

Now, if you want to save the output into textfile (.txt), you can type this before running testbed from terminal:
**./testbed > ballgravity.txt**
it will save the textfile named **ballgravity.txt** inside the directory containing the **testbed**, watch out that the first two lines usually contain strings, for my case it is the data of my OpenGL, GLSL and Mesa version, just delete that strings and leave the two columns of numerical data only.

Now, we can use **Gnuplot** to help plot the $y$ position data (the height) of each ball with respect to time, now open terminal at the directory containing the **ballgravity.txt** and type:
**gnuplot**
**set xlabel "time"**
**set ylabel "y position"**
**plot "ballgravity.txt" using 1 title "Ball mass 5.73" with linespoints, "ballgravity.txt" using 2 title "Ball mass 2.59" with linespoints**

## IV. Simulation for Gravity with ReactPhysics3D

**Figure 15.4:** *The falling object due to gravity simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/gravity_check.cpp****).*

**Figure 15.5:** *The Gnuplot of the* ***ballgravity.txt****. We want to see the behavior of same shape objects fall from same height, same restitution / bounciness and friction. The only different is the mass. Turns out they have the exact same behavior while falling.*

# Chapter 16

# DFSimulatorC++ X: Fluids

*"My mama told me no more breastfeed, but I said 'yes to breastfeed everyday'" - **Sweden at 4 months old***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 92:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 17

# DFSimulatorC++ XI: Oscillations

*"You don't need a reason to study science and engineering, it is better than wasting time partying or gossiping. Great minds always discuss idea not discuss people."* - ***DS Glanzsche***

*"Physical problems cannot be analyzed by mathematics alone"* - ***from Mathematical Models book (Richard Haberman)***

When objects move back and forth repeatedly or periodically then that objects are oscillating [6]. If you play PlayStation with the vibrating joystick, when it vibrates, it is oscillating. I believe we don't need a lot of reasons to learn science, learn it as a means to spend time worthwhile.

Breakthrough for real are made by people like Leonardo Da Vinci and Isaac Newton (never marry and die as virgin, after hundreds of years their inventions still used and for hundreds of years more), transmuting sexual energy and desire into productive energy worth more than exponential power of compound interest. Real scientists and engineers don't do orgy or sex party, their party or celebration granted by the divine, as an example one of their creations bought by royalty for USD 400 million for a painting of "Salvator Mundi", or named as one of the great airport that sell a nice Insalata Salmon and Croissant.

## I. Simple Harmonic Motion

The frequency of the oscillation, denoted by $f$, is the number of times per second that it completes a full oscillation (a cycle) and has the unit of hertz (Hz), where

$$1\ Hz = 1\ s^{-1}$$

The time for one full cycle is the period $T$ of the oscillation, which is

$$T = \frac{1}{f} \tag{17.1}$$

A simple harmonic motion (SHM) is a sinusoidal function of time $t$. That is, it can be written as a sine or a cosine of time $t$. Therefore

$$x(t) = x_m\ \cos(\omega t + \phi) \tag{17.2}$$

with $x(t)$ is the displacement at time $t$, $x_m$ is the amplitude, $t$ is time, $\phi$ is the phase constant or phase angle, $\omega$ is the angular frequency of the motion, together $(\omega t + \phi)$ is the phase.

To relate the angular frequency to the frequency $f$ and period $T$, note that the position $x(t)$ of the object / particle must return to its initial value at the end of a period. That is, if $x(t)$ is the position at some chose time $t$, then the particle must return to that same position at time $t + T$. Returning to the same position can be written as

$$x_m \ \cos \ \omega t = x_m \ \cos \ \omega(t + T) \tag{17.3}$$

with

$$\begin{aligned} \omega(t + T) &= \omega t + 2\pi \\ \omega T &= 2\pi \\ \omega &= \frac{2\pi}{T} = 2\pi f \end{aligned}$$

the SI unit of angular frequency is the radian per second.

Now to find the velocity of the simple harmonic motion, we use the derivative with respect to time.

$$\begin{aligned} v(t) &= \frac{dx(t)}{dt} \\ &= \frac{d}{dt}[x_m \ \cos(\omega t + \phi)] \\ &= -\omega \ x_m \ \sin(\omega t + \phi) \end{aligned}$$

The velocity depends on time because the sine function varies with time, with the range of value for sine is $[-1, 1]$.

Now, if we differentiate the velocity of the simple harmonic motion, we will obtain the acceleration function of the particle in simple harmonic motion:

$$\begin{aligned} a(t) &= \frac{dv(t)}{dt} \\ &= \frac{d}{dt}[-\omega \ x_m \ \sin(\omega t + \phi)] \\ &= -\omega^2 \ x_m \ \cos(\omega t + \phi) \end{aligned}$$

The acceleration varies as well, because cosine function varies with time. Thus, we will see that

$$a(t) = -\omega^2 x(t)$$

- The particle's acceleration is always opposite its displacement
- To tell that a physical phenomena is in the category of simple harmonic motion, its acceleration and displacement will be related by a constant $\omega^2$, with $\omega$ is the angular frequency of the motion.

## i. The Force Law for Simple Harmonic Motion

We can apply Newton's second law to describe the force responsible for simple harmonic motion:

$$F = ma = m(-\omega^2 x) = -(m\omega^2)x$$

The minus sign means that the direction of the force on the particle is opposite the direction of the displacement. In SHM, the force is a restoring force that it fights against the displacement, attempting to restore the particle to the center point at $x = 0$.

If we relate it with Hooke's law that stated

$$F = -kx$$

we can relate the spring constant $k$ (a measure of the stiffness of the spring) to the mass of the block and the resulting angular frequency of the simple harmonic motion, we will obtain:

$$k = m\omega^2$$

The block-spring system is called a linear simple harmonic oscillator, since $F$ is proportional to $x$ to the first power. If you ever see a situation in which the force in an oscillation is always proportional to the displacement but in the opposite direction, you can say that the oscillation is simple harmonic motion. If you know the oscillating mass, you can determine the angular frequency of the motion as

$$\omega = \sqrt{\frac{k}{m}}$$

you can also determine the period of the motion with

$$T = 2\pi\sqrt{\frac{m}{k}}$$

Every oscillating system, like a violin string, has some element of "springiness" and some element of "inertia" or mass.

## II. Energy in Simple Harmonic Motion

A particle in simple harmonic motion has, at any time, kinetic energy of

$$K = \frac{1}{2}mv^2$$

and potential energy of

$$U = \frac{1}{2}kx^2$$

If no friction is present, the mechanical energy of the oscillator will be

$$E = K + U \tag{17.4}$$

remains constant, even though $K$ and $U$ change over time.

Now, if we see the potential energy as the function of time

$$U(t) = \frac{1}{2}kx^2 = \frac{1}{2}kx_m^2\cos^2(\omega t + \phi)$$

we can also write the kinetic energy as the function of time

$$K(t) = \frac{1}{2}mv^2 = \frac{1}{2}kx_m^2\sin^2(\omega t + \phi)$$

combining both, we will get

$$\begin{aligned}
E &= U(t) + K(t) \\
&= \frac{1}{2}kx_m^2\cos^2(\omega t + \phi) + \frac{1}{2}kx_m^2\sin^2(\omega t + \phi) \\
&= \frac{1}{2}kx_m^2[\cos^2(\omega t + \phi) + \sin^2(\omega t + \phi)] \\
E &= \frac{1}{2}kx_m^2
\end{aligned}$$

it is stated that the mechanical energy of a linear oscillator is indeed constant and independent of time.

## III. An Angular Simple Harmonic Oscillator

A torsion pendulum is an angular version of a linear simple harmonic oscillator. The disk oscillates in a horizontal plane between the angular amplitude $-\theta_m$ and $\theta_m$. The element of springiness or elasticity is associated with the twisting of a suspension wire rather than the extension and compression of a spring.

If we rotate the disk by some angular displacement $\theta$ from its rest position ($\theta = 0$) and release it, it will oscillate about that position in angular simple harmonic motion. Rotating the disk through an angle $\theta$ introduces a restoring torque given by

$$\tau = -\kappa\theta$$

with $\kappa$ is the torsion constant, that depends on the length, diameter and material of the suspension wire. If we think of it as the angular form of Hooke's law, the period for the angular simple harmonic motion will be

$$T = 2\pi\sqrt{\frac{I}{\kappa}}$$

we replace the mass from simple harmonic motion with its equivalent, $I$, the rotational inertia of the oscillating disk. The same as we replace $k$, the spring constant, with $\kappa$, the torsion constant.

## IV. Pendulums, Circular Motion

Consider a pendulum of length $L$, at one end is attached to a fixed point and free to rotate about it. A mass of $m$ is attached to the other end of the pendulum. From observations, a pendulum oscillates in a manner at least qualitatively similar to a spring-mass system.

We assume the mass $M$ is large enough so that, as an approximation, we state that all the mass is contained at the bob of the pendulum (the mass of the rigid shaft of the pendulum is assumed negligible).

- The pendulum moves in two dimensions, while the spring-mass system was constrained to move in one dimension.
- A pendulum also involves only one degree of freedom as it is constrained to move along the circumference of a circle of radius $L$.

**Figure 17.1:** *A simple pendulum with the forces acting on the pendulum' particle (called bob of the pendulum) with mass m are the gravitational force $\vec{F}_g$ and the force $\vec{T}$ from the string. The tangential component $F_g$ sin $\theta$ of the gravitational force is a restoring force that tends to bring the pendulum back to its central position.*

By applying Newton's second law of motion then develop it into polar coordinates.

$$\vec{F} = m\vec{a}$$
$$m\frac{d^2\vec{x}}{dt^2} = \vec{F}$$

where $\vec{x}$ is the position vector of the mass (the vector from the origin to the mass). In 3 dimensional rectangular coordinates, $\vec{x} = x\hat{i} + y\hat{j} + z\hat{k}$, thus the acceleration is given by

$$\frac{d^2\vec{x}}{dt^2} = \frac{d^2\vec{x}}{dt^2}\hat{i} + \frac{d^2\vec{y}}{dt^2}\hat{j} + \frac{d^2\vec{z}}{dt^2}\hat{k}$$

with $\hat{i}, \hat{j}, \hat{k}$ are unit vectors with fixed magnitude and fixed directions.

In polar coordinates (centered at the fixed vertex of the pendulum), the position vector is pointed outward with length $L$.

$$\vec{x} = L\vec{r} \tag{17.5}$$

where $\vec{r}$ is the radial unit vector. The polar angle is introduced such that $\theta = 0$ corresponds to the pendulum in its "natural" position. $L$ is constant since the pendulum does not vary in length. Thus

$$\frac{d^2\vec{x}}{dt^2} = L\frac{d^2\vec{r}}{dt^2} \tag{17.6}$$

Although the magnitude $\vec{r}$ is constant with $|\vec{r}| = 1$ (the pendulum is always moving circling), its direction varies in space. To determine the change in the radial unit vector $\vec{r}$, we express it in terms of the cartesian unit vectors,

$$\vec{r} = \sin\ \theta\ \hat{i} - \cos\ \theta\ \hat{j} \tag{17.7}$$

The $\theta$-unit vector is perpendicular to $\vec{r}$ and of unit length (and is in the direction of increasing $\theta$), thus

$$\hat{\theta} = \cos\ \theta\ \hat{i} + \sin\ \theta\ \hat{j} \tag{17.8}$$

In order to calculate the acceleration vector $\frac{d^2x}{dt^2}$, the velocity vector $\frac{d\vec{x}}{dt}$ must first be calculated

$$\begin{aligned}\vec{x} &= L\vec{r}\\ \frac{d\vec{x}}{dt} &= L\frac{d\vec{r}}{dt} + \frac{dL}{dt}\vec{r}\\ \frac{d\vec{x}}{dt} &= L\frac{d\vec{r}}{dt}\end{aligned}$$

since $L$ is a constant for a pendulum and $\frac{dL}{dt} = 0$.

With the chain rule we will obtain

$$\begin{aligned}\frac{d\hat{r}}{dt} &= \frac{d\hat{r}}{d\theta}\frac{d\theta}{dt}\\ &= \frac{d}{d\theta}\left(\sin\ \theta\ \hat{\imath} - \cos\ \theta\ \hat{\jmath}\right)\frac{d\theta}{dt}\\ &= \frac{d\theta}{dt}\left(\cos\ \theta\ \hat{\imath} + \sin\ \theta\ \hat{\jmath}\right)\\ \frac{d\hat{r}}{dt} &= \frac{d\theta}{dt}\hat{\theta}\end{aligned} \tag{17.9}$$

Now, for the change in $\theta$ with respect to time, we will use chain rule again to obtain

$$\begin{aligned}\frac{d\hat{\theta}}{dt} &= \frac{d\hat{\theta}}{d\theta}\frac{d\theta}{dt}\\ &= \frac{d}{d\theta}\left(\cos\ \theta\ \hat{\imath} + \sin\ \theta\ \hat{\jmath}\right)\frac{d\theta}{dt}\\ &= \frac{d\theta}{dt}\left(-\sin\ \theta\ \hat{\imath} + \cos\ \theta\ \hat{\jmath}\right)\\ \frac{d\hat{\theta}}{dt} &= -\frac{d\theta}{dt}\hat{r}\end{aligned} \tag{17.10}$$

Thus, the velocity vector is in the direction of $\hat{\theta}$.

$$\frac{d\vec{x}}{dt} = L\frac{d\theta}{dt}\hat{\theta}$$

The magnitude of the velocity is $L\frac{d\theta}{dt}$, if motion lies along the circumference of a circle. If $L$ is constant, then in a short length of time the position vector has changed only a little in the $\theta$ direction.

$$\frac{d\vec{x}}{dt} \approx \frac{\vec{x}(t+\triangle t) - \vec{x}(t)}{\triangle t} \approx \frac{L\triangle\theta\ \hat{\theta}}{\triangle t} \approx L\frac{d\theta}{dt}\hat{\theta}$$

The $\theta$ component of the velocity is the distance $L$ times the angular velocity $d\theta/dt$. The acceleration vector is obtained as the derivative of the velocity vector:

$$\begin{aligned}\frac{d^2\vec{x}}{dt^2} &= L\frac{d}{dt}\left(\frac{d\theta}{dt}\hat{\theta}\right)\\ &= L\left[\frac{d^2\theta}{dt^2}\hat{\theta} - \left(\frac{d\theta}{dt}\right)^2\hat{r}\right]\end{aligned}$$

**Figure 17.2:** *The direction of $\hat{r}$ and $\hat{\theta}$ vary in space, both has the magnitude of the same unit vectors.*

The angular component of the acceleration is $L\frac{d^2\theta}{dt^2}$. It exists only if the angle is accelerating. If $L$ is constant, the radial component of the acceleration, $L\left(\frac{d\theta}{dt}\right)^2$, is always directed inwards. In physics, tt is called the centripetal acceleration and will occur even if the angle is only steadily increasing (if the angular velocity $\frac{d\theta}{dt}$ is constant).

For any forces $\vec{F}$, when a mass is constrained to move in a circle, Newton's law implies

$$mL\frac{d^2\theta}{dt^2}\hat{\theta} - mL\left(\frac{d\theta}{dt}\right)^2\hat{r} = \vec{F} \tag{17.11}$$

Now, we will exammine the forces for a pendulum:

- Gravitational force, $-mg\ \hat{j}$, which can be expressed in terms of polar coordinates

$$\begin{aligned} \hat{i} &= \hat{r}\ \sin\ \theta + \hat{theta}\ \cos\ \theta \\ \hat{j} &= -\hat{r}\ \cos\ \theta + \hat{theta}\ \sin\ \theta \end{aligned} \tag{17.12}$$

  Thus, the gravitational force

$$m\vec{g} = -mg\ \hat{j} = mg\ \cos\ \theta\ \hat{r} - mg\ \sin\ \theta\ \hat{\theta}$$

- The forces on the bob of the pendulum that make it to be in motion along the circle.
  If there were no these forces, then the mass would not move along the circle. The mass is held by the rigid shaft of the pendulum, which exerts a force $T\hat{r}$ towards the origin.

  The forces on the bob of the pendulum that resulting in motion along the circle is

$$mL\frac{d^2\theta}{dt^2}\hat{\theta} - mL\left(\frac{d\theta}{dt}\right)^2\hat{r} = mg\ \cos\ \theta\ \hat{r} - mg\ \sin\ \theta\ \hat{\theta} - T\hat{r}$$

Each component of this vector force equation yields an ordinary differential equation:

$$mL\frac{d^2\theta}{dt^2} = -mg\ \sin\ \theta \tag{17.13}$$

$$-mL\left(\frac{d\theta}{dt}\right)^2 = mg\ \cos\ \theta - T \tag{17.14}$$

A two-dimensional vector equationis equivalent to two scalar equations. $T$ could be obtained from the second equation, equation **(17.14)**, after determining $\theta$ from the first equation, equation **(17.13)**. The first equation implies that the mass times the $\theta$ component of the acceleration must balance the $\theta$ component of the gravitational force.

The mass $m$ can be cancelled from both sides of the first equation. Thus, the motion of the pnedulum does not depend on the magnitude of the mass $m$ attached to the pendulum. Only varying the length,$L$, or the gravity $g$, will affect the motion.

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\ \theta \tag{17.15}$$

The pendulum is governed by a nonlinear differential equation, equation **(17.15)**, and hece is called a nonlinear pendulum. The restoring force, $mg\ \sin\ \theta$ does not depend linearly on $\theta$. Nonlinear problems are usually considerably more difficult to solve than linear ones.

Now, recall from calculus for small $\theta$,

$$\sin\ \theta \approx \theta$$

Geometrically, near the origin the functions $\sin\ \theta$ and $\theta$ are nearly identical. By using this approximation, the differential equation becomes

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta \tag{17.16}$$

this is called the linearized pendulum. This is the same type of differential equation as the one governing a linearized spring-mass system without friction. Hence, a linearized pendulum also executes simple harmonic motion. The pendulum oscillates with circular frequency

$$\omega = \sqrt{\frac{g}{L}} \tag{17.17}$$

and the period

$$T = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{L}{g}} \tag{17.18}$$

The three equations **(17.16), (17.17), (17.18)** valid only as long as $\theta$ is small. The period of oscillation is independent of amplitude. As a reminder, when we say $\theta$ it means in radians (in pi terms), not in degree.

## V. Nonlinear Pendulum

Finish Potential Energy of Physics, then we finish this along with the phase plane and direction field plot with C++. Pendulum is harder than spring-mass system, consider put it below -Sentinel

## VI. (optional?)The Equation of Motion for Simple Pendulum

A simple pendulum [3], which consists of a string of length $l$ with a point mass $m$ on the end will have the position of the pendulum specified by the angle $\theta$ between the string and the vertical line. The equation of motion of the pendulum is given by:

$$\tau = I\theta^{(2)} = I\frac{d^2\theta}{dt^2} \tag{17.19}$$

where $I$ is the moment of inertia and is given by

$$I = ml^2$$

The torque $\tau$ is given by

$$\tau = -mgl\ \sin\ \theta$$

thus

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\ \theta = 0 \tag{17.20}$$

## VII. Simulation of Simple Pendulum with Box2D

### i. Build DianFreya Modified Box2D Testbed

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Oscillations/Simple Pendulum**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class SimplePendulum : public Test
{
        public:

        SimplePendulum()
        {
                b2Body* b1;
                {
                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                            0.0f));
```

```
        b2BodyDef bd;
        b1 = m_world->CreateBody(&bd);
        b1->CreateFixture(&shape, 0.0f);
    }
    // the two blocks below for creating boxes are only for
        sweetener.
    {
        b2PolygonShape shape;
        shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, 0.7f); //
            Gradient is 0.7

        b2BodyDef bd;
        bd.position.Set(6.0f, 6.0f);
        b2Body* ground = m_world->CreateBody(&bd);
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2PolygonShape shape;
        shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, -0.7f); //
            Gradient is -0.7

        b2BodyDef bd;
        bd.position.Set(-6.0f, 6.0f);
        b2Body* ground = m_world->CreateBody(&bd);
        ground->CreateFixture(&shape, 0.0f);
    }

    b2Body* b2; // the hanging bar for the pendulum
    {
        b2PolygonShape shape;
        shape.SetAsBox(7.25f, 0.25f);

        b2BodyDef bd;
        bd.position.Set(0.0f, 37.0f);
        b2 = m_world->CreateBody(&bd);
        b2->CreateFixture(&shape, 0.0f);
    }

    b2RevoluteJointDef jd;
    b2Vec2 anchor;

    // Create the pendulum ball
    b2CircleShape ballShape;
    ballShape.m_p.SetZero();
    ballShape.m_radius = 0.5f;

    b2FixtureDef ballFixtureDef;
```

```
        ballFixtureDef.restitution = 0.75f;
        ballFixtureDef.density = 3.3f; // this will affect the ball
            mass
        ballFixtureDef.friction = 0.1f;
        ballFixtureDef.shape = &ballShape;

        b2BodyDef ballBodyDef;
        ballBodyDef.type = b2_dynamicBody;
        ballBodyDef.position.Set(0.0f, 25.0f);
        // ballBodyDef.angularDamping = 0.2f;

        m_ball = m_world->CreateBody(&ballBodyDef);
        b2Fixture *ballFixture = m_ball->CreateFixture(&
            ballFixtureDef);

        // Create the anchor and connect it to the ball
        anchor.Set(0.0f, 36.0f); // x and y axis position for the
            Pendulum anchor
        jd.Initialize(b2, m_ball, anchor);
        //jd.collideConnected = true;
        m_world->CreateJoint(&jd); // Create the Pendulum anchor
    }

    void Step(Settings& settings) override
    {
        b2Vec2 v = m_ball->GetLinearVelocity();
        float omega = m_ball->GetAngularVelocity();
        float angle = m_ball->GetAngle();
        b2MassData massData = m_ball->GetMassData();
        b2Vec2 position = m_ball->GetPosition();

        float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
            massData.I * omega * omega;

        g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
            f", position.x);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
            f", position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
            .mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
            , ke);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
            ", v);
```

```
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
                    %.6f", angle*RADTODEG);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything
                printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
                    RADTODEG);
                Test::Step(settings);
        }

        static Test* Create()
        {
                return new SimplePendulum;
        }
        b2Body* m_ball;
};

static int testIndex = RegisterTest("Oscillations", "Simple Pendulum",
    SimplePendulum::Create);
```

**C++ Code 93:** *tests/simple_pendulum.cpp "Simple Pendulum Box2D"*

Some explanations for the codes:

- To create a line below:

```
b2Body* b1;
{
        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
            0.0f));

        b2BodyDef bd;
        b1 = m_world->CreateBody(&bd);
        b1->CreateFixture(&shape, 0.0f);
}
```

- Create a Revolute Joint named **jd** that can revolve

```
b2RevoluteJointDef jd;
b2Vec2 anchor;
```

- Create the pendulum ball, set its' position, friction, restitution, radius, density, angular damping (so the pendulum will eventually stop instead of forever oscillating) with shape of a circle, but we call it **ballShape**.

```
// Create the pendulum ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;
```

```
b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
    );
```

- Now after the ball for the pendulum is created we can connect it to the anchor with **CreateJoint();**

```
// Create the anchor and connect it to the ball
anchor.Set(0.0f, 36.0f); // x and y axis position for the
    Pendulum anchor
jd.Initialize(b2, m_ball, anchor);
//jd.collideConnected = true;
m_world->CreateJoint(&jd); // Create the Pendulum anchor
```

**Figure 17.3:** *The simple pendulum simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/simple_pendulum.cpp****).*

- To print the output of the $x$ position, $y$ position and the angle (in degrees) of the pendulum into the terminal / xterm

```
            printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
                RADTODEG);
```

After recompiling this, you can save it into textfile by opening the testbed with this command:
**./testbed > /root/output.txt**
you may drag the pendulum to certain degree and let it swing for 5 periods or around that, afterwards close the testbed to record the result, then see it at **/root/output.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 3 columns.

To plot it you can use gnuplot, as it is very handy when we have raw data, then need to plot it fast, easy, and powerful even for 3-dimensional surface plot. Open terminal from the directory that contain the "output.txt" and type:

**gnuplot**
**plot "output.txt" using 1 title "x_{m}" with lines**

you can also try:
**plot "output.txt" using 1 title "x_{m}" with lines, "output.txt" using 3 title "angle"**

**Figure 17.4:** *The "output.txt" shall only contains 3 columns of numerical data.*

Box2D is an amazing Physics Engine library that can be used not only to create game but for learning all kinds of science, since we are all bounds to Physics even atoms and quantum are still kneel to the Physics. It won't be too long to gain mastery over learning this for anyone who want to invest their time, interest and disk space of their brain.

**Figure 17.5:** *The "output.txt" is being plotted by using gnuplot for the data of the $x_m$ (the displacement) with respect to time.*

**Figure 17.6:** *The "output.txt" is being plotted by using gnuplot for the data of the $x_m$ (the displacement) and the angle of the pendulum with respect to time. We may see that the angle and the displacement of the pendulum have a linear relationship.*

## VIII. Horizontal Spring-Mass System

When we observe a spring-mass system, once it is set in motion the object with mass moves back and forth (oscillates) [5]. This simple spring-mass sytem exhibits behavior of complex system resemble the motions of clock-like mechanisms and aid in the understanding of the up-and-down motion of the ocean surface. Remember the Newton's second law of motion that describing how a

**Figure 17.7:** *The spring-mass system at rest with $x = 0$ as the position of the mass on the $x$ axis, and $l_0$ is the original length of the spring, it is called the equilibrium or unstretched position of the spring.*

particle reacts to a force

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2}$$

Easily remembered as "$F$ equals $ma$."

The distance $x$ is then referred to as the displacement from equilibrium or the amount of stretching of the spring.

- If we stretch the spring (let $x > 0$), then the spring exerts a force pulling the mass back towards the equilibrium position (that is $F < 0$). As we increase the stretching of the spring, the force exerted by the spring would increase. For stretched spring, we will have this relation: $x_1 < x_2$, hence $F_2 < F_1$.
- If the spring is contracted ($x < 0$), then the spring pushes the mass again towards the equilibrium position ($F > 0$). For conracted spring, we will have this relation: $x_1 < x_2$, hence $F_1 < F_2$.

Such a force, $F$, is called the restoring force. We have assumed that the force only depends on the amount of stretching of the spring, not on any other quantities.

On the seventeenth century, physicist Hooke approximate the relationship between the force and the stretching of the spring with a straight line, thus Hooke's law is

$$F = -kx \tag{17.21}$$

with $k$ as the spring constant. Higher positive force is required to get the mass to its equilibrium position after being contracted deeper. Bigger negative force is required to get the mass to its equilibrium position after being stretched longer.

Using Hookes' law, Newton's second law of motion yields

$$m\frac{d^2x}{dt^2} = -kx \tag{17.22}$$

as the simplest mathematical model of a spring-mass system.

## IX. Simulation of Horizontal Spring-Mass System with Box2D

We are going to simulate the horizontal spring-mass system at rest then be given force toward positive $x$ axis, and toward negative $x$ axis.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringTest : public Test
{
        public:
        SpringTest()
        {
                b2Body* ground = NULL;
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                            0.0f));
                        ground->CreateFixture(&shape, 0.0f);
                }
                // Create a static body as the box for the spring
                b2BodyDef bd1;
                bd1.type = b2_staticBody;
                bd1.angularDamping = 0.1f;

                bd1.position.Set(1.0f, 0.5f);
                b2Body* springbox = m_world->CreateBody(&bd1);

                b2PolygonShape shape1;
                shape1.SetAsBox(0.5f, 0.5f);
                springbox->CreateFixture(&shape1, 5.0f);

                // Create the box as the movable object
                b2PolygonShape boxShape;
                boxShape.SetAsBox(0.5f, 0.5f);

                b2FixtureDef boxFixtureDef;
                boxFixtureDef.restitution = 0.75f;
                boxFixtureDef.density = 7.3f; // this will affect the box
                    mass
                boxFixtureDef.friction = 0.1f;
                boxFixtureDef.shape = &boxShape;

                b2BodyDef boxBodyDef;
```

```
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(5.0f, 0.5f);

        m_box = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
            ;
        //m_box->SetGravityScale(-7); // negative means it will goes
             upward, positive it will goes downward
        // Make a distance joint for the box / ball with the static
            box above
        m_hertz = 1.0f;
        m_dampingRatio = 0.1f;

        b2DistanceJointDef jd;
        jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
            boxBodyDef.position);
        jd.collideConnected = true; // In this case we decide to
            allow the bodies to collide.
        m_length = jd.length;
        m_minLength = 2.0f;
        m_maxLength = 10.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
            m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
            //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
            m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f),
                true);
            break;
```

```
                        case GLFW_KEY_S:
                        //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f));
                        m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f),
                            true);
                        break;
                        case GLFW_KEY_D:
                        m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f), true)
                            ;
                        break;
                        case GLFW_KEY_F:
                        m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f), true)
                            ;
                        break;
                        case GLFW_KEY_G:
                        m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f), true
                            );
                        break;
                }
        }
        void UpdateUI() override
        {
                ImGui::SetNextWindowPos(ImVec2(10.0f, 100.0f));
                ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
                ImGui::Begin("Joint Controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

                if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                    f"))
                {
                        m_length = m_joint->SetLength(m_length);
                }

                if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                    ))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                    , 2.0f, "%.1f"))
                {
                        float stiffness;
```

```
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                ImGui::End();
        }
        void Step(Settings& settings) override
        {
                b2MassData massData = m_box->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                b2Vec2 velocity = m_box->GetLinearVelocity();
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz

                g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
                    apply different force to the box");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass position = (%4.1f
                    , %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass velocity = (%4.1f
                    , %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                    .mass);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f\n", position.x);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new SpringTest;
        }

};

static int testIndex = RegisterTest("Oscillations", "Spring Test",
```

```
        SpringTest::Create);
```

**C++ Code 94:** *tests/spring_test.cpp "Horizontal Spring Mass System Box2D"*

Some explanations for the codes:

- To create the static body as the wall block that will be attached to the spring

```
            b2BodyDef bd1;
            bd1.type = b2_staticBody;
            bd1.angularDamping = 0.1f;

            bd1.position.Set(1.0f, 0.5f);
            b2Body* springbox = m_world->CreateBody(&bd1);

            b2PolygonShape shape1;
            shape1.SetAsBox(0.5f, 0.5f);
            springbox->CreateFixture(&shape1, 5.0f);
```

- To create the "hallucination" spring in Box2D we can use **DistanceJoint** that is joining the static body as the wall block and the dynamic body as the mass / a movable object.

```
            m_hertz = 1.0f;
            m_dampingRatio = 0.1f;

            b2DistanceJointDef jd;
            jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f), boxBodyDef.
                position);
            jd.collideConnected = true; // In this case we decide to allow
                the bodies to collide.
            m_length = jd.length;
            m_minLength = 2.0f;
            m_maxLength = 10.0f;
            b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
                m_dampingRatio, jd.bodyA, jd.bodyB);

            m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
            m_joint->SetMinLength(m_minLength);
            m_joint->SetMaxLength(m_maxLength);
```

  Distance joint is one of the simplest joint, we can make it work like a spring, it makes the distance between two bodies must be constant.
- The keyboard press events, the function **ApplyForceToCenter** with negative force means it will direct the force toward the negative $x$ axis, while positive force means it will direct the force toward the positive $x$ axis. The coding in Box2D is quite the opposite of the theory when we stretch the spring ($x > 0$) thus the force to pull the mass back to equilibrium position will be negative, that is $F < 0$. It should be understood, so all the confusions shall be cleared up, as in Box2D, **ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f), true);** means applying force of 8000 $N$ to the box toward the negative $x$ axis, it has nothing to do with the force to bring the mass to equilibrium position.

```
void Keyboard(int key) override
{
        switch (key)
        {
                case GLFW_KEY_A:
                //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
                m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f)
                    , true);
                break;
                case GLFW_KEY_S:
                //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f))
                    ;
                m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f)
                    , true);
                break;
                case GLFW_KEY_D:
                m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f),
                     true);
                break;
        }
}
```

- A feature from ImGui at the top left, we can change the length for the joint / the "hallucination" spring, the damping ratio that will determine the speed of convergence to the equilibrium position, and the frequency / Hertz parameter, the higher the Hertz, the faster the movement that our eyes see, it will looks like it suddenly finish.

```
void UpdateUI() override
{
        ImGui::SetNextWindowPos(ImVec2(10.0f, 100.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
            ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize)
            ;

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f,
             "%.0f"))
        {
                m_length = m_joint->SetLength(m_length);
        }
```

```
                if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "
                    %.1f"))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint
                            ->GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio,
                     0.0f, 2.0f, "%.1f"))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint
                            ->GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                ImGui::End();
        }
```

After recompiling this, you can save it into textfile by opening the testbed with this command: **./testbed > spring.txt**
you may press "G" or "F" or any other key or do mouse click to drag the box and let it oscillates, afterwards close the testbed to record the result, then see it at **../(current working directory)/spring.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 1 column.

To plot it you can use gnuplot. Open terminal from the directory that contain the "spring.txt" and type:

**gnuplot**
**set xlabel "t"**
**plot "spring.txt" using 1 title "x (t)" with lines**

**Figure 17.8:** *The horizontal spring-mass system simulation , besides keyboard press events you can click the box with mouse and drag it toward positive or negative x axis to see it oscillating till it rests at equilibrium position ($x = 5, y = 0.5$) again (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_test.cpp****).*

**Figure 17.9:** *The gnuplot of a horizontal spring-mass system, with friction* 0.1 *and damping ratio of* 0.1*, mass of* $m = 7.3$*, and the rest position at* $x = 5, y = 0.5$ *after being pushed toward positive x axis with a force of* $F = 10,000$*.*

**Figure 17.10:** *The gnuplot of a horizontal spring-mass system, friction* 0 *and damping ratio of* 0*, mass of* $m = 7.3$*, and the rest position at* $x = 5, y = 0.5$ *after being pushed toward positive* $x$ *axis with a force of* $F = 10,000$*.*

## X. Vertical Spring-Mass System

Now, consider a vertical spring-mass system, which the derivation of the equation governing the horizontal spring-mass system does not apply to the vertical system. There is another force, gravity, that will be approximated as a constant $mg$, or the weight. Hence the Newton's law for the vertical system becomes

$$m\frac{d^2y}{dt^2} = -ky - mg \tag{17.23}$$

where $y$ is the vertical coordinate. $y = 0$ is the position at which the spring exerts no force. The

**Figure 17.11:** *Gravitational effect on spring-mass equilibrium for vertical spring. The spring sags downwards a distance* $\frac{mg}{k}$ *when the mass is added. The stiffer the spring (k larger), the smaller the sag of the spring.*

force at which we could place the mass and it would not move will be called an equilibrium

position. It follows that

$$\begin{aligned}\sum F_y &= ma_y \\ -ky - mg &= m(0) \\ -ky &= mg \\ y &= -\frac{m}{k}g\end{aligned}$$

where $y = -\frac{m}{k}g$ is the equilibrium position of this spring-mass-gravity system. Only at that position will the force due to gravity balance the upward force of the spring. The negative sign is to represent the opposite direction between the spring' force and the gravity force toward the mass. In some cases, we can also use $\triangle y$ instead of negative sign, since it means displacement from the equilibrium position

$$k\triangle y = mg$$

To translate coordinate systems from one with an origin at $y = 0$ (the position of the unstretched spring) to one with an origin at $y = -\frac{mg}{k}$ (the equilibrium position with the mass). Let $Z$ to be the variable representing the displacement from this equilibrium position:

$$\begin{aligned}Z &= y - \left(-\frac{mg}{k}\right) \\ &= y + \frac{mg}{k}\end{aligned}$$

$Z$ here for the vertical spring-mass system is replacing $x$ for the horizontal spring-mass system, thus

$$m\frac{d^2Z}{dt^2} = -kZ \tag{17.24}$$

## XI. Simulation of Vertical Spring-Mass System with Box2D

Taken from problem no 88 from [6]. I modify some parameters a bit.

A block weighing 200 $N$ oscillates at one end of a vertical spring for which $k = 100\ N/m$; the other end of the spring is attached to a ceiling. At a certain instant the spring is stretched 3.0 $m$ beyond its relaxed length (the length when no object is attached) and the block has zero velocity.

(a) What is the net force on the block at this instant?

(b) What is the amplitude?

(c) What is the period of the resulting simple harmonic motion?

(d) What is the maximum kinetic energy of the block as it oscillates?

**Solution:**

(a) The Hooke's law force will have opposite direction with the weight' force, we choose the positive direction to be upward thus

$$\begin{aligned}\sum F &= kx - mg \\ &= 100(3.0) - 200 \\ &= 100\end{aligned}$$

the net force is 100 $N$ upward.

(b) The equilibrium position is where the upward Hooke's law force balances the weight, which corresponds to the spring being stretched (from unstretched length) by

$$y = \frac{mg}{k} = \frac{200}{100} = 2$$

Thus, relative to the equilibrium position, the block(at the instant) is the bottom turning point (when $v = 0$) at $x = -x_m$, where the amplitude is

$$x_m = 3.0 - 2.0 = 1.0$$

the amplitude is 1.0 $m$. How to plot this amplitude is explained below along with the C++ code to create the simulation for this problem.

(c) We know that

$$m = \frac{W}{g} = \frac{200}{9.8} \approx 20$$

the mass is 20 $kg$, thus

$$T = 2\pi\sqrt{\frac{m}{k}} = 2\pi\sqrt{\frac{20}{100}} = 2.81$$

the period is 2.81 $s$.

(d) The maximum kinetic energy is equal to the maximum potential energy $\frac{1}{2}kx_m^2$. Thus,

$$\begin{aligned} K_m &= U_m \\ &= \frac{1}{2}(100)(1.0) \\ &= 50 \end{aligned}$$

the maximum kinetic energy is 50 $J$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class VerticalspringTest : public Test
{
        public:
        VerticalspringTest()
        {
                b2Body* ground = NULL;
                m_world->SetGravity(b2Vec2(0.0f,-9.8f));
                {
                        b2BodyDef bd;
                        ground = m_world->CreateBody(&bd);

                        b2EdgeShape shape;
                        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                            0.0f));
```

```
        ground->CreateFixture(&shape, 0.0f);
}
// Create a static body as the box for the spring
b2BodyDef bd1;
bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 23.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 200.0f/9.8f; // this will affect the
    box mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(1.0f, 19.5f); // the box will be
    located in (1,19.5), 2.0 m beyond the spring relaxed
    length. It is the equilibrium position where ky = mg

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
    ;
//m_box->SetGravityScale(-7); // negative means it will goes
     upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 23.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
```

```
                m_minLength
        m_maxLength = 14.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
            m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_W:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, 10000.0f), true
                );
            break;
            case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, -9500.0f),
                true);
            break;
            case GLFW_KEY_T:
            m_time = 0.0f;
            break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
            ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
            f"))
        {
            m_length = m_joint->SetLength(m_length);
```

```
                }

                if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                    ))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                    , 2.0f, "%.1f"))
                {
                        float stiffness;
                        float damping;
                        b2LinearStiffness(stiffness, damping, m_hertz,
                            m_dampingRatio, m_joint->GetBodyA(), m_joint->
                            GetBodyB());
                        m_joint->SetStiffness(stiffness);
                        m_joint->SetDamping(damping);
                }

                ImGui::End();
        }
        void Step(Settings& settings) override
        {
                b2MassData massData = m_box->GetMassData();
                b2Vec2 position = m_box->GetPosition();
                b2Vec2 velocity = m_box->GetLinearVelocity();
                m_time += 1.0f / 60.0f; // assuming we are using frequency of
                     60 Hertz
                float k;
                float m = massData.mass;
                float g = 9.8;
                float y = 23.5 - m_minLength - 19.5;
                // y = the position at which when we place the mass it would
                    not move / equlibrium position
                // y = y position of the ceiling - m_minLength - initial y
                    position of the mass
                k = (m*g)/y;
                float y_eq;
                y_eq = -m*g/k;

                g_debugDraw.DrawString(5, m_textLine, "Press W to apply force
```

```
                    10N upward / S to apply force 10 N downward");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
                    );
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass position = (%4.1f
                    , %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass velocity = (%4.1f
                    , %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                    .mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
                     = %4.1f", k);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "The position at which
                    the spring exerts no force / mass hanging, y = 0");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
                    for the mass, y = -mg/k= %4.1f", y_eq);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f\n", position.y);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new VerticalspringTest;
        }

    };

    static int testIndex = RegisterTest("Oscillations", "Vertical Spring Test",
         VerticalspringTest::Create);
```

**C++ Code 95:** *tests/verticalspring_test.cpp "Simple Vertical Spring Box2D"*

Some explanations for the codes:

- It is said that distance joint can be used as a spring. Thus to create a distance joint that can act like a spring / "hallucination" spring, which does not look like a spring, but acts like a

spring. We set the **m_minLength** to be 2, since the problem stated that the relaxed length is 2 $m$ for the spring (the length when nothing is attached on the spring).

```
// Make a distance joint for the box / ball with the static box
     above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 23.5f), boxBodyDef
    .position);
jd.collideConnected = true; // In this case we decide to allow
    the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
    m_minLength
m_maxLength = 14.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);
```

The distance joint can also be made soft, like a spring-damper connection. Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than a half the frequency of the time step. So if you are using a 60Hz time step, the frequency of the distance joint should be less than 30Hz. The reason is related to the Nyquist frequency.

The damping ratio is non-dimensional and is typically between 0 and 1, but can be larger. At 1, the damping is critical (all oscillations should vanish).

```
jointDef.frequencyHz = 4.0f;
jointDef.dampingRatio = 0.5f;
b2LinearStiffness(jointDef.stiffness, jointDef.damping,
    frequencyHz, dampingRatio, jointDef.bodyA, jointDef.bodyB);
```

To plot the $x(t)$ with respect to time and see the amplitude, we need to plot the $y$ position after we push the mass downward with a force, I choose a force of 9500 $N$, that you can apply when you press "S".

Now, at the working directory, open the testbed with this command:
**./testbed > verticalspring.txt**
then see it at **(current working directory)/verticalspring.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in column format.

To plot it you can use gnuplot. Open terminal from the directory that contain the "vertical-spring.txt" and type:

```
gnuplot
set xlabel "time"
plot "verticalspring.txt" using 1 title "x(t)" with lines
```

**Figure 17.12:** *The amplitude of this problem of vertical spring-mass system is* 1 *m, the equilibrium position is at* $y = 19.5$ *but the mass fell a bit lower to 19.2 due to gravity, the relaxed length of the spring is at* $y = 21.5$. *When we stretched the mass downward for 1 meter then it will oscillates till it comes back to its equilibrium position again. The x axis represents the time in* $1/60$ *seconds*

**Figure 17.13:** *The vertical spring-mass system simulation for this current problem (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/verticalspring_test.cpp****).*

## XII. Oscillation of a Spring-Mass System

Oscillations resides as chapter 15 for Fundamental of Physics book [6], this explains why we are going to examine the second order differential equation that governs this spring-mass system. The differential equation describing a spring mass sytem is

$$m\frac{d^2x}{dt^2} = -kx$$

with the restoring force is proportional to the stretching of the spring. A differential equation that differentiating the dependent variable twice is called second-order differential equation($x$ is the dependent variable, while $t$ is an independent variable), it is homogeneous and linear differential equation, you can read [1] to know how to solve that differential equation. The general solution for the second order differential equation is

$$x = c_1 \cos\ \omega t + c_2 \sin\ \omega t \tag{17.25}$$

with $c_1$ and $c_2$ are constants that can be determined with initial values or conditions, where

$$\omega^2 = \frac{k}{m}$$

This second-order ordinary differential equation has the general form

$$\frac{d^2x}{dt^2} = f\left(t, x, \frac{dx}{dt}\right)$$

where $f$ is some given function, in spring-mass system case $f = -\frac{kx}{m}$.

The initial conditions for second-order differential equations of the spring-mass system are usually in the form of these

$$x(0) = x_0$$
$$x'(0) = x'_0$$

where $x_0$ and $x'_0$ are constants that represent the value of $x$ at time 0, and the value of $x' = \frac{dx}{dt} = v$ at time 0, respectively.

The general solution of a second-order linear homogeneous differential equation is a linear combination of two homogeneous solutions. For constant coefficient differential equations, the homogeneous solutions are usually in the form of simple exponentials, $e^{rt}$. We will explore about this, now let's go back to the differential equation:

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x$$

we see that $\frac{k}{m}$ can be regarded as constant, where $\omega^2 = \frac{k}{m}$, now what is the function that the second derivative is itself? None other than exponential function, we derive $e^x$ with respect to $x$ for thousand of times it will still be $e^x$, it is amazing as it is a transcendental function and I spend more than 1 month just to summarize that chapter in Calculus. The application for exponential function is enormous as compound interest uses exponential as well.

Back to the differential equation, with $x = x(t)$, $\frac{dx}{dt} = x'(t)$, $\frac{d^2x}{dt^2} = x''(t)$ we will get

$$x'' = -\omega^2 x$$
$$x'' + \omega^2 x = 0$$

we can see it in the form of $ax'' + bx' + cx = 0$ with $a = 1, b = 0, c = \omega^2$, now if we seek solutions of the form $x(t) = e^{rt}$, based on [1], then $r$ must be a root of the characteristic equation, the characteristic equation for that is

$$r^2 + \omega^2 = 0$$
$$r^2 = -\omega^2$$
$$r = \sqrt{-\omega^2}$$
$$r = \pm i\omega$$

so its roots are

$$r_1 = i\omega, \quad r_2 = -i\omega$$

if the roots $r_1$ and $r_2$ are conjugate complex numbers $r_{1,2} = \pm i\omega$, which occurs whenever the discriminant $b^2 - 4ac$ of the $ax'' + bx' + cx = 0$ is negative $(0 - ((4)(1)\omega^2) < 0)$, then the general solution is

$$x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$$
$$x(t) = c_1 e^{i\omega t} + c_2 e^{-i\omega t}$$

with $c_1$ and $c_2$ are constants. We know that from the Euler 'formula,

$$e^{i\omega t} = \cos(\omega t) + i\ \sin(\omega t)$$
$$e^{-i\omega t} = \cos(\omega t) - i\ \sin(\omega t)$$

thus

$$\begin{aligned}
x(t) &= c_1 e^{i\omega t} + c_2 e^{-i\omega t} \\
&= c_1 \left(\cos(\omega t) + i\ \sin(\omega t)\right) + c_2 \left(\cos(\omega t) - i\ \sin(\omega t)\right) \\
&= (c_1 + c_2)\cos(\omega t) + i(c_1 - c_2)\sin(\omega t) \\
x(t) &= d_1 \cos(\omega t) + d_2 \sin(\omega t)
\end{aligned}$$

with the constants $d_1$ and $d_2$ defined by

$$\begin{aligned}
d_1 &= c_1 + c_2 \\
d_2 &= i(c_1 - c_2)
\end{aligned}$$

The constants $d_1$ and $d_2$ are arbitrary since given any value of $d_1$ and $d_2$, there exists values of $c_1$ and $c_2$, namely

$$\begin{aligned}
c_1 &= \frac{1}{2}(d_1 - id_2) \\
c_2 &= \frac{1}{2}(d_1 + id_2)
\end{aligned}$$

**Definition 17.1: Equivalent of Linear Combination for $\sin(\omega t)$ and $\cos(\omega t)$**

An arbitrary linear combination of $e^{i\omega t}$ and $e^{-i\omega t}$,

$$x(t) = c_1 e^{i\omega t} + c_2 e^{-i\omega t}$$

is equivalent to an arbitrary linear combination of $\cos\ \omega t$ and $\sin\ \omega t$,

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$$

with

$$\begin{aligned}
d_1 &= c_1 + c_2 \\
d_2 &= i(c_1 - c_2)
\end{aligned}$$

$$\begin{aligned}
c_1 &= \frac{1}{2}(d_1 - id_2) \\
c_2 &= \frac{1}{2}(d_1 + id_2)
\end{aligned}$$

and the Euler 'formula,

$$\begin{aligned}
e^{i\omega t} &= \cos(\omega t) + i\ \sin(\omega t) \\
e^{-i\omega t} &= \cos(\omega t) - i\ \sin(\omega t)
\end{aligned}$$

After long deriving the formula, we should now be able to state that the general solution of

$$m\frac{d^2x}{dt^2} = -kx$$

is

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$$

where $\omega = \sqrt{\frac{k}{m}}$ and $d_1$ and $d_2$ are arbitrary constants. The general solution is a linear combination of two oscillatory functions, a cosine and a sine. An equivalent expression for the solution is

$$x(t) = A\sin(\omega t + \phi_0) \tag{17.26}$$

This is shown by noting

$$\sin(\omega t + \phi_0) = \sin\ \omega t\ \cos\ \phi_0 + \cos\ \omega t\ \sin\ \phi_0$$

in which case This is shown by noting

$$d_1 = A\ \sin\phi_0$$
$$d_2 = A\ \cos\phi_0$$

If you are given $d_1$ and $d_2$, you can determine both $A$ and $\phi_0$, and using some trigonometric properties $\sin^2\phi_0 + \cos^2\phi_0 = 1$ results in an equation for $A^2$

$$A = (d_1^2 + d_2^2)^{1/2}$$
$$\phi_0 = \tan^{-1}\left(\frac{d_1}{d_2}\right)$$

The expression, $x(t) = A\sin(\omega t + \phi_0)$, is expecially convenient for sketching the displacement as a function of time. It shows that the sum of any multiple of cos $\omega t$ plus any multitude of sin $\omega\ t$ is itself a sinusoidal function.

**Figure 17.14:** *The period and amplitude of spring-mass system oscillation.*

- $A$ is called the amplitude of the oscillation, it can be easily computed from the above equation if $d_1$ and $d_2$ are known.
- The phase of oscillation is $\omega t + \phi_0$, with $\phi_0$ being the phase at $t = 0$.
- The mass after reaching its maximum displacement ($x$ largest), returns to the same position $T$ units of time later.
- The entire oscillation repeats itself every $T$ units of time, $T$ is called the period of oscillation.
- Larger $k$ will make shorter period of $T$.

This motion is referred to as simple harmonic motion. The mass oscillates sinusoidally around the equilibrium position $x = 0$. The solution is periodic in time.

Mathematically, a function $f(t)$ is said to be periodic with period $T$ if

$$f(t + T) = f(t)$$

To determine the period $T$, we recall that the trigonometric functions are periodic with period $2\pi$. Thus, for a complete oscillation, as $t$ increases to $t + T$

$$[(\omega t + T) + \phi_0] - [(\omega t + \phi_0)] = 2\pi$$

Consequently the period is

$$T = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}} \tag{17.27}$$

with $\omega$ as the circular frequency, or the number of periods in $2\pi$ units of time:

$$\omega = \frac{2\pi}{T} = \sqrt{\frac{k}{m}} \tag{17.28}$$

The number of oscillations in one unit of time is the frequency $f$,

$$f = \frac{1}{T} = \frac{\omega}{2\pi} = \frac{1}{2\pi\sqrt{\frac{k}{m}}}$$

with $f$ is measured in cycles per second (Hertz). Since a spring-mass system normally oscillates with frequency $\frac{1}{2\pi}\sqrt{\frac{k}{m}}$, this value is referred to as the natural frequency of a spring-mass system of mass $m$ and spring constant $k$. Other physical systems have natural frequencies of oscillation, like a vibrating string, you might love to learn about wave equation, it requires knowledge in partial differential equation.

Now, we are going to dig onto the initial value problem, remember that

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t) \tag{17.29}$$

is the general solution of the differential equation describing a spring-mass system.

$$m\frac{d^2x}{dt^2} = -kx \tag{17.30}$$

where $d_1$ and $d_2$ are arbitrary constants and $\omega = \sqrt{\frac{k}{m}}$. We can determine the constants $d_1$ and $d_2$ from the initial conditions of the spring-mass system. In fact for a spring-mass system, the two parameters $k$ and $m$ are not important, it is their ratio $k/m$ that is important.

One way to initiate motion in a spring-mass system is to strike the mass, or to pull/push the mass to some position $x_0$ and then let go, thus

$$x(0) = x_0$$

and at $t = 0$, the velocity of the mass, $\frac{dx}{dt}$, is zero,

$$\frac{dx}{dt}(0) = 0$$

Thus, we have two initial conditions:

$$x(0) = x_0$$
$$x'(0) = 0$$

Two initial conditions are necessary since the differential equation involves the second derivative in time. To sove this initial value problem, the arbitrary constants $d_1$ and $d_2$ are determined so the equation $x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$ satisfies the initial conditions.

To solve the initial value problem above:

$$\begin{aligned} x(t) &= d_1 \cos(\omega t) + d_2 \sin(\omega t) \\ x(0) &= x_0 \\ d_1 \cos(\omega(0)) + d_2 \sin(\omega(0)) &= x_0 \\ d_1 &= x_0 \end{aligned}$$

$$\begin{aligned} \frac{dx(t)}{dt} &= -d_1\omega \ \sin(\omega t) + d_2\omega \ \cos(\omega t) \\ x'(t) &= -d_1\omega \ \sin(\omega t) + d_2\omega \ \cos(\omega t) \\ x'(0) &= -d_1\omega \ \sin(\omega(0)) + d_2\omega \ \cos(\omega(0)) \\ d_2\omega &= 0 \\ d_2 &= 0 \end{aligned}$$

We will obtain the solution from the initial value problem:

$$x(t) = x_0 \cos \ \omega t \tag{17.31}$$

we can also use another initial conditions, where $v(0) = v_0 \neq 0$, where there is an initial velocity when the mass is being let go after stretched / contracted.

## XIII. A Two-Massses Oscillator

The force of a spring-mass system can be approximated as being simply proportional to the stretching of the spring. Now, instead of attaching a spring and a mass to a rigid wall, we attach a spring and a mass to another mass, which is also free to move. Let us assume that the two masses are $m_1$ and $m_2$, while the connecting spring is known to have an unstretched length $l$ and spring constant $k$.

We must formulate Newton's law of motion for each mass. The force on each mass equals its mass times its acceleration. To obtain acceleration we will need the position of each mass, $x_1$ and $x_2$ are the distances each mass is from a fixed origin.

**Figure 17.15:** *The illustration for two-masses-spring system.*

Although the unstretched length of the spring is $l$, it is not necessary that $x_2 - x_1 = l$, for in many circumstances the spring may be stretched or compressed.

We may impose initial conditions such that initially

$$x_2 - x_1 \neq l$$

Now, from Newton's law of motion, it follows that if $F_1$ is the force on mass $m_1$ and if $F_2$ is the force on mass $m_2$, then

$$m_1 \frac{d^2 x_1}{dt^2} = F_1$$
$$m_2 \frac{d^2 x_2}{dt^2} = F_2$$

To complete the derivation of the equations of motion, we must determine the two forces, $F_1$ and $F_2$.

The only force on each mass is due to the spring. Each force is an application of Hooke's law; the force is proportional to the stretching of the spring. The stretching of the spring is the length of the spring $x_2 - x_1$ minus the unstretched length $l$, hence

$$x_2 - x_1 - l$$

The magnitude of the force is just the spring constant $k$ times the stretching, and the direction of the force should be carefully analyzed.

- If the spring is stretched $(x_2 - x_1 - l > 0)$, then the mass $m_1$ is being pulled to the right.
- If the spring is contracted $(x_2 - x_1 - l < 0)$, then the mass $m_1$ is being pushed to the left.

The force on mass $m_1$:

$$m_1 \frac{d^2 x_1}{dt^2} = k(x_2 - x_1 - l) \tag{17.32}$$

$$m_2 \frac{d^2 x_2}{dt^2} = -k(x_2 - x_1 - l) \tag{17.33}$$

Although the amgnitude of the force on $m_2$ is the same as $m_1$ the spring force acts in the opposite direction.

Now, we are going to combine two coupled second order ordinary differential equations involving two unknowns $x_1$ and $x_2$.

$$m_1 \frac{d^2 x_1}{dt^2} + m_2 \frac{d^2 x_2}{dt^2} = 0$$
$$\frac{d^2}{dt^2}(m_1 x_1 + m_2 x_2)$$

thus, we will have the center of mass of the system:

$$\frac{m_1 x_1 + m_2 x_2}{m_1 + m_2}$$

the center of mass does not accelerate, but moves at a constant velocity (determined from initial conditions). If we view the system of two masses and a spring as a single entity, then there are no external forces on it, the system obeys Newton's first law and will not accelerate. When there is no acceleration, the system of two masses moves at a constant velocity.

Now, let $z$ be the stretching of the spring between two masses,

$$\frac{d^2}{dt^2}(x_2 - x_1) = -\frac{k}{m_2}(x_2 - x_1 - l) - \frac{k}{m_1}(x_2 - x_1 - l)$$

$$z = x_2 - x_1 - l$$

$$\frac{d^2 z}{dt^2} = x_2 - x_1$$

thus

$$\frac{d^2 z}{dt^2}(x_2 - x_1) = -k\left(\frac{1}{m_1} + \frac{1}{m_2}\right) z$$

We see that the stretching of the spring executes simple harmonic motion. The circular frequency is $\sqrt{k\left(\frac{1}{m_1} + \frac{1}{m_2}\right)}$.

The two-masses spring system is the same type of oscilaltion if certain mass $m$ were placed on the same spring

$$\frac{1}{m} = \frac{1}{m_1} + \frac{1}{m_2} \tag{17.34}$$

This mass $m$ is less than either $m_1$ or $m_2$, and can be called reduced mass with

$$m = \frac{m_1 m_2}{m_1 + m_2} \tag{17.35}$$

Attaching a spring-mass system to a movable mass reduces the effective mass. The stretching of the spring executes simple harmonic motion as though a smaller mass was attached, the entire system may move, i.e., the center of mass moves at a constant velocity.

## XIV. Simulation of Two Masses Spring System with Box2D

This example is taken from exercise 9.7 "Mechanical Vibrations" chapter from [5].

Consider a mass $m_1$ attached to a spring (of unstretched length $d$) and pulled by a constant force $F_2 = m_2 g$.

(a) Suppose that the system is in equilibrium when $x = L$. Is $L > d$ or is $L < d$ ? If $L$ and $d$ are known, what is the spring constant $k$?

(b) If the system is at rest in the position $x = L$ and the mass $m_2$ is suddenly removed (for example, by cutting the string connecting $m_1$ and $m_2$), then what is the period and amplitude of oscillation of $m_1$?

**Solution:**

(a) First, we need to use Hooke's law, that is

$$F = -kx$$

with $F$ is the force of the string pulling / pushing the mass back to its' rest position. This force is always have opposite direction with the force that pull or push the mass away form

**Figure 17.16:** *The two masses spring-mass system illustration, when there is no mass $m_2$, the length of unstretched spring connected to mass $m_1$ is $d$.*

its' rest / equilibrium position.

The equilibrium position for this system is where the leftward Hooke's law balances the force from the mass $m_2$, which is $F = m_2 g$, which corresponds to the spring being stretched to $L$ distance.

For $m_1$ when at rest, we will have

$$F_1 = -k(\triangle x) = -k(x_1 - d)$$

with $x_1$ is the displacement of the mass $m_1$ from $d$,the unstretched length of the spring that is attached to $m_1$. We can assing the number such as $d = 0$, hence a pull toward $m_1$ will make $x_1 > 0$, a push toward $m_1$ will make $x_1 < 0$

Adding another mass, $m_2$ that pull the mass $m_1$, then we know that there will be a pull of mass $m_1$ from mass $m_2$ that makes the spring being stretched to $x = L$, by using Newton's second law for equilibrium system

$$\begin{aligned} \sum F_1 &= m_1 a \\ T - kx &= m_1 a \\ T &= m_1 a + kx \end{aligned}$$

The cord tension $T$ is positive, since it is having the same direction as the acceleration, $a$, while $-kx$ the spring force is negative since it is a force that will pull the mass $m_1$ back to its' equilibrium position, and has opposite direction of the acceleration, $a$.

$$\begin{aligned} \sum F_2 &= m_2 a \\ m_2 g - T &= m_2 a \\ T &= m_2 g - m_2 a \end{aligned}$$

The weight force due to gravity is positive due to the same direction with the acceleration, $a$, while the tension on the cord $T$ has the opposite direction toward the acceleration,$a$. In summary, $F_1$ is the force working on mass $m_1$ and $F_2$ is the force working on mass $m_2$, and $T$

is the cord tension that connecting $m_1$ and $m_2$, hence

$$\begin{aligned} m_1 a + kx &= m_2 g - m_2 a \\ kx &= -m_2 a - m_1 a + m_2 g \\ k &= \frac{-(m_1 + m_2)a + m_2 g}{x} \end{aligned}$$

when the system is in equilibrium, the acceleration should be $a = 0$, thus

$$\begin{aligned} k &= \frac{-(m_1 + m_2)(0) + m_2 g}{x} \\ k &= \frac{m_2 g}{x} \\ k &= \frac{m_2 g}{L} \end{aligned}$$

because the system is in equilibrium when $x = L$. Thus, $L > d$.

(b) Destroy the pulley joint, see what happens to box 1

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringTwomasses : public Test
{
	public:
	SpringTwomasses()
	{
		b2Body* ground = NULL;
		{// Create the pulley
			b2BodyDef bdp;
			ground = m_world->CreateBody(&bdp);

			b2CircleShape circle;
			circle.m_radius = 0.5f;

			circle.m_p.Set(12.0f, 0.5f); // circle with center of
				(12,0.5)
			ground->CreateFixture(&circle, 0.0f);
		}
		{
			b2BodyDef bd;
			ground = m_world->CreateBody(&bd);

			b2EdgeShape shape;
			shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(12.0f,
				0.0f));
```

```
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(12.0f, 0.0f), b2Vec2(12.0f
            ,-20.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    // Create a static body as the box for the spring
    b2BodyDef bd1;
    bd1.type = b2_staticBody;
    bd1.angularDamping = 0.1f;

    bd1.position.Set(1.0f, 0.5f);
    b2Body* springbox = m_world->CreateBody(&bd1);

    b2PolygonShape shape1;
    shape1.SetAsBox(0.5f,5.5f);
    springbox->CreateFixture(&shape1, 5.0f);


    // Create the left box connected to the spring as the movable
         object
    b2PolygonShape leftboxShape;
    leftboxShape.SetAsBox(0.5f, 0.5f);

    b2FixtureDef leftboxFixtureDef;
    leftboxFixtureDef.restitution = 0.75f;
    leftboxFixtureDef.density = 7.3f; // this will affect the box
         mass
    leftboxFixtureDef.friction = 0.1f;
    leftboxFixtureDef.shape = &leftboxShape;

    b2BodyDef leftboxBodyDef;
    leftboxBodyDef.type = b2_dynamicBody;
    leftboxBodyDef.position.Set(5.0f, 0.5f);

    m_box = m_world->CreateBody(&leftboxBodyDef);
    b2Fixture *leftboxFixture = m_box->CreateFixture(&
        leftboxFixtureDef);
    //m_box->SetGravityScale(-7); // negative means it will goes
         upward, positive it will goes downward

    // Create the box hanging on the right
    b2PolygonShape boxShape2;
```

```
boxShape2.SetAsBox(0.5f, 0.5f); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 3.3835f; // this will affect the box
     mass, mass = density*5.9969
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(12.8f, -1.5f);

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Make a distance joint for the box / ball with the static
    box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    leftboxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
    allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 10.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(5.5f, 0.5f); // the position of the end string
     of the left cord is (5.5,0.5) connecting to the left box
b2Vec2 anchor2(12.8f, -1.0f); // the position of the end
    string of the right cord is (14.0f,-1) connecting to the
     right hanging box
b2Vec2 groundAnchor1(11.5f, 0.5f ); // the string of the cord
     is tightened at (11.5, 0.5)
b2Vec2 groundAnchor2(12.5f, 0.5f); // the string of the cord
```

```
                    is tightened at (12.5, 0.5)
            // the last float is the ratio
            pulleyDef.Initialize(m_box, m_boxr, groundAnchor1,
                groundAnchor2, anchor1, anchor2, 1.0f);

            m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

            m_time = 0.0f;
        }
        b2Body* m_box;
        b2Body* m_boxr;
        b2DistanceJoint* m_joint;
        b2PulleyJoint* m_joint1;
        float m_length;
        float m_time;
        float m_minLength;
        float m_maxLength;
        float m_hertz;
        float m_dampingRatio;

        void Keyboard(int key) override
        {
            switch (key)
            {
                case GLFW_KEY_A:
                //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
                m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_S:
                //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f));
                m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f),
                    true);
                break;
                case GLFW_KEY_D:
                m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f), true)
                    ;
                break;
                case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f), true)
                    ;
                break;
                case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f), true
                    );
                break;
            }
        }
```

```
        void UpdateUI() override
        {
            ImGui::SetNextWindowPos(ImVec2(10.0f, 300.0f));
            ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
            ImGui::Begin("Joint Controls", nullptr,
                ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

            if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                f"))
            {
                m_length = m_joint->SetLength(m_length);
            }

            if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"
                ))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                , 2.0f, "%.1f"))
            {
                float stiffness;
                float damping;
                b2LinearStiffness(stiffness, damping, m_hertz,
                    m_dampingRatio, m_joint->GetBodyA(), m_joint->
                    GetBodyB());
                m_joint->SetStiffness(stiffness);
                m_joint->SetDamping(damping);
            }

            ImGui::End();
        }
        void Step(Settings& settings) override
        {
            b2MassData massData = m_box->GetMassData();
            b2Vec2 position = m_box->GetPosition();
            b2Vec2 velocity = m_box->GetLinearVelocity();
            b2MassData massDatar = m_boxr->GetMassData();
            b2Vec2 positionr = m_boxr->GetPosition();
            b2Vec2 velocityr = m_boxr->GetLinearVelocity();
            m_time += 1.0f / 60.0f; // assuming we are using frequency of
```

```
                    60 Hertz

                g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
                    apply different force to the box");
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                    f", m_time);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Mass position =
                    (%4.1f, %4.1f)", position.x, position.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Mass velocity =
                    (%4.1f, %4.1f)", velocity.x, velocity.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Mass position =
                    (%4.1f, %4.1f)", positionr.x, positionr.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Mass velocity =
                    (%4.1f, %4.1f)", velocityr.x, velocityr.y);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Left Mass = %.6f",
                    massData.mass);
                m_textLine += m_textIncrement;
                g_debugDraw.DrawString(5, m_textLine, "Right Mass = %.6f",
                    massDatar.mass);
                m_textLine += m_textIncrement;
                // Print the result in every time step then plot it into
                    graph with either gnuplot or anything

                printf("%4.2f\n", position.x);

                Test::Step(settings);
        }
        static Test* Create()
        {
                return new SpringTwomasses;
        }

    };

    static int testIndex = RegisterTest("Oscillations", "Spring Two Masses",
        SpringTwomasses::Create);
```

**C++ Code 96:** *tests/spring_twomasses.cpp "Two Masses Spring-Mass System Box2D"*

Some explanations for the codes:

- ____

- ____

**Figure 17.17:** *The two masses spring-mass system simulation , besides keyboard press events you can click the box with mouse and drag it toward positive or negative x axis to see it oscillating and affecting the other body that is hanging on the right (the current simulation code can be located in:* ***DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_twomasses.cpp****).*

## XV. Friction and Oscillations

The amplitude of a real-life spring-mass system when oscillating will decrease over time, the mass will oscillates around its' equilibrium position with smaller and smaller magnitude until it stops.

For a spring-mass system, the friction between the mass and the surrounding air media cause the amplitude of the oscillation to diminish. When the spring is moving to the left, the friction force is moving to the right, and when the spring is moving to the right, the friction force is moving to the left.

- When the velocity is positive, $\frac{dx}{dt} > 0$, then the frictional force $f_k$ must be negative, $f_k < 0$.
- When the velocity is negative, $\frac{dx}{dt} < 0$, then the frictional force $f_k$ must be positive, $f_k > 0$.

To assume that the frictional force is linearly dependent on the velocity:

$$f_k = -c\frac{dx}{dt} \tag{17.36}$$

where $c$ is a positive constant referred to as the friction coefficient. This force-velocity relationship is called a linear damping force; a damped oscillation is the same as an oscillations that decays.

Now, we will have the differential equation describing spring-mass system with a linear damping force and a linear restoring force as

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{17.37}$$

It must be verified that solutions to the second order differential equation above behave in a manner consistent with the real-life observations. There are a lot of forces that can make the spring-mass system decays, other than a linear damping force.

To solve the second order differential equation, we are going to use exponential again, $e^{rt}$, as two linearly independent solutions can almost be written in the form of exponential. Where $r$ is the characteristic equation obtained by direct substitution.

$$mr^2 + cr + k = 0 \tag{17.38}$$

is the characteristic equation of $m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0$. The two roots of the characteristic equation are

$$r = \frac{-c \pm \sqrt{c^2 - 4mk}}{2m} \tag{17.39}$$

with $c$ and $mk$ must have the same dimension.

There are three cases:

- **Case I: Underdamped Oscillations ($c^2 < 4mk$)**
  If $c^2 < 4mk$, then the coefficient of friction is small, thus the damping force is not particularly large. In this case, the roots of the characteristic equation are complex conjugates of each other(complex number)
  $$r = -\frac{c}{2m} \pm i\omega$$
  where
  $$\omega = \frac{\sqrt{4mk - c^2}}{2m} = \sqrt{\frac{k}{m} - \frac{c^2}{4m^2}}$$

The general solution for this case is

$$\begin{aligned} x(t) &= ae^{(-\frac{c}{2m}+i\omega)t} + be^{(-\frac{c}{2m}-i\omega)t} \\ &= e^{-\frac{ct}{2m}}\left(ae^{i\omega t} + be^{-i\omega t}\right) \end{aligned} \tag{17.40}$$

An arbitrary linear combination of $e^{i\omega t}$ and $e^{-i\omega t}$ is equivalent to an arbitrary linear combination of cos $\omega t$ and sin $\omega t$, thus the motion of a linearly damped spring-mass system in the underdamped case is described by

$$\begin{aligned} x(t) &= e^{-\frac{ct}{2m}}\left(c_1 \cos \omega t + c_2 \sin \omega t\right) \\ &= Ae^{-\frac{ct}{2m}} \sin(\omega t + \phi_0) \end{aligned} \tag{17.41}$$

The solution is the product of an exponential and a sinusoidal function.

**Figure 17.18:** *The exponential function $Ae^{-\frac{ct}{2m}}$ and its negative $-Ae^{-\frac{ct}{2m}}$ along with the solution $x(t) = Ae^{-\frac{ct}{2m}} \sin(\omega t + \phi_0)$ for the underdamped case (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/C++ Gnuplot/ch15-underdampedoscillation/main.cpp****).*

- At the maximum value of the sinusoidal function, $x(t)$ equals the exponential alone.
- At the minimum value of the sinusoidal function, $x$ equals minus the exponential.
- The exponential $Ae^{-\frac{ct}{2m}}$ is called the amplitude of the oscillation. Thus, the amplitude is exponentially decays.

- **Case II: Critically Damped Oscillations ($c^2 = 4mk$)**
  When $c^2 = 4mk$, the spring-mass system is said to be critically damped. Mathematically, it makes both the exponential solutions become the same. But, from the physical point of view, this case is insignificant. This is because the quantities $c, m$, and $k$ are all experimentally measured quantities. Any small deviation can shift the system' condition into overdamped or underdamped oscillation.

The solution for critically damped oscillation is

$$x(t) = e^{-\frac{ct}{2m}}(At + B) \tag{17.42}$$

The solution will returns to its equilibrium position as $t \to \infty$, since the exponential decay is much stronger than an algebraic growth.

- **Case III: Overdamped Oscillations ($c^2 > 4mk$)**
  If the friction is sufficiently large, then

$$c^2 > 4mk$$

this is called as an overdamped system. The motion of the mass is no longer a decaying oscillation. Thus, the solution is

$$x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t} \tag{17.43}$$

where $r_1$ and $r_2$ are real and both negative,

$$r_1 = \frac{-c + \sqrt{c^2 - 4mk}}{2m}$$
$$r_2 = \frac{-c - \sqrt{c^2 - 4mk}}{2m}$$

If the friction is sufficiently large, we should expect that the mass decays to its equilibrium position quite quickly.

**Figure 17.19:** *The solution for the motion of the mass $x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$ with different values of $c_1$ and $c_2$ for the overdamped case (the code can be located in:* ***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch17-Oscillations/Overdamped Oscillation/main.cpp****).*

# Chapter 18

# DFSimulatorC++ XII: Waves

*"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - **Albert Silverberg***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 97:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 19

# DFSimulatorC++ XIII: Temperature, Heat, and the First Law of Thermodynamics

*"Kaing bla kaing.. I was abused, I'm always abused.. Sweden is very abusive.." - **Piano Bludut***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 98:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 20

# DFSimulatorC++ XIV: The Kinetic Theory of Gases

*"Hasta la meo (then jump very high when want to go out)" - **Bludut Piano***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 99:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 21

# DFSimulatorC++ XV: Entropy and the Second Law of Thermodynamics

*"When muzzle meets puzzle it becomes buzzle, when muzzle meets buzzle it becomes zuzzle bla, when muzzle meets zuzzle it becomes duzzle.. so why bla why bla why?" - **Sweden Sexy***

## Momentum

### I. Simulation for Momentum with Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./testbed**

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

**C++ Code 100:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 22

# DFSimulatorC++ XVI: Probability and Data Analysis

*"My only place is by your side. If you want to carry out your wish, I will follow you. That is my Nature." -* ***Sarah (Suikoden III)***

When we have movement and we
All the files and data used in this chapter can be found in this book repository:

**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch22-Probability and Data Analysis/**

## I. Statistics and Data Analysis

Statistical methods are used to analyze data from an industrial process (manufacturing, energy sources, drug testing). Statistical methods are involved with gathering of information or **scientific data**, which then will be processed to extract the knowledge from the data and improve the **quality** of the process.

## II. Variability

Life would be simple for scientists if the observed product were always the same and were always on target, there would be no need for for statistical methods. Statisticians make use of fundamental laws of probability and statistical inference to draw conclusions about scientific systems.

Information is gathered in the form of **samples**, or collections of **observations**. Samples are collected from **populations**, which are collections of all individuals or individual items of a particular type.

Research scientists gain much from scientific data. Data provide understanding of scientific phenomena.

# Chapter 23

# DFSimulatorC++ XVII: Numerical Linear Algebra

*"Plaisir d'amour, ne dure qu'un moment. Chagrin d'amour dure toute la vie."* - ***Plaisir D'amour (Nana Mouskori)***

When I read the book "Engineering Circuit Analysis" [8] there are nonlinear circuits we encounter every day, for examples they capture and decode signals for TVs and radios, perform calculations hundreds of billions of times a second inside microprocessors, convert speech into electrical signals for transmission over fibre-optic cables as well as cellular networks. The problem is nonlinear systems are not easy to solve that is why we often use linearization for the nonlinear systems so it will be easy to solve that problem, even if no physical system (including electrical circuits) is ever perfectly linear. It is the linear approximations to physical electrical circuits that are used for the computation. This is why we need to learn some techniques in Numerical Linear Algebra, since every mathematical scientist needs to work effectively with vectors and matrices, that will expand more to functions and operators. Let the fun begins.

All the files and data used in this chapter can be found in this book repository:

**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/**

## I. Matrix

- Rectangular array of numbers are known as matrix, it can represent linear systems and has application in various fields from engineering, physical science, to economics. An $n \times m$ matrix is a rectangular array of elements with $n$ rows and $m$ columns, in which not only is the value of an element important, but also its position in the array.

  The notation for an $n \times m$ matrix will be a capital letter such as $A$ for the matrix and lowercase letters with double subscripts, such as $a_{ij}$, to refer to the entry at the intersection of the $i$th row and $j$th column.

**Definition 23.1: Matrix**

A matrix is a rectangular array of numbers. The numbers in the array are called the entries in the matrix.

$$A = \begin{matrix} a_{11} & + a_{12} & \dots & a_{1m} \\ a_{21} & + a_{22} & \dots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & + a_{n2} & \dots & a_{nm} \end{matrix}$$

Two matrices are defined to be equal if they have the same size and their corresponding entries are equal.

**Definition 23.2: Sum and Difference of Matrices**

If $A$ and $B$ are matrices of the same size, then the sum $A + B$ is the matrix obtained by adding the entries of $B$ to the corresponding entries of $A$.

The difference $A - B$ is the matrix obtained by subtracting the entries of $B$ from the corresponding entries of $A$. Matrices of different sizes cannot be added or subtracted.

In matrix notation, if $A = [a_{ij}]$ and $B = [b_{ij}]$ have the same size, then

$$(A + B)_{ij} = (A)_{ij} + (B)_{ij} = a_{ij} + b_{ij} \tag{23.1}$$

$$(A - B)_{ij} = (A)_{ij} - (B)_{ij} = a_{ij} - b_{ij} \tag{23.2}$$

**Definition 23.3: Scalar Multiple of a Matrix**

If $A$ is any matrix and $c$ is any scalar, then the product $cA$ is the matrix obtained by multiplying each entry of matrix $A$ by $c$. The matrix $cA$ is said to be a scalar multiple of $A$.

In matrix notation, if $A = [a_{ij}]$, then

$$(cA)_{ij} = c(A)_{ij} = ca_{ij} \tag{23.3}$$

**Definition 23.4: Multiplying Matrices**

If $A$ is an $m \times r$ matrix and $B$ is an $r \times n$ matrix, then the product $AB$ is the $m \times n$ matrix whose entries are determined as follows:

To find the entry in row $i$ and column $j$ of $AB$, single out row $i$ from the matrix $A$ and column $j$ from the matrix $B$. Multiply the corresponding entries from the row and column together, and then add up the resulting products.
In matrix notation, if $A = [a_{ij}]$ is an $m \times r$ matrix and $B = [b_{ij}]$ is an $r \times n$ matrix, then the entry $(AB)_{ij}$ in row $i$ and column $j$ of $AB$ is given by

$$(AB)_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{ir}b_{rj} \tag{23.4}$$

- Matrix multiplication has an important application to systems of linear equations. Consider a system of $m$ linear equations in $n$ unknowns:

$$\begin{array}{lllll} a_{11}x_1 & + a_{12}x_2 & + \dots & + a_{1n}x_n & = b_1 \\ a_{21}x_1 & + a_{22}x_2 & + \dots & + a_{2n}x_n & = b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1}x_1 & + a_{n2}x_2 & + \dots & + a_{nn}x_n & = b_n \end{array}$$

Since two matrices are equal if and only if their corresponding entries are equal, we can replace the $m$ equations in this system by the single matrix equation

$$\begin{bmatrix} a_{11}x_1 & + a_{12}x_2 & + \dots & + a_{1n}x_n \\ a_{21}x_1 & + a_{22}x_2 & + \dots & + a_{2n}x_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}x_1 & + a_{m2}x_2 & + \dots & + a_{mn}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

The $m \times 1$ matrix on the left side of this equation can be written as product to give

$$\begin{bmatrix} a_{11} & + a_{12} & + \dots & + a_{1n} \\ a_{21} & + a_{22} & + \dots & + a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & + a_{m2} & + \dots & + a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

If we designate these matrices by $A, \boldsymbol{x}$, and $\boldsymbol{b}$, respectively, then we can replace the original system of $m$ equations in $n$ unknowns by the single matrix equation

$$A\boldsymbol{x} = \boldsymbol{b} \tag{23.5}$$

The matrix $A$ in this equation is called the coefficient matrix of the system. The augmented matrix for the system is obtained by adjoining $\boldsymbol{b}$ to $A$ as the last column; thus the augmented matrix is

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{array}\right]$$

**Definition 23.5: Diagonal Matrix**

A diagonal matrix is a square matrix (a square matrix has the same number of rows and columns) $D = (d_{ij})$ with $d_{ij} = 0$ whenever $i \neq j$. The identity matrix of order $n$, $I_n = (\delta_{ij})$, is a diagonal matrix with entries

$$\begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

- The identity matrix $I_n$ commutes with any $n \times n$ matrix $A$; that is, the order of multiplication does not matter.

$$I_n A = A = A I_n$$

For random matrix $A$ and $B$ with correct size that can be multiplied, $AB$ is not always equal to $BA$.

**Definition 23.6: Transpose of a Matrix**

If $A$ is any $m \times n$ matrix, then the transpose of $A$, denoted by $A^T$, is defined to be the $n \times m$ matrix that results by interchanging the rows and columns of $A$; that is, the first column of $A^T$ is the first row of $A$, the second column of $A^T$ is the second row of $A$, and so forth.

**Definition 23.7: Trace of a Matrix**

If $A$ is a square matrix, then the trace of $A$, denoted by $tr(A)$, is defined to be the sum of the entries on the main diagonal of $A$. The trace of $A$ is undefined if $A$ is not a square matrix.

**Theorem 23.1: Properties of Matrix Arithmetic**

Assuming that the sizes of the matrices are such that the indicated operations can be performed, the following rules of matrix arithmetic are valid.

(a) $A + B = B + A$

(b) $A + (B + C) = (A + B) + C$

(c) $A(BC) = (AB)C$

(d) $A(B + C) = AB + AC$

(e) $(B + C)A = BA + CA$

(f) $A(B - C) = AB - AC$

(g) $(B - C)A = BA - CA$

(h) $a(B + C) = aB + aC$

(i) $a(B - C) = aB - aC$

(j) $(a + b)C = aC + bC$

(k) $(a - b)C = aC - bC$

(l) $a(bC) = (ab)C$

(m) $a(BC) = (ab)C = B(aC)$

**Theorem 23.2: Properties of Zero Matrices**

If $c$ is any scalar, and if the sizes of the matrices are such that the operations can be performed, then:

(a) $A + 0 = 0 + A = A$

(b) $A - 0 = A$

(c) $A - A = A + (-A) = 0$

(d) $0A = 0$

(e) If $cA = 0$, then $c = 0$ or $A = 0$

- The commutative law of real arithmetic is not valid in matrix arithmetic.
- A square matrix with 1's on the main diagonal and zeros elsewhere is called an identity matrix. It is denoted by the letter $I$ or $I_n$, with $n$ is the size of the matrix.

**Definition 23.8: Inverse of a Matrix**

If $A$ is a square matrix, and if a matrix $B$ of the same size can be found such that $AB = BA = I$, then $A$ is said to be invertible (or nonsingular) and $B$ is called an inverse of $A$. If no such matrix $B$ can be found, then $A$ is said to be singular.

When

$$AB = BA = I$$

we say that $A$ and $B$ are inverses of one another.

If $B$ and $C$ are both inverses of the matrix $A$, then $B = C$. This means that an invertible matrix has exactly one inverse.

**Theorem 23.3: Simpe Inverse Formula**

The matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

is invertible if and only if $ad - bc \neq 0$, in which case the inverse is given by the formula

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{23.6}$$

The quantity $ad - bc$ is called the determinant of matrix $A$.

**Theorem 23.4: Inverse of a Matrices Product**

If $A$ and $B$ are invertible matrices with the same size, then $AB$ is invertible and

$$(AB)^{-1} = B^{-1}A^{-1}$$

A product of any number of invertible matrices is invertible, and the inverse of the product is the product of the inverses in the reverse order.

**Definition 23.9: Singular and Nonsingular Matrix**

An $n \times n$ matrix $A$ is said to be nonsingular (invertible) if an $n \times n$ matrix $A^{-1}$ exists with

$$AA^{-1} = A^{-1}A = I$$

The matrix $A^{-1}$ is the inverse of matrix $A$.

A matrix without an inverse is called a singular (noninvertible).

**Definition 23.10: Powers of a Matrix**

If $A$ is a square matrix, then

$$A^0 = I$$

$$A^n = AA \dots A \text{ [n factors]}$$

If $A$ is invertible, then

$$A^{-n} = (A^{-1})^n = A^{-1}A^{-1} \dots A^{-1} \text{ [n factors]}$$

Because these definitions parallel those for real numbers, the usual laws of nonnegative exponents hold

$$A^r A^s = A^{r+s}$$

$$(A^r)^s = A^{rs}$$

**Theorem 23.5: Properties of Negative Exponents**

If $A$ is invertible and $n$ is a nonnegative integer, then:

(a) $A^{-1}$ is invertible and $(A^{-1})^{-1} = A$.

(b) $A^n$ is invertible and $(A^n)^{-1} = A^{-n} = (A^{-1})^n$.

(c) $kA$ is invertible for any nonzero scalar $k$, and $(kA)^{-1} = k^{-1}A^{-1}$.

- In matrix arithmetic, where we have no commutative law for multiplication, we write

$$(A+B)^2 = A^2 + AB + BA + B^2$$

It is only in the special case where $A$ and $B$ commute (i.e., $AB = BA$) that we can go a step further and write

$$(A+B)^2 = A^2 + 2AB + B^2$$

- If $A$ is a square matrix, say $n \times n$, and ig

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_m x^m$$

is any polynomial, then we define the $n \times n$ matrix $p(A)$ to be

$$p(A) = a_0 I + a_1 A + a_2 A^2 + \dots + a_m A^m \tag{23.7}$$

this is called matrix polynomial in $A$, with $p(A)$ is obtained by substituting $A$ for $x$ and replacing the constant term $a_0$ by the matrix $a_0 I$.

Any two matrix polynomials in $A$ also commute; that is, for any polynomials $p_1$ and $p_2$ we have

$$p_1(A)p_2(A) = p_2(A)p_1(A) \tag{23.8}$$

**Theorem 23.6: Properties of the Transpose**

If the sizes of the matrices are such that the stated operations can be performed, then:

(a) $(A^T)^T = A$

(b) $(A+B)^T = A^T + B^T$

(c) $(A-B)^T = A^T - B^T$

(d) $(kA)^T = kA^T$

(e) $(AB)^T = B^T A^T$

The transpose of a product of any number of matrices is the product of the transposes in the reverse order.

If $A$ is an invertible matrix, then $A^T$ is also invertible and

$$(A^T)^{-1} = (A^{-1})^T$$

## II. Matrix-Vector Multiplication

[DF*] A matrix with only one column is called a column vector. Unless indicated otherwise, a vector $x \in \mathbb{R}^n$ will be a column vector,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

A matrix with only one row is called a row vector. The transpose of $x$ is the row vector

$$x^T = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix}$$

The symbol $x_i$ will denote component $i$ of the vector $x$. Superscripts will generally denote distinct vectors.
For example, $e^j$ will denote the vector where all components are zero except for component $j$ which is unity;

$$e^j = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \text{ (at index } j) \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

that is, $e^j$ is the unit vector in dimension $j$.

Another example, $0_n$ is the vector of zeros, $0_{n\times n}$ is the matrix of zeros, $1_n \in \mathbb{R}^n$ is the vector of ones, $I_n$ is the $n \times n$ identity matrix.

[DF*] Let $x$ be an $n$-dimensional column vector and let $A$ be an $m \times n$ matrix ($m$ rows, $n$ columns). Then the matrix-vector product $b$ is a $m$-dimensional column vector such that

$$b = Ax$$

then $b$ is a linear combination of the columns of $A$, with

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

we can then defined $b$ as follows:

$$b_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \dots, m \tag{23.9}$$

Here $b_i$ denotes the $i$th entry of $b$, $a_{ij}$ denotes the $i, j$ entry of $A$, and $x_j$ denotes the $j$th entry of $x$.

[DF*] Few assumptions:

- All quantities belong to $\mathbb{C}$, the field of complex numbers.

[DF*] The map $x \rightarrowtail Ax$ is linear, which means, for any $x, y \in \mathbb{C}^n$ and any $\alpha \in \mathbb{C}$,

$$A(x + y) = Ax + Ay$$
$$A(\alpha x) = \alpha Ax$$

## i. C++ Computation: A Matrix Times a Vector

Let $a_j$ denote the $j$th column of $A$, an $m$-vector, then

$$b = Ax = \sum_{j=1}^{n} x_j a_j \tag{23.10}$$

we see that $b$ is expressed as a linear combination of the column $a_j$.

Now for the computation we can use this simple code from **bottomscience.com**, you can create this c++ file with name of **multiplicationofmatrices.cpp** or just copy the related code from the repository of this book.

```
#include <iostream>
using namespace std;

int main()
{
        int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;

        cout << "Enter rows and columns for first matrix: ";
        cin >> r1 >> c1;
        cout << "Enter rows and columns for second matrix: ";
        cin >> r2 >> c2;

        // If column of first matrix in not equal to row of second matrix,
        // ask the user to enter the size of matrix again.
        while (c1!=r2)
        {
                cout << "Error! column of first matrix not equal to row of
                    second.";

                cout << "Enter rows and columns for first matrix: ";
                cin >> r1 >> c1;

                cout << "Enter rows and columns for second matrix: ";
                cin >> r2 >> c2;
        }

        // Storing elements of first matrix.
        cout << endl << "Enter elements of matrix 1:" << endl;
        for(i = 0; i < r1; ++i)
        for(j = 0; j < c1; ++j)
        {
                cout << "Enter element a" << i + 1 << j + 1 << " : ";
                cin >> a[i][j];
        }

        // Storing elements of second matrix.
        cout << endl << "Enter elements of matrix 2:" << endl;
        for(i = 0; i < r2; ++i)
        for(j = 0; j < c2; ++j)
        {
                cout << "Enter element b" << i + 1 << j + 1 << " : ";
                cin >> b[i][j];
        }

        // Initializing elements of matrix mult to 0.
        for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
```

```
            {
                  mult[i][j]=0;
            }

            // Multiplying matrix a and b and storing in array mult.
            for(i = 0; i < r1; ++i)
            for(j = 0; j < c2; ++j)
            for(k = 0; k < c1; ++k)
            {
                  mult[i][j] += a[i][k] * b[k][j];
            }

            // Displaying the multiplication of two matrix.
            cout << endl << "Output Matrix: " << endl;
            for(i = 0; i < r1; ++i)
            for(j = 0; j < c2; ++j)
            {
                  cout << " " << mult[i][j];
                  if(j == c2-1)
                  cout << endl;
            }

            return 0;
      }
```

**C++ Code 101:** *multiplicationofmatrices.cpp "Computation: Multiplication of matrices"*

To compile it, type:
**g++ -o main multiplicationofmatrices.cpp**
**./main**

Now, we can try to have a matrix $A$ of size $3 \times 3$ with vector $x$ a column vector with size of 3.

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To compute for $b$ we just use the code above and we will obtain

$$b = \begin{bmatrix} 14 \\ 32 \\ 50 \end{bmatrix}$$

## ii. C++ Computation: A Matrix Times a Vector with Data from Textfile

The difference from the previous section is that we don't have to type one by one the entries of the matrix and the vector, if we have a textfile filled with columns of data, and we know the number

```
Enter rows and columns for first matrix: 3
3
Enter rows and columns for second matrix: 3
1

Enter elements of matrix 1:
Enter element a11 : 1
Enter element a12 : 2
Enter element a13 : 3
Enter element a21 : 4
Enter element a22 : 5
Enter element a23 : 6
Enter element a31 : 7
Enter element a32 : 8
Enter element a33 : 9

Enter elements of matrix 2:
Enter element b11 : 1
Enter element b21 : 2
Enter element b31 : 3

Output Matrix:
 14
 32
 50
```

**Figure 23.1:** *The computation for matrix $A$ times a vector $x$, to obtain $b = Ax$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/multiplicationofmatrices.cpp**).*

of elements for the vector and the matrix size, we can just put it under the working directory along with the C++ code. We are also using **struct** and **pointers** here to take the vector elements and matrix indices from textfile that can then be processed or computed at the **int main()** / the main driver for C++.

In a working directory, create 2 textfiles, one for storing vector elements (**vector1.txt**), the other for storing matrix (**matrix1.txt**). Then, create a C++ file name **main.cpp**, the code is below, and it is also available at the repository.

```
#include <fstream>
#include <vector>
#include <cmath>

#include "gnuplot-iostream.h"

const int N = 4;

using std::cout;
using std::endl;
using namespace std;

void printArray(int** arr) {

        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        cout <<arr[i][j] << ' ';
                }
                cout << endl;
        }
}
```

```
int* vec() {
        static int x[N];
        std::fstream in("vector1.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

struct matrix
{
        int arr1[N][N];
};

struct matrix func(int N)
{
        struct matrix matrix_mem;
        std::fstream in("matrix1.txt");
        float matrixtiles[N][N];
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        in >> matrixtiles[i][j];
                        matrix_mem.arr1[i][j] = matrixtiles[i][j] ;
                }
        }
        return matrix_mem;
}

int main() {

        int* ptrvec;
        ptrvec = vec();
        cout << "Vector x is: " << endl;
        for (int i = 0; i < N; ++i)
        {
                cout <<ptrvec[i]<< ' ' << endl;
        }


        cout << "Matrix A is: " << endl;

        struct matrix a;
        a = func(N);
```

```
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                cout << a.arr1[i][j]<< ' ';
            }
            cout << endl;
        }


        float b[N][N];
        float B[N];
        for (int i = 0; i < N; ++i)
        {
            B[i] = 0;
        }

        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                b[i][j] = 0;
            }
        }

        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                b[i][j] += a.arr1[i][j] * ptrvec[j];
                //cout << b[i][j] << ' ';
            }
        }

        cout << "b = A*x : " << endl;
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                B[i] += b[i][j] ;
            }

            cout << B[i] << endl;
        }
        return 0;

    }
```

**C++ Code 102:** *main.cpp "Matrix times a Vector from Textfile Data"*

```
1
8
8
2
```

**C++ Code 103:** *"vector1.txt"*

```
1 8 8 2
2 8 8 1
8 2 1 8
8 1 2 8
```

**C++ Code 104:** *"matrix1.txt"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

**Figure 23.2:** *The computation for matrix $A$ times a vector $x$, to obtain $b = Ax$ with C++, with the vector $x$ and matrix $A$ are obtained from loading a textfile (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Multiplication of Matrix and Vector from Textfile/main.cpp**).*

Explanation for the codes:

- To obtain the vector data from textfile we are going to use pointers. Returning a normal array from a function using pointers sometimes can give unexpected results. But this behavior and warnings can be avoided by declaring the array to be a **static** one.

```
int* vec() {
        static int x[N];
        std::fstream in("vector1.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
```

**Figure 23.3:** *The computation for matrix $A$ times a vector $x$, to obtain $b = Ax$ with Julia, the code to obtained the result is only 3 lines (we need to type the elements not loading it from textfile), but when we have billions of elements for the vector and matrix with huge dimensions we should use C++ code, since it is faster to compile and obtain the final result.*

```
        {
                in >>vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}
```

**int* vec** -> return type- address of integer array.
Inside the look **for (..)** we are taking the data form textfile and save it to become array, as the array initialisation **a[i] = ....**
**return x;** -> address of x returned.

- Inside the **int main()**

```
int main() {
        ...

        int* ptrvec;
        ptrvec = vec();
        cout << "Array is: " << endl;
        for (int i = 0; i < N; i++)
        {
                cout <<ptrvec[i]<< ' ' << endl;
        }
        return 0;

}
```

**int* ptrvec** -> a pointer to hold address.
**ptr = vec();** -> address of x.
**ptrvec[i]** -> ptrvec[i] is equivalent to *(ptrvec+i)

The great thing of using pointer to get the vector' elements from textfile is that we can do computation as well, we get each of the elements for granted, thus not only for matrix times vector but we can try **cout «ptrvec[i]*ptrvec[i]« ' ' « endl;** to obtain the square of the vector element, then find the norm or do another computation that we need.

- To obtain the matric entries / indices, we are going to use **struct**. We can make a function returns an array by declaring it inside a structure.

```
struct matrix
{
        int arr1[N][N];
};

struct matrix func(int N)
{
        struct matrix matrix_mem;
        std::fstream in("matrix1.txt");
        float matrixtiles[N][N];
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        in >> matrixtiles[i][j];
                        matrix_mem.arr1[i][j] = matrixtiles[i][j] ;
                }
        }
        return matrix_mem;
}
```

**int arr1[N][N];** -> Array declared inside structure, matrix of size $N \times N$.
**struct matrix func (int N)** -> Return type is struct matrix.
**struct matrix matrix_mem;** -> matrix structure member declared.
**in » matrixtiles[i][j]; matrix_mem.arr1[i][j] = matrixtiles[i][j] ;** -> Array initialisation, taking the data from textfile.
**return matrix_mem** -> Address of structure member returned.

What can this code do for physical simulation and scientific computing? When we observe in simulation from Box2D, for example, taking data of linear velocity, kinetic energy, force applied, we can try to create a matrix for certain of the forces / energy or other variable and see the pattern if it is multiplied by another vector force or a vector with elements filled with the acceleration. Is the pattern linear or nonlinear or have inverse relation, all can be easily determined with this code.

## iii. C++ Computation: Matrix Sum and Subtraction with Data from Textfiles

In this section, we are going to use **Armadillo** library, so we do not need to write the function to calculate the matrix sum and subtraction, as a bonus since Armadillo has created a lot of functions for matrix computation we also include the matrix inverse, determinant and product of two matrices.

Inside the working directory prepare two textfiles for matrix $A$ and matrix $B$.

**Figure 23.4:** *The textfile containing the entries for matrix B with size of* $3 \times 3$.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
{
        mat A;
        mat B;

        A.load("MatrixA.txt");
        B.load("MatrixB.txt");

        cout << "Matrix A:" << endl;
        A.print();
        cout << endl;
        cout << "Matrix B:" << endl;
        B.print();

        // determinant
        cout << "det(A): " << det(A) << endl;
        cout << endl;
        cout << "det(B): " << det(B) << endl;
        cout << endl;

        // inverse
        cout << "inv(A): " << endl << inv(A) << endl;
        cout << "inv(B): " << endl << inv(B) << endl;
        cout << endl;

        // Matrix summation, subtraction and product
        cout << "A+B: " << endl << A+B << endl;
        cout << "A-B: " << endl << A-B << endl;
        cout << "AB: " << endl << A*B.t() << endl;

        return 0;
```

```
    }
```

**C++ Code 105:** *main.cpp "Matrix Sum and Subtraction with Armadillo"*

To compile it, type:
**g++ -o result main.cpp -larmadillo**
**./result**

or with the Makefile, type:
**make**
**./main**

Using Armadillo library is quite easy to follow, you just need to read the official tutorial and documentation available on their website, the code created here will be shorter unlike creating your own C++ source code from scratch to compute the adjoint of a matrix to determine the inverse of a matrix without using any library.

```
Matrix A:
   1.0000   2.0000   2.0000
   3.0000   3.0000   4.0000
   5.0000   3.0000   7.0000

Matrix B:
   11.0000    2.0000   12.0000
   13.0000    3.0000   40.0000
   51.0000   36.0000   72.0000
det(A): -5

det(B): -7476

inv(A):
  -1.8000   1.6000  -0.4000
   0.2000   0.6000  -0.4000
   1.2000  -1.4000   0.6000

inv(B):
   0.1637  -0.0385  -0.0059
  -0.1477  -0.0241   0.0380
  -0.0421   0.0393  -0.0009


A+B:
   12.0000    4.0000   14.0000
   16.0000    6.0000   44.0000
   56.0000   39.0000   79.0000

A-B:
  -10.0000         0  -10.0000
  -10.0000         0  -36.0000
  -46.0000  -33.0000  -65.0000

AB:
   1.3900e+02   8.0000e+01   2.3600e+02
   2.7600e+02   1.5900e+02   4.4400e+02
   4.5100e+02   2.7100e+02   6.8400e+02
```

**Figure 23.5:** *The computation for matrices summation, subtraction and product (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Armadillo-Matrix Sum and Subtraction/main.cpp**).*

## iv. C++ Computation: Matrices Multiplication with Data from Textfile

In this section, we are going to use data from textfiles, loaded into C++ code and then compose a vandermonde matrix out of it and then create the transpose of that matrix and multiply the matrices. This time create a function for matrices multiplication, so we are not depending on Armadillo.

We need to prepare one textfile, name **vectorX.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
using namespace std;

const int N = 3;
const int C1 = 3;
const int C2 = 3;

float* vecx() {
	static float x[N];
	std::fstream in("vectorX.txt");
	float vectortiles[N];
	for (int i = 0; i < N; ++i)
	{
		in >> vectortiles[i];
		x[i] =vectortiles[i];
	}
	return x;
}

float* vecy() {
	static float y[N];
	std::fstream in("vectorY.txt");
	float vectortiles[N];
	for (int i = 0; i < N; ++i)
	{
		in >> vectortiles[i];
		y[i] =vectortiles[i];
	}
	return y;
}

void displayfloat(float mat[N][N], int row, int col)
{
	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < col; j++)
```

```
                cout<<setw(8) << fixed << setprecision(3) << mat[i][j]<<"\t";

                cout<<endl;
        }
        cout<<endl;
}

void multiplyMatrix(float matrix1[][C1], float matrix2[C1][C2])
{
        float mat_result[N][C2];
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C2; ++j)
                {
                        mat_result[i][j] = 0;
                        for(int k = 0; k<C1; ++k)
                        {
                                mat_result[i][j] += matrix1[i][k]*matrix2[k][j
                                    ];
                        }
                }
        }
        displayfloat(mat_result,N,C2);
}

// Driver code
int main()
{
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        // Create the vandermonde matrix
        float mat[N][N];
        float mat_transpose[N][N];
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        mat[j][i] = pow(ptrvecx[j],i);
                        mat[j][0] = 1;
                }
        }
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
```

```
                        mat_transpose[j][i] = mat[i][j];
                }
        }
        cout << setw(14) << "Matrix M:" << endl;
        displayfloat(mat,N,N);
        cout << setw(14) << "Matrix M^T:" << endl;
        displayfloat(mat_transpose,N,N);

        cout << setw(14) << "Matrix M^T * M:" << endl;
        multiplyMatrix(mat_transpose, mat);

        float mat_result[N][C2];
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C2; ++j)
                {
                        mat_result[i][j] = 0;
                        for(int k = 0; k<C1; ++k)
                        {
                                mat_result[i][j] += mat_transpose[i][k]*mat[k][
                                    j];
                        }
                }
        }
        cout << endl;
        return 0;
}
```

**C++ Code 106:** *main.cpp "Matrices Multiplication"*

To compile it, type:
**g++ -o result main.cpp**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- To create the vandermonde matrix from vector $X$ and then the transpose of that matrix, we first define a 2-dimensional array **float mat** and **float mat_transpose**.

```
        float mat[N][N];
        float mat_transpose[N][N];
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        mat[j][i] = pow(ptrvecx[j],i);
```

```
                        mat[j][0] = 1;
                }
        }
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        mat_transpose[j][i] = mat[i][j];
                }
        }
```

- For simplicity and easeness we create the function **void multiplyMatrix()** with input is matrix with entries in float term. We can just call this function in the main driver **int main()**

```
void multiplyMatrix(float matrix1[][C1], float matrix2[C1][C2])
{
        float mat_result[N][C2];
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C2; ++j)
                {
                        mat_result[i][j] = 0;
                        for(int k = 0; k<C1; ++k)
                        {
                                mat_result[i][j] += matrix1[i][k]*
                                    matrix2[k][j];
                        }
                }
        }
        displayfloat(mat_result,N,C2);
}
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot
mbolicC++/ch23-Numerical Linear Algebra/Matrices Multiplication ]# ./main
     Matrix M:
   1.000          6.100         37.210
   1.000          7.600         57.760
   1.000          8.700         75.690

   Matrix M^T:
   1.000          1.000          1.000
   6.100          7.600          8.700
  37.210         57.760         75.690

Matrix M^T * M:
   3.000         22.400        170.660
  22.400        170.660       1324.460
 170.660       1324.460      10449.776
```

**Figure 23.6:** *The computation for matrices multiplication (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Matrices Multiplication/main.cpp**).*

## III. Determinants of a Matrix

[DF*] Suppose we have a $2 \times 2$ matrix $A$

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the matrix $A$ is invertible if $ad - bc \neq 0$, and the expression $ad - bc$ is called the determinant.

$$\det(A) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Thus, the inverse of $A$ can be expressed in terms of the determinant as

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{23.11}$$

Now, we will find the formula that applicable to square matrices of all orders, not only of size 2.

**Definition 23.11: Minor and Cofactor**

If $A$ is a square matrix, then the minor of entry $a_{ij}$ is denoted by $M_{ij}$ and is defined to be the determinant of the submatrix that remains after the $i$th row and $j$th column are deleted from $A$. The number

$$C_{ij} = (-1)^{i+j} M_{ij} \tag{23.12}$$

is called the cofactor of entry $a_{ij}$.

**Theorem 23.7: Row or Column of Cofactors**

If $A$ is an $n \times n$ matrix, then regardless of which row or column of $A$ is chosen, the number obtained by multiplyinh the entries in that row or column by the corresponding cofactors and adding the resulting products is always the same.

**Definition 23.12: Cofactor Expansion**

If $A$ is an $n \times n$ matrix, then the number obtained by multiplying the entries in any row or column of $A$ by the corresponding cofactors and adding the resulting products is called the determinant of $A$, and the sums themselves are called cofactor expansions of $A$. That is,

$$\det(A) = a_{1j}C_{1j} + a_{2j}C_{2j} + \cdots + a_{nj}C_{nj} \tag{23.13}$$

(cofactor expansion along the $j$th column)

$$\det(A) = a_{i1}C_{i1} + a_{i2}C_{i2} + \cdots + a_{in}C_{in} \tag{23.14}$$

(cofactor expansion along the $i$th row)

[DF*] To find the determinant of size $n$ that is very large will need more computational cost, since we will peel the cofactor expansion along the $i$th row or $j$th column down to the matrix of size $2 \times 2$. It is an ordered-recursive computation technique.

[DF*] The trick to find determinant quickly with cofactor expansion is to choose a row or column that has most zeros, if any exist.

**Theorem 23.8: Determinant of an Upper Triangular Matrix**

If $A$ is an $n \times n$ triangular matrix (upper triangular, lower triangular, or diagonal), then $\det(A)$ is the product of the entries on the main diagonal of the matrix, that is,

$$\det(A) = a_{11}a_{22}\ldots a_{nn}$$

**Theorem 23.9: Row or Column of Zeros in Matrix**

Let $A$ be an $n \times n$ matrix. If $A$ has a row of zeros or a column of zeros, then $\det(A) = 0$.

**Theorem 23.10: Determinant of Transpose**

Let $A$ be a square matrix. Then $\det(A) = \det(A^T)$

**Theorem 23.11: Elementary Row Operations**

Let $A$ be an $n \times n$ matrix.

(a) If $B$ is the matrix that results when a single row or single column of $A$ is multiplied by a scalar $k$, then $\det(B) = k\ \det(A)$.

(b) If $B$ is the matrix that results when two rows or two columns of $A$ are interchanged, then $\det(B) = -\det(A)$.

(c) If $B$ is the amtrix that results when a multiple of one row of $A$ is added to another row or when a multiple of one column is added to another column, then $\det(B) = \det(A)$.

**Theorem 23.12: Elementary Matrices**

Let $E$ be an $n \times n$ elementary matrix.

(a) If $E$ results from multiplying a row of $I_n$ by a nonzero number $k$, then $\det(E) = k$.

(b) If $E$ results from interchanging two rows of $I_n$, then $\det(E) = -1$.

(c) If $E$ results from adding a multiple of one row of $I_n$ to another, then $\det(E) = 1$.

**Theorem 23.13: Matrices with Proportional Rows or Columns**

If $A$ is a square matrix with two proportional rows or two proportional columns, then $\det(A) = 0$

[DF*] Since a common factor of any row of a matrix can be moved through the determinant sign,

and since each of the $n$ rows in $kA$ has a common factor of $k$, it follows that

$$\det(kA) = k^n \det(A) \tag{23.15}$$

**Theorem 23.14: Determinants from Sum of Matrices**

Let $A, B$, and $C$ be an $n \times n$ matrices that differ only in a single row, say the $r$th row of $C$ can be obtained by adding corresponding entries in the $r$th rows of $A$ and $B$. Then

$$\det(C) = \det(A) + \det(B)$$

The same result holds for columns.

**Theorem 23.15: Determinant Test for Invertibility**

A square matrix $A$ is invertible if and only if $\det(A) \neq 0$.

**Theorem 23.16: Determinants from Product of Matrices**

If $A$ and $B$ are square matrices of the same size, then

$$\det(AB) = \det(A)\det(B)$$

There is a lemma saying if $B$ is an $n \times n$ matrix and $E$ is an $n \times n$ elementary matrix, then

$$\det(EB) = \det(E)\det(B)$$

**Theorem 23.17: Determinants from Invers of a Matrix**

If $A$ is invertible, then

$$\det(A^{-1}) = \frac{1}{\det(A)}$$

**Definition 23.13: Adjoint of a Matrix**

If $A$ is any $n \times n$ matrix and $C_{ij}$ is the cofactor of $a_{ij}$, then the matrix

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{bmatrix} \tag{23.16}$$

is called the matrix of cofactors from $A$. The transpose of this matrix is called the adjoint of $A$ and is denoted by $adj(A)$.

$$adj(A) = \begin{bmatrix} C_{11} & C_{21} & \dots & C_{n1} \\ C_{12} & C_{22} & \dots & C_{n2} \\ \vdots & \vdots & & \vdots \\ C_{1n} & C_{2n} & \dots & C_{nn} \end{bmatrix} \tag{23.17}$$

**Theorem 23.18: Invers of a Matrix Using Its Adjoint**

If $A$ is an invertible matrix, then

$$A^{-1} = \frac{1}{\det(A)}\text{adj}(A)$$

**Theorem 23.19: Cramer's Rule**

If $A\boldsymbol{x} = \boldsymbol{b}$ is a system of $n$ linear equations in $n$ unknowns such that $\det(A) \neq 0$, then the system has a unique solution. This solution is

$$x_1 = \frac{\det(A_1)}{\det(A)}, \quad x_2 = \frac{\det(A_2)}{\det(A)}, \ldots \quad x_n = \frac{\det(A_n)}{\det(A)}$$

where $A_j$ is the matrix obtained by replacing the entries in the $j$th column of $A$ by the entries in the matrix

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

**Theorem 23.20: Equivalent Statements**

If $A$ is an $n \times n$, then the following statements are equivalent

(a) $A$ is invertible.

(b) $A\boldsymbol{x} = \boldsymbol{0}$ has only the trivial solution.

(c) The reduced row echelon form of $A$ is $I_n$.

(d) $A$ can be expressed as a product of elementary matrices.

(e) $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for ever $n \times 1$ matrix $\boldsymbol{b}$.

(f) $A\boldsymbol{x} = \boldsymbol{b}$ has exactly one solution for every $n \times 1$ matrix $b$.

(g) $\det(A) \neq 0$.

[DF*]

## i. C++ Computation: Find the Determinant of a Square Matrix

We are going to create C++ code to compute the determinant of a square matrix. The input can be written from the source code, for a small size matrix this code can really come in handy, the C++ code is obtained from:
**https://www.tutorialspoint.com/determinant-of-a-matrix-in-cplusplus**

```
#include <iostream>
using namespace std;
```

```
const int N = 4;
void cofactor(int mat[N][N], int temp[N][N], int p,int q, int n)
{
	int i = 0, j = 0;
	for (int row = 0; row < n; row++)
	{
		for (int column = 0; column < n; column++)
		{
			if (row != p && column != q)
			{
				temp[i][j++] = mat[row][column];
				if (j == n - 1)
				{
					j = 0;
					i++;
				}
			}
		}
	}
}

int determinantOfMatrix(int mat[N][N], int dimension)
{
	int Det = 0;

	if (dimension == 1)
	return mat[0][0];

	int cofactorMat[N][N];
	int sign = 1;

	for (int firstRow = 0; firstRow < dimension; firstRow++)
	{
		cofactor(mat, cofactorMat, 0, firstRow, dimension);
		Det += sign * mat[0][firstRow] * determinantOfMatrix(
		    cofactorMat, dimension - 1);
		sign = -sign;
	}
	return Det;
}

void display(int mat[N][N], int row, int col)
{
	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < col; j++)
		cout<<mat[i][j]<<" ";
		cout<<endl;
```

```
        }
        cout<<endl;
    }

    int main(){
        int mat[4][4] = {
            { 1, 0, 3, 5},
            { 5, 5, 6, 6},
            { 15, 45, 60, 33},
            { 17, 18, 7, 9}};

        cout<<"The matrix A is "<<endl;
        display(mat,4,4);
        cout<<"Determinant of the matrix is "<<determinantOfMatrix(mat, N)
            << endl ;
        return 0;
    }
```

**C++ Code 107:** *main.cpp "Compute Determinant of a Square Matrix"*

To compile it, type:
**g++ -o result main.cpp**
**./result**

Explanation for the codes:

- The determinant of a matrix can be calculated only for a square matrix by multiplying the first row cofactor by the determinant of the corresponding cofactor and adding them with alternate signs to get the final result.
- First, we have the **determinantOfMatrix(int mat[N][N], int dimension)** function that takes the matrix and the dimension value of the matrix. If the matrix is of only 1 dimension then it returns the **[0][0]** matrix value. This condition is also used as a base condition since we recursively iterate our matrix with reducing the dimensions on each recursive call.
- We then declare the **cofactorMat[N][N]** which will be passed to the **cofactor(int mat[N][N], int temp[N][N], int p, int q, int n)** function till the **firstRow** is less then the dimension. The determinant of the matrix is stored in the **Det** variable with signs alternating on each for loop iteration. This det is then returned to the main function where it is printed.
- The **cofactor(int mat[N][N], int temp[N][N], int p,int q, int n)** function takes the matrix, the cofactor matrix, 0, **firstRow** value and the dimension of the matrix as parameter values. The nested for loop then helps us iterate over the matrix and where the **p** & **q** values are not equal to row and column values respectively, those values are stored in the temp matrix.

  Once the row gets filled we increase the row index and reset the col index.
- Finally we have our **display(int mat[N][N], int row, int col)** that takes the matrix and number of rows and columns and iterate over the matrix as 2d array and print those values on each row and column.

We are going to check with Julia as well, just for verification and to check the computation from different programming language, in the end they return the same computation.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Determinant/Test ]# g++ -o result main.c
pp
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Determinant/Test ]# ./result
The matrix is
1 2 3
5 5 6
17 18 9

Determinant of the matrix is 66
```

**Figure 23.7:** *The computation to find the determinant for square matrix $A$ of size $3 \times 3$ with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Determinant of Square Matrix/main.cpp**).*

```
julia> A = [1 2 3;5 5 6; 17 18 9]
3×3 Matrix{Int64}:
  1   2  3
  5   5  6
 17  18  9

julia> using LinearAlgebra

julia> det(A)
66.0
```

**Figure 23.8:** *The computation to find the determinant for square matrix $A$ of size $3 \times 3$ with Julia.*

There is also another code that have been created to compute determinant by loading input from textfile, this can be used when you are in engineering field and take data from sensors that can be stored into textfile and need to be processed directly.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Determinant/Test ]# g++ -o result main.c
pp
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Determinant/Test ]# ./result
The matrix A is
1 0 3 5
5 5 6 6
15 45 60 33
17 18 7 9

Determinant of the matrix is 3834
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Determinant/Test ]#
```

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.2 (2023-07-05)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> using LinearAlgebra

julia> A = [1 0 3 5;5 5 6 6;15 45 60 33; 17 18 7 9]
4×4 Matrix{Int64}:
  1   0   3   5
  5   5   6   6
 15  45  60  33
 17  18   7   9

julia> det(A)
3834.0000000000005
```

**Figure 23.9:** *The computation to find the determinant for square matrix $A$ of size $4 \times 4$ with CPP (above) and Julia below.*

## ii. C++ Computation: Using the Adjoint to Find Numerical Inverse of a Matrix

Suppose we have a $3 \times 3$ matrix $A$

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

with hand we can compute the inverse as

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix}$$

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Compute Determinant of Square Matrix fro
m Textfile ]# ./main
The matrix A is
     1       0       2
    -3       4       6
    -1      -2       3

Determinant of the matrix is 44
```

**Figure 23.10:** *The computation to find the determinant for square matrix $A$ of size $3 \times 3$ loading data from textfile with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Determinant of Square Matrix from Textfile/main.cpp**).*

If $A$ is an invertible matrix, then

$$A^{-1} = \frac{1}{\det(A)}\mathrm{adj}(A)$$

With adjoint $A$ / $adj(A)$ is the transpose of the cofactor matrix and is denoted by

$$adj(A) = \begin{bmatrix} C_{11} & C_{21} & \dots & C_{n1} \\ C_{12} & C_{22} & \dots & C_{n2} \\ \vdots & \vdots & & \vdots \\ C_{1n} & C_{2n} & \dots & C_{nn} \end{bmatrix}$$

With these algorithms we can then construct the C++ code to find the inverse of a matrix.

The code is mainly obtained from:
**https://www.tutorialspoint.com/cplusplus-program-to-find-inverse-of-a-graph-matrix**
then we modify it a bit.

```
#include<bits/stdc++.h>
using namespace std;
#define N 3

void cofactor(int M[N][N], int temp[N][N], int p,int q, int n)
{
	int i = 0, j = 0;
	for (int row = 0; row < n; row++)
	{
		for (int column = 0; column < n; column++)
		{
			if (row != p && column != q)
			{
				temp[i][j++] = M[row][column];
				if (j == n - 1)
				{
					j = 0;
					i++;
				}
			}
		}
	}
}

int determinantOfMatrix(int M[N][N], int dimension)
{
	int Det = 0;

	if (dimension == 1)
	return M[0][0];

	int cofactorMat[N][N];
```

```
        int sign = 1;

        for (int firstRow = 0; firstRow < dimension; firstRow++)
        {
                cofactor(M, cofactorMat, 0, firstRow, dimension);
                Det += sign * M[0][firstRow] * determinantOfMatrix(
                    cofactorMat, dimension - 1);
                sign = -sign;
        }
        return Det;
}

void ADJ(int M[N][N],int adj[N][N])
//to find adjoint matrix
{
        if (N == 1)
        {
                adj[0][0] = 1;
                return;
        }
        int s = 1,
        t[N][N];

        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++)
                {
                        //To get cofactor of M[i][j]
                        cofactor(M, t, i, j, N);
                        s = ((i+j)%2==0)? 1: -1; //sign of adj[j][i] positive
                            if sum of row and column indexes is even.
                        adj[j][i] = (s)*(determinantOfMatrix(t, N-1)); //
                            Interchange rows and columns to get the transpose
                            of the cofactor matrix
                }
        }
}

bool INV(int M[N][N], float inv[N][N])
{
        int det = determinantOfMatrix(M, N);
        if (det == 0)
        {
                cout << "can't find its inverse";
                cout<<endl;
                return false;
        }
        int adj[N][N];
```

```
        ADJ(M, adj);

        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++) inv[i][j] = adj[i][j]/float(det);
        }
        return true;
}

template<class T> void print(T A[N][N]) //print the matrix.
{
        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++)
                {
                        cout << A[i][j] << " ";
                }
                cout<<endl;
        }
}

void display(int M[N][N], int row, int col)
{
        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < col; j++)
                cout<<M[i][j]<<" ";
                cout<<endl;
        }
        cout<<endl;
}

int main()
{
        int M[N][N] = {
                {1,0,0}, {0,-3,0}, {0,0,2}};
        float inv[N][N];
        cout << "The input matrix is :\n";
        display(M,3,3);
        cout << "The matrix determinant is :\n";
        cout << determinantOfMatrix(M, N) << endl;
        //cout << ADJ(M, N) << endl;
        cout << "\nThe Inverse is :\n";
        if (INV(M, inv))
        {
                print(inv);
        }
        return 0;
```

```
}
```

**C++ Code 108:** *main.cpp "Inverse of a Matrix With Adjoint"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp**
**./main**

Explanation for the codes:

- The first two functions **void cofactor()** and **int dewterminantOfMatrix()** are related and used to find the determinant of matrix $A$.

```
void cofactor(int M[N][N], int temp[N][N], int p,int q, int n)
{
	int i = 0, j = 0;
	for (int row = 0; row < n; row++)
	{
		for (int column = 0; column < n; column++)
		{
			if (row != p && column != q)
			{
				temp[i][j++] = M[row][column];
				if (j == n - 1)
				{
					j = 0;
					i++;
				}
			}
		}
	}
}

int determinantOfMatrix(int M[N][N], int dimension)
{
	int Det = 0;

	if (dimension == 1)
	return M[0][0];

	int cofactorMat[N][N];
	int sign = 1;

	for (int firstRow = 0; firstRow < dimension; firstRow++)
	{
		cofactor(M, cofactorMat, 0, firstRow, dimension);
```

```
                        Det += sign * M[0][firstRow] *
                            determinantOfMatrix(cofactorMat, dimension -
                            1);
                        sign = -sign;
                }
                return Det;
        }
```

- The next two functions **void ADJ()** and **bool INV()** (we use bool to determine whether the matrix $A$ has inverse or not, if it has then the inverse will be shown) are related to compute the inverse, the **void ADJ()** will call the function **cofactor()** to obtain the cofactor at each index of the matrix then transpose it to become the adjoint matrix.

  Afterwards, if the matrix has determinant that is not equal to 0, the inverse can be computed inside the function **bool INV()**.

```
void ADJ(int M[N][N],int adj[N][N])
//to find adjoint matrix
{
        if (N == 1)
        {
                adj[0][0] = 1;
                return;
        }
        int s = 1,
        t[N][N];

        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++)
                {
                        //To get cofactor of M[i][j]
                        cofactor(M, t, i, j, N);
                        s = ((i+j)%2==0)? 1: -1; //sign of adj[j][
                            i] positive if sum of row and column
                            indexes is even.
                        adj[j][i] = (s)*(determinantOfMatrix(t, N
                            -1)); //Interchange rows and columns
                            to get the transpose of the cofactor
                            matrix
                }
        }
}

bool INV(int M[N][N], float inv[N][N])
{
        int det = determinantOfMatrix(M, N);
        if (det == 0)
        {
```

```
                cout << "can't find its inverse";
                cout<<endl;
                return false;
        }
        int adj[N][N];
        ADJ(M, adj);

        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++) inv[i][j] = adj[i][j]/
                    float(det);
        }
        return true;
}
```

- The last two functions **void print()** and **void display** are used to print the matrix in a nice orderly manner, with clear look on the matrix row and column.

  The **int main()** will then show all the computations that have been done by the functions above.

```
template<class T> void print(T A[N][N]) //print the matrix.
{
        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++)
                {
                        cout << A[i][j] << " ";
                }
                cout<<endl;
        }
}

void display(int M[N][N], int row, int col)
{
        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < col; j++)
                cout<<M[i][j]<<" ";
                cout<<endl;
        }
        cout<<endl;
}

int main()
{
        int M[N][N] = {
                {1,0,0}, {0,-3,0}, {0,0,2}};
        float inv[N][N];
```

```
                cout << "The input matrix is :\n";
                display(M,3,3);
                cout << "The matrix determinant is :\n";
                cout << determinantOfMatrix(M, N) << endl;
                //cout << ADJ(M, N) << endl;
                cout << "\nThe Inverse is :\n";
                if (INV(M, inv))
                {
                        print(inv);
                }
                return 0;
        }
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Inverse of a Matrix ]# ./main
The input matrix is :
1 0 0
0 -3 0
0 0 2

The matrix determinant is :
-6

The Inverse is :
1 -0 -0
-0 -0.333333 -0
-0 -0 0.5
```

**Figure 23.11:** *The computation for finding the inverse of matrix A using Adjoint with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Inverse of a Matrix/-main.cpp**).*

## iii. C++ Computation: Symbolic Inverse of a Matrix

After we are able to compute inverse for matrix with any kind of numerical entries, now we would want to compute inverse of a matrix with symbolic entries, such as $a, b, c, d$.

Suppose we have a $3 \times 3$ matrix $A$

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

with hand we can compute the inverse as

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

In this case, we can use the C++ library **SymbolicC++**, it is very efficient and we only need to put fewer codes than the one we have done on the numerical inverse of a matrix.

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
        Symbolic a("a");
        Symbolic b("b");
        Symbolic c("c");
        Symbolic d("d");

        Symbolic A = ( ( a, b ),
        ( c, d ) );
        cout << "A = " << A << endl; //
        Symbolic AI = A.inverse();
        cout << "A^{-1} = " << AI << endl;

        return 0;
}
```

**C++ Code 109:** *main.cpp "Symbolic Inverse of a Matrix"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

Explanation for the codes:

- As we only need **int main()** for this, we need to first declare the symbolic one by one, then define the matrix $A$ as symbolic as well.

The function **A.inverse()** have to be defined / declared as symbolic $AI$, then it will compute the inverse of the symbolic matrix automatically from the algorithm already made in the **symbolicc++** library.

```
int main(void)
{
        Symbolic a("a");
        Symbolic b("b");
        Symbolic c("c");
        Symbolic d("d");

        Symbolic A = ( ( a, b ),
        ( c, d ) );
        cout << "A = " << A << endl; //
        Symbolic AI = A.inverse();
        cout << "A^{-1} = " << AI << endl;

        return 0;
}
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Symbolic Inverse ]# ./main
A =
[a b]
[c d]

A^{-1} =
[-d*(-a*d+b*c)^(-1)  b*(-a*d+b*c)^(-1)]
[ c*(-a*d+b*c)^(-1) -a*(-a*d+b*c)^(-1)]
```

**Figure 23.12:** *The computation for finding the symbolic inverse of matrix A with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Symbolic Inverse/-main.cpp**).*

## iv. C++ Computation: Solving Linear Systems with Cramer's Rule

To solve a linear system, we can use Cramer's Rule as one of many method available. Suppose we have this linear system:

$$\begin{array}{rcrcrcl} x_1 & + & & + 2x_3 & = & 6 \\ -3x_1 & & + 4x_2 & + 6x_3 & = & 30 \\ -x_1 & + & -2x_2 & + 3x_3 & = & 8 \end{array}$$

then we will have the matrix $A$

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -3 & 4 & 6 \\ -1 & -2 & 3 \end{bmatrix}$$

with vector $b$

$$\boldsymbol{b} = \begin{bmatrix} 6 \\ 30 \\ 8 \end{bmatrix}$$

and the matrix $A_1, A_2$, and $A_3$

$$A_1 = \begin{bmatrix} 6 & 0 & 2 \\ 30 & 4 & 6 \\ 8 & -2 & 3 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 6 & 2 \\ -3 & 30 & 6 \\ -1 & 8 & 3 \end{bmatrix} \quad A_3 = \begin{bmatrix} 1 & 0 & 6 \\ -3 & 4 & 30 \\ -1 & -2 & 8 \end{bmatrix}$$

Since $\det(A) \neq 0$ the solution is

$$x_1 = \frac{\det(A_1)}{\det(A)}, \quad x_2 = \frac{\det(A_2)}{\det(A)}, \quad x_3 = \frac{\det(A_3)}{\det(A)}$$

the code below will do the computation for finding the solution for the linear system by using Cramer's Rule, instead of writing the matrix and vector manually in the source code, we take the inputs for matrix $A$ and vector $\boldsymbol{b}$ from textfile, **matrixA.txt** and **vectorb.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>

using namespace std;

const int N = 3;

struct matrix
{
        int arr1[N][N];
};

struct matrix func(int N)
{
        struct matrix matrix_mem;
        std::fstream in("matrixA.txt");
        float matrixtiles[N][N];
```

```
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        in >> matrixtiles[i][j];
                        matrix_mem.arr1[i][j] = matrixtiles[i][j] ;
                }
        }
        return matrix_mem;
}

// take vectorb.txt as input from textfile
int* vec() {
        static int x[N];
        std::fstream in("vectorb.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

void cofactor(float mat[N][N], float temp[N][N], int p,int q, int n)
{
        int i = 0, j = 0;
        for (int row = 0; row < n; row++)
        {
                for (int column = 0; column < n; column++)
                {
                        if (row != p && column != q)
                        {
                                temp[i][j++] = mat[row][column];
                                if (j == n - 1)
                                {
                                        j = 0;
                                        i++;
                                }
                        }
                }
        }
}

int determinantOfMatrix(float mat[N][N], int dimension)
{
        float Det = 0;
```

```
        if (dimension == 1)
        return mat[0][0];

        float cofactorMat[N][N];
        int sign = 1;

        for (int firstRow = 0; firstRow < dimension; firstRow++)
        {
                cofactor(mat, cofactorMat, 0, firstRow, dimension);
                Det += sign * mat[0][firstRow] * determinantOfMatrix(
                    cofactorMat, dimension - 1);
                sign = -sign;
        }
        return Det;
}

void display(float mat[N][N], int row, int col)
{
        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < col; j++)
                cout<< setw(6) << mat[i][j]<<"\t";
                cout<<endl;
        }
        cout<<endl;
}

int main(){
        // construct matrix A
        float mat[N][N] = {};
        // contruct vector b
        std::vector<std::pair<double, double>> xy_pts;
        int* ptrvec;
        ptrvec = vec();

        struct matrix a;
        a = func(N); // address of arr1
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        mat[i][j] = a.arr1[i][j];
                }
        }

        // construct matrix A1
        float mat1[N][N] = {0};
        for(int i=0; i<N; i++)
```

```
{
        for(int j=0; j<N; j++)
        {
                mat1[i][j] = mat[i][j];
                mat1[i][0] = ptrvec[i];
        }
}
// construct matrix A2
float mat2[N][N] = {0};
for(int i=0; i<N; i++)
{
        for(int j=0; j<N; j++)
        {
                mat2[i][j] = mat[i][j];
                mat2[i][1] = ptrvec[i];
        }
}
// construct matrix A3
float mat3[N][N] = {0};
for(int i=0; i<N; i++)
{
        for(int j=0; j<N; j++)
        {
                mat3[i][j] = mat[i][j];
                mat3[i][2] = ptrvec[i];
        }
}
int r=0, c=0;// cofactor of M
float cofactorMat[N][N];
int sign = 1;

cout<<"The matrix A is "<<endl;
display(mat,N,N);

cout << "Vector b is: " << endl;
for (int i = 0; i < N; ++i)
{
        cout <<ptrvec[i]<< ' ' << endl;
}
cout<< endl ;
cout<<"The matrix A1 is "<<endl;
display(mat1,N,N);
cout<< endl ;
cout<<"The matrix A2 is "<<endl;
display(mat2,N,N);
cout<< endl ;
cout<<"The matrix A3 is "<<endl;
display(mat3,N,N);
```

```
            cout<< endl ;

            cout<<"Determinant of the matrix A is "<<determinantOfMatrix(mat, N)
                 << endl ;
            cout<<"Determinant of the matrix A1 is "<<determinantOfMatrix(mat1,
                N) << endl ;
            cout<<"Determinant of the matrix A2 is "<<determinantOfMatrix(mat2,
                N) << endl ;
            cout<<"Determinant of the matrix A3 is "<<determinantOfMatrix(mat3,
                N) << endl ;
            cout<< endl ;

            float x1 = float(determinantOfMatrix(mat1, N))/float(
                determinantOfMatrix(mat, N)) ;
            float x2 = float(determinantOfMatrix(mat2, N))/float(
                determinantOfMatrix(mat, N)) ;
            float x3 = float(determinantOfMatrix(mat3, N))/float(
                determinantOfMatrix(mat, N)) ;
            cout<<"The solution is: "<< endl ;
            cout<<"x1 = "<< fixed << setprecision(6) << x1 << endl ;
            cout<<"x2 = "<< fixed << setprecision(6) << x2 << endl ;
            cout<<"x3 = "<< fixed << setprecision(6) << x3 << endl ;

            return 0;
        }
```

**C++ Code 110:** *main.cpp "Solving Linear Systems With Cramer's Rule"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp**
**./main**

The code is very intuitive and I believe the reader won't have a hard time to comprehend it.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algel
a/Cramer's Rule to Solve Linear System ]# ./main
The matrix A is
     1       0       2
    -3       4       6
    -1      -2       3

Vector b is:
6
30
8

The matrix A1 is
     6       0       2
    30       4       6
     8      -2       3


The matrix A2 is
     1       6       2
    -3      30       6
    -1       8       3


The matrix A3 is
     1       0       6
    -3       4      30
    -1      -2       8


Determinant of the matrix A is 44
Determinant of the matrix A1 is -40
Determinant of the matrix A2 is 72
Determinant of the matrix A3 is 152

The solution is:
x1 = -0.909091
x2 = 1.636364
x3 = 3.454545
```

**Figure 23.13:** *The computation for finding the solution for a linear system with Cramer's Rule and C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Cramer's Rule to Solve Linear System/main.cpp**).*

## v. C++ Computation: Solving Linear Systems with Gauss-Jordan Elimination

Linear systems of equations are associated with many problems in engineering and science, as well as with applications of mathematics to the social sciences and the quantitative study of business and economic problems.

With Gauss-Jordan Elimination (a modified version of Gauss Elimination Method) we can solve linear systems easily, we will have to form an upper triangular matrix (a matrix with reduced row echelon form) / diagonal matrix from the linear systems we want to solve.

The C++ code is obtained from:
**https://www.geeksforgeeks.org/program-for-gauss-jordan-elimination-method/**
Consider this linear system

$$\begin{array}{lllll} E_1: & a_{11}x_1 & +\,a_{12}x_2 & +\dots & +\,a_{1n}x_n & = b_1 \\ E_2: & a_{21}x_1 & +\,a_{22}x_2 & +\dots & +\,a_{2n}x_n & = b_2 \\ & \vdots & \vdots & \dots & \vdots & \vdots \\ E_n: & a_{n1}x_1 & +\,a_{n2}x_2 & +\dots & +\,a_{nn}x_n & = b_n \end{array}$$

for $x_1, x_2, \dots, x_n$, given the constants $a_{ij}$, for each $i, j = 1, 2, \dots, n$ and $b_i$ for each $i = 1, 2, \dots, n$.

We then use three operations to simplify the linear system above:

1. Equation $E_i$ can be multiplied by any nonzero constant $\lambda$ with the resulting equation used in place of $E_i$. This operation is denoted by

$$(\lambda E_i) \to (E_i)$$

2. Equation $E_j$ can be multiplied by any constant $\lambda$ and added to equation $E_i$ with the resulting equation used in place of $E_i$. This operation is denoted by

$$(E_i + \lambda E_j) \to (E_i)$$

3. Equations $E_i$ and $E_j$ can be swapped / transposed in order. This operation is denoted by

$$(E_i) \leftrightarrow (E_j)$$

In order to solve :

$$\begin{array}{llll} x_1 & + & +2x_3 & = 6 \\ -3x_1 & +4x_2 & +6x_3 & = 30 \\ -x_1 & -2x_2 & +3x_3 & = 8 \end{array}$$

we can turn the linear system into a matrix to make the computation easier, thus we will have matrix $A$ as

$$\begin{bmatrix} 1 & 0 & 2 \\ -3 & 4 & 6 \\ -1 & -2 & 3 \end{bmatrix}$$

and the augmented matrix $A$ is

$$\begin{bmatrix} 1 & 0 & 2 & 6 \\ -3 & 4 & 6 & 30 \\ -1 & -2 & 3 & 8 \end{bmatrix}$$

the solution of the linear system $x_1, x_2, \ldots, x_n$ can be found by reducing this augmented matrix to reduced row echelon form with Gauss-Jordan elimination.

Applications for Gauss-Jordan elimination:

- Solving System of Linear Equations: Gauss-Jordan Elimination Method can be used for finding the solution of a systems of linear equations which is applied throughout the mathematics.
- Finding Determinant: The Gaussian Elimination can be applied to a square matrix in order to find determinant of the matrix.
- Finding Inverse of Matrix: The Gauss-Jordan Elimination method can be used in determining the inverse of a square matrix.
- Finding Ranks and Bases: Using reduced row echelon form, the ranks as well as bases of square matrices can be computed by Gaussian elimination method.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

#define M 10

// Function to print the matrix
void PrintMatrix(float a[][M], int n)
{
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j <= n; j++)
		cout << a[i][j] << " ";
		cout << endl;
	}
}

// function to reduce matrix to reduced
// row echelon form.
int PerformOperation(float a[][M], int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < n; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < n && a[i + c][i] == 0)
			c++;
			if ((i + c) == n)
```

```
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

// Function to print the desired result
// if unique solutions exists, otherwise
// prints no solution or infinite solutions
// depending upon the input given.
void PrintResult(float a[][M], int n, int flag)
{
        cout << "The solution/s : ";

        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "x(" << i << ") = "<< a[i][n] / a[i][i] << ", ";
        }
```

```
}

// To check whether infinite solutions
// exists or no solution exists
int CheckConsistency(float a[][M], int n, int flag)
{
	int i, j;
	float sum;

	// flag == 2 for infinite solution
	// flag == 3 for No solution
	flag = 3;
	for (i = 0; i < n; i++)
	{
		sum = 0;
		for (j = 0; j < n; j++)
		sum = sum + a[i][j];
		if (sum == a[i][j])
		flag = 2;
	}
	return flag;
}

// Driver code
int main()
{
	float a[M][M] = {{ 1, 0, 2, 6 },
		{ -3, 4, 6, 30 },
		{ -1, -2, 3, 8 }};

	// Printing matrix A
	cout << "Matrix A : " << endl;
	int n1 = 4;
	for (int i = 0; i < n1; ++i)
	{
		for (int j = 0; j < n1; ++j)
		{
			cout << a[i][j]<< ' ';
		}
		cout << endl;
	}

	// Order of Matrix(n)
	int n = 3, flag = 0;

	// Performing Matrix transformation
	flag = PerformOperation(a, n);
```

```
        if (flag == 1)
        flag = CheckConsistency(a, n, flag);

        // Printing Final Matrix
        cout << "Final Augmented Matrix is : " << endl;
        PrintMatrix(a, n);
        cout << endl;

        // Printing Solutions(if exist)
        cout << "Solutions for Matrix A : " << endl;
        PrintResult(a,n, flag);
        cout << endl;

        return 0;
    }
```

**C++ Code 111:** *main.cpp "Gauss-Jordan Elimination"*

To compile it, type:
**g++ -o result main.cpp**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- In order to print the matrix $A$, we need to put the codes before the function to perform the Matrix transformation

```
int main()
{
    float a[M][M] = {{ 1, 0, 2, 6 },
        { -3, 4, 6, 30 },
        { -1, -2, 3, 8 }};

    // Printing matrix A
    cout << "Matrix␣A␣:␣" << endl;
    int n1 = 4;
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n1; ++j)
        {
            cout << a[i][j]<< '␣';
        }
        cout << endl;
    }
...
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination ]# make
g++    -c -o main.o main.cpp
g++ -o main -ggdb main.o -lstdc++
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination ]# ./main
Matrix A :
1 0 2 6
-3 4 6 30
-1 -2 3 8
0 0 0 0

Final Augmented Matrix is :
1 0 0 -0.909091
0 4 0 6.54545
0 0 11 38

Solutions for Matrix A :
The solution/s : x(0) = -0.909091,  x(1) = 1.63636,  x(2) = 3.45455,
```

**Figure 23.14:** *The computation to find reduced row echelon form with Gauss-Jordan elimination for linear systems represented by an augmented matrix $A$ of size $3 \times 4$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination/main.cpp**).*

## IV. Euclidean Vector Spaces

[DF*] Engineers and physicists represent vectors in two dimensions or in three dimensions by arrows. The direction of the arrowhead specifies the direction of the vector and the length of the arrow specifies the magnitude. The tail of the arrow is called the initial point of the vector and the tip the terminal point.

We usually denote vectors with boldface such as $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{v}, \boldsymbol{w}$ or with an arrow above it $a, b, \vec{v}, w$, and we denote scalars in lowercase letters like $a, b, c, d$. When we want to indicate that a vector has initial point $A$ and terminal point $B$, we will write

$$\boldsymbol{v} = \vec{AB}$$

Vectors with the same length and direction are said to be equivalent. Even if they are located at different position.

[DF*] The vector whose initial and terminal points coincide has length zero, so we call this the zero vector and denote it by $\mathbf{0}$.

[DF*] There are a number of important algebraic operations on vectors, all of which we have their origin in laws of physics

1. **Parallelogram Rule for Vector Addition**
   If $\boldsymbol{v}$ and $\boldsymbol{w}$ are vectors in 2-space and 3-space that are positioned so their intial points coincide, then the two vectors form adjacent sides of a parallelogram, and the sum

   $$\boldsymbol{v} + \boldsymbol{w}$$

   is the vector represented by the arrow from the common initial point of $\boldsymbol{v}$ and $\boldsymbol{w}$ to the opposite vertex of the parallelogram.
2. **Triangle Rule for Vector Addition**
   If $\boldsymbol{v}$ and $\boldsymbol{w}$ are vectors in 2-space or 3-space that are positioned so the initial point of $\boldsymbol{w}$ is at the terminal point of $\boldsymbol{v}$, then the sum

   $$\boldsymbol{v} + \boldsymbol{w}$$

is represented by the arrow from the initial point of $\boldsymbol{v}$ to the terminal point of $\boldsymbol{w}$.

[DF*] The negative of a vector $\boldsymbol{v}$, denoted by $-\boldsymbol{v}$, is the vector that has the same length as $\boldsymbol{v}$ but is oppositely directed, and the difference of $\boldsymbol{v}$ from $\boldsymbol{w}$, is taken to be the sum

$$\boldsymbol{w} - \boldsymbol{v} = \boldsymbol{w} + (-\boldsymbol{v})$$

[DF*] If $\boldsymbol{v}$ is a nonzero vector in 2-space or 3-space, and if $k$ is a nonzero scalar, then we define the scalar product of $\boldsymbol{v}$ by $k$ to be the vector whose length is $|k|$ times the length of $\boldsymbol{v}$ and whose direction is the same as that of $\boldsymbol{v}$ if $k$ is positive and opposite to that of $\boldsymbol{v}$ if $k$ is negative.

If $k = 0$ or $\boldsymbol{v} = \mathbf{0}$, then we define $k\boldsymbol{v}$ to be $\mathbf{0}$.

[DF*] Translating a vector does not change it.

[DF*] Suppose that $\boldsymbol{v}$ and $\boldsymbol{w}$ are vectors in 2-space and 3-space with a common initial point. If one of the vectors is a scalar multiple of the other, then the vectors lie on a common line, or can be said that they are collinear.

[DF*] The vector addition satisfies the associative law for addition.

$$\boldsymbol{u} + (\boldsymbol{v} + \boldsymbol{w}) = (\boldsymbol{u} + \boldsymbol{v}) + \boldsymbol{w}$$

[DF*] Computation with vectors are much simpler to perform if a coordinate system is present to work with. We write

$$\boldsymbol{v} = (v_1, v_2)$$

to denote a vector $\boldsymbol{v}$ in 2-space with components $(v_1, v_2)$, and

$$\boldsymbol{v} = (v_1, v_2, v_3)$$

to denote a vector $\boldsymbol{v}$ in 3-space with components $(v_1, v_2, v_3)$.

[DF*] The vector in 2-space with initial point $P_1(x_1, y_1)$ and terminal point $P_2(x_2, y_2)$, the components can be written as

$$\vec{P_1P_2} = (x_2 - x_1, y_2 - y_1)$$

For a vector in 3-space that has initial point $P_1(x_1, y_1, z_1)$ and terminal point $P_2(x_2, y_2, z_2)$ the components can be written as

$$\vec{P_1P_2} = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

[DF*] We denote a vector in $\mathbb{R}^n$ using the notation

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$$

and we will call

$$\mathbf{0} = (0, 0, \ldots, 0)$$

the zero vector.

**Definition 23.14: Equivalent Vector**

Vectors $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ and $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$ in $\mathbb{R}^n$ are said to be equivalent if

$$v_1 = w_1,\ v_2 = w_2,\ \ldots,\ v_n = w_n$$

we indicate this by writing $\boldsymbol{v} = \boldsymbol{w}$.

**Definition 23.15: Operations on Vectors in $\mathbb{R}^n$**

If $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ and $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$ are vectors in $\mathbb{R}^n$ and if $k$ is any scalar, then we define

$$\boldsymbol{v} + \boldsymbol{v} = (v_1 + w_1, v_2 + w_2, \ldots, v_n + w_n) \tag{23.18}$$

$$k\boldsymbol{v} = (kv_1, kv_2, \ldots, kv_n) \tag{23.19}$$

$$-\boldsymbol{v} = (-v_1, -v_2, \ldots, -v_n) \tag{23.20}$$

$$\boldsymbol{w} - \boldsymbol{v} = \boldsymbol{w} + (-\boldsymbol{v}) = (w_1 - v_1, w_2 - v_2, \ldots, w_n - v_n) \tag{23.21}$$

**Theorem 23.21: Properties of Vectors in $\mathbb{R}^n$**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in $\mathbb{R}^n$, and if $k$ and $m$ are scalars, then

(a) $\boldsymbol{u} + \boldsymbol{v} = \boldsymbol{v} + \boldsymbol{u}$

(b) $(\boldsymbol{u} + \boldsymbol{v}) + \boldsymbol{w} = \boldsymbol{u} + (\boldsymbol{v} + \boldsymbol{w})$

(c) $\boldsymbol{v} + \mathbf{0} = \mathbf{0} + \boldsymbol{v} = \boldsymbol{v}$

(d) $\boldsymbol{v} + (-\boldsymbol{v}) = \mathbf{0}$

(e) $k(\boldsymbol{u} + \boldsymbol{v}) = k\boldsymbol{u} + k\boldsymbol{v}$

(f) $(k + m)\boldsymbol{v} = k\boldsymbol{v} + m\boldsymbol{v}$

(g) $k(m\boldsymbol{v}) = (km)\boldsymbol{v}$

(h) $1\boldsymbol{v} = \boldsymbol{v}$

The following additional properties of vectors in $\mathbb{R}^n$ can be deduced easily by expressing the vectors in terms of components.

(a) $0\boldsymbol{v} = \mathbf{0}$

(b) $k\mathbf{0} = \mathbf{0}$

(c) $(-1)\boldsymbol{v} = -\boldsymbol{v}$

**Definition 23.16: Linear Combination**

If $\boldsymbol{w}$ is a vector in $\mathbb{R}^n$, then $\boldsymbol{w}$ is said to be a linear combination of the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r$ in $\mathbb{R}^n$ if it can be expressed in the from

$$\boldsymbol{w} = k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r \tag{23.22}$$

where $k_1, k_2, \ldots, k_r$ are scalars. These scalars are called the coefficients of the linear combination. In the case where $r = 1$, it becomes $\boldsymbol{w} = k_1\boldsymbol{v}_1$, so that a linear combination of a single vector is just a scalar multiple of that vector.

[DF*] The comma-delimited form when writing vectors in $\mathbb{R}^n$ is

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$$

The row-matrix form to write vector has notation as

$$\boldsymbol{v} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$$

The column-matrix form to write vector has notation as

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

[DF*] We denote the length of a vector $\boldsymbol{v}$ by the symbol $||\boldsymbol{v}||$, which is read as norm of $\boldsymbol{v}$, or the magnitude of $\boldsymbol{v}$.

**Definition 23.17: Norm**

If $\boldsymbol{v} = (v_1, v_2, \dots, v_n)$ is a vector in $\mathbb{R}^n$, then the norm of $\boldsymbol{v}$ is denoted by $||\boldsymbol{v}||$, and is defined by the formula

$$||\boldsymbol{v}|| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \tag{23.23}$$

**Theorem 23.22: Norm for Vectors in $\mathbb{R}^n$**

If $\boldsymbol{v}$ is a vector in $\mathbb{R}^n$, and if $k$ is any scalar, then:

(a) $||\boldsymbol{v}|| \geq 0$

(b) $||\boldsymbol{v}|| = 0$ if and only if $\boldsymbol{v} = \mathbf{0}$

(c) $||k\boldsymbol{v}|| = |k|||\boldsymbol{v}||$

[DF*] A vector of norm 1 is called a unit vector. Such vectors are useful for specifying a direction when length is not relevant to the problem at hand. If $\boldsymbol{v}$ is any nonzero vector in $\mathbb{R}^n$, then

$$\boldsymbol{u} = \frac{1}{||\boldsymbol{v}||}\boldsymbol{v} \tag{23.24}$$

defines a unit vector that is in the same direction as $\boldsymbol{v}$. The process of multiplying a nonzero vector by the reciprocal of its length to obtain a unit vector is called normalizing $\boldsymbol{v}$.

[DF*] The unit vectors in the positive directions of the coordinate axes are called the standard unit vectors, in $\mathbb{R}^2$ these vectors are denoted by

$$\boldsymbol{i} = (1, 0), \quad \boldsymbol{j} = (0, 1)$$

and in $\mathbb{R}^3$ by

$$\boldsymbol{i} = (1, 0, 0), \quad \boldsymbol{j} = (0, 1, 0), \quad \boldsymbol{k} = (0, 0, 1)$$

The standard unit vectors in $\mathbb{R}^n$ is

$$\boldsymbol{e_1} = (1, 0, 0, \dots, 0), \quad \boldsymbol{e_2} = (0, 1, 0, \dots, 0), \dots, \ \boldsymbol{e_n} = (0, 0, 0, \dots, 1)$$

in which case every vector $\boldsymbol{v} = (v_1, v_2, \dots, v_n)$ in $\mathbb{R}^n$ can be expressed as

$$\boldsymbol{v} = (v_1, v_2, \dots, v_n) = v_1\boldsymbol{e_1} + v_2\boldsymbol{e_2} + \cdots + v_n\boldsymbol{e_n} \tag{23.25}$$

**Definition 23.18: Distance in $\mathbb{R}^n$**

If $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ are points in $\mathbb{R}^n$, then we denote the distance between $\boldsymbol{u}$ and $\boldsymbol{v}$ by $d(\boldsymbol{u}, \boldsymbol{v})$ and define it to be

$$d(\boldsymbol{u}, \boldsymbol{v}) = ||\boldsymbol{u} - \boldsymbol{v}|| = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_n - v_n)^2} \tag{23.26}$$

**Definition 23.19: Angle Between Two Vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are nonzero vectors in $\mathbb{R}^2$ or $\mathbb{R}^3$, and if $\theta$ is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$, then the dot product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is denoted by $\boldsymbol{v} \cdot \boldsymbol{v}$ and is defined as

$$\boldsymbol{u} \cdot \boldsymbol{v} = ||\boldsymbol{u}||||\boldsymbol{v}|| \ \cos \ \theta \tag{23.27}$$

If $\boldsymbol{u} = \mathbf{0}$ or $\boldsymbol{v} = \mathbf{0}$, then we define $\boldsymbol{u} \cdot \boldsymbol{v}$ to be 0.

Since $0 \leq \theta \leq \pi$, it follows from the properties of the cosine function that

- $\theta$ is acute if $\boldsymbol{u} \cdot \boldsymbol{v} > 0$.
- $\theta$ is obtuse if $\boldsymbol{u} \cdot \boldsymbol{v} < 0$.
- $\theta = \frac{\pi}{2}$ if $\boldsymbol{u} \cdot \boldsymbol{v} = 0$.

**Definition 23.20: Dot Product in $\mathbb{R}^n$**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{R}^n$, then the dot product or the Euclidean inner product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is denoted by $\boldsymbol{u} \cdot \boldsymbol{v}$ and is defined by

$$\boldsymbol{u} \cdot \boldsymbol{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n \tag{23.28}$$

[DF*] The formula for expressing the length of a vector in terms of a dot product:

$$||\boldsymbol{v}|| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}} \tag{23.29}$$

**Theorem 23.23: Algebraic Properties of Dot Product**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in $\mathbb{R}^n$, and if $k$ is a scalar, then:

(a) $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{u}$

(b) $\boldsymbol{u} \cdot (\boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{u} \cdot \boldsymbol{w}$

(c) $k(\boldsymbol{u} \cdot \boldsymbol{v}) = (k\boldsymbol{u}) \cdot \boldsymbol{v}$

(d) $\boldsymbol{v} \cdot \boldsymbol{v} \geq 0$ and $\boldsymbol{v} \cdot \boldsymbol{v} = 0$ if and only if $\boldsymbol{v} = \mathbf{0}$

(e) $\mathbf{0} \cdot \boldsymbol{v} = \boldsymbol{v} \cdot \mathbf{0} = 0$

(f) $(\boldsymbol{u} + \boldsymbol{v}) \cdot \boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{w} + \boldsymbol{v} \cdot \boldsymbol{w}$

(g) $\boldsymbol{u} \cdot (\boldsymbol{v} - \boldsymbol{w}) = \boldsymbol{u} \cdot \boldsymbol{v} - \boldsymbol{u} \cdot \boldsymbol{w}$

(h) $(\boldsymbol{u} - \boldsymbol{v}) \cdot \boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{w} - \boldsymbol{v} \cdot \boldsymbol{w}$

(i) $k(\boldsymbol{u} \cdot \boldsymbol{v}) = \boldsymbol{u} \cdot (k\boldsymbol{v})$

**Theorem 23.24: Cauchy-Schwarz Inequality**

If $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ are vectors in $\mathbb{R}^n$, then

$$|\boldsymbol{u} \cdot \boldsymbol{v}| \leq ||\boldsymbol{u}||||\boldsymbol{v}|| \tag{23.30}$$

or in terms of components

$$|u_1 v_1 + u_2 v_2 + \cdots + u_n v_n| \leq (u_1^2 + u_2^2 + \ldots u_n^2)^{1/2} \, (v_1^2 + v_2^2 + \ldots v_n^2)^{1/2} \tag{23.31}$$

[DF*] Since dot products and norms have been derived for vectors in $\mathbb{R}^n$, thus the formula

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{u} \cdot \boldsymbol{u}}{||\boldsymbol{u}||||\boldsymbol{v}||}\right)$$

has all the ingredients to serve as a definition of the angle $\theta$ between two vectors, $\boldsymbol{u}$ and $\boldsymbol{v}$, in $\mathbb{R}^n$. The formula above is not defined unless its argument satisfies the inequalities

$$-1 \leq \frac{\boldsymbol{u} \cdot \boldsymbol{u}}{||\boldsymbol{u}||||\boldsymbol{v}||} \leq 1$$

These inequalities do hold for all nonzero vectors in $\mathbb{R}^n$.

[DF*] Two fundamental theorems from plane geometry whose validity extends to $\mathbb{R}^n$:

1. The sum of the lengths of two side of a triangle is at least as large as the third

2. The shortest distance between two points is a straight line

**Theorem 23.25: Geometry in $\mathbb{R}^n$**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in $\mathbb{R}^n$, and if $k$ is any scalar, then:

(a) $||\boldsymbol{u}+\boldsymbol{v}|| \leq ||\boldsymbol{u}|| + ||\boldsymbol{v}||$

(b) $d(\boldsymbol{u}, \boldsymbol{v}) \leq d(\boldsymbol{u}, \boldsymbol{w}) + d(\boldsymbol{w}, \boldsymbol{v})$

**Theorem 23.26: Parallelogram Equation for Vectors**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{R}^n$, then:

$$||\boldsymbol{u}+\boldsymbol{v}||^2 + ||\boldsymbol{u}-\boldsymbol{v}||^2 = 2(||\boldsymbol{u}||^2 + ||\boldsymbol{v}||^2) \tag{23.32}$$

**Theorem 23.27: Dot Product and Norm in $\mathbb{R}^n$**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{R}^n$ with the Euclidean inner product, then:

$$\boldsymbol{u} \cdot \boldsymbol{v} = \frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2 \tag{23.33}$$

[DF*] An important link between multiplication by an $n \times n$ matrix $A$ and multiplication by $A^T$

$$A\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \cdot A^T\boldsymbol{v} \tag{23.34}$$

$$\boldsymbol{u} \cdot A\boldsymbol{v} = A^T\boldsymbol{u} \cdot \boldsymbol{v} \tag{23.35}$$

[DF*] If the row vectors of $A$ (an $m \times r$ matrix) are $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_m$ and the column vectors of $B$ (an $r \times n$ matrix) are $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_m$, then the matrix product $AB$ can be expressed as

$$AB = \begin{bmatrix} \boldsymbol{r}_1 \cdot \boldsymbol{c}_1 & \boldsymbol{r}_1 \cdot \boldsymbol{c}_2 & \ldots & \boldsymbol{r}_1 \cdot \boldsymbol{c}_n \\ \boldsymbol{r}_2 \cdot \boldsymbol{c}_1 & \boldsymbol{r}_2 \cdot \boldsymbol{c}_2 & \ldots & \boldsymbol{r}_2 \cdot \boldsymbol{c}_n \\ \vdots & \vdots & & \vdots \\ \boldsymbol{r}_m \cdot \boldsymbol{c}_1 & \boldsymbol{r}_m \cdot \boldsymbol{c}_2 & \ldots & \boldsymbol{r}_m \cdot \boldsymbol{c}_n \end{bmatrix} \tag{23.36}$$

**Definition 23.21: Orthogonal and Orthonormal**

Two nonzero vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ are said to be orthogonal (or perpendicular) if $\boldsymbol{u} \cdot \boldsymbol{v} = 0$.

The zero vector $(\mathbf{0})$ in $\mathbb{R}^n$ is orthogonal to every vector in $\mathbb{R}^n$.

A nonempty set of vectors in $\mathbb{R}^n$ is called an orthogonal set if all pairs of distinct vectors in the set are orthogonal.

An orthogonal set of unit vectors is called an orthonormal set.

[DF*] In analytic geometry a line in $\mathbb{R}^2$ is determined uniquely by its slope and one of its points. A plane in $\mathbb{R}^3$ is determined uniquely by its "inclination" and one of its points.

One way of specifying slope and inclination is to use a nonzero vector $\boldsymbol{n}$, called a normal, that is orthogonal to the corresponding line or plane.

[DF*] The vector equation

$$\boldsymbol{n} \cdot \vec{P_0P} = 0$$

is the representation of of a line through the point $P_0(x_0, y_0)$ that has normal $\boldsymbol{n} = (a, b)$ or a plane through the point $P_0(x_0, y_0, z_0)$ that has normal $\boldsymbol{n} = (a, b, c)$.

For line, the vector $\vec{P_0P}$ can be expressed in terms of components as

$$\vec{P_0P} = (x - x_0, y - y_0)$$

For plane, the vector $\vec{P_0P}$ can be expressed in terms of components as

$$\vec{P_0P} = (x - x_0, y - y_0, z - z_0)$$

[DF*] The point-normal equation of the line:

$$a(x - x_0) + b(y - y_0) = 0 \tag{23.37}$$

**Theorem 23.28: Equation for Line and Plane**

If $a$ and $b$ are constants that are not both zero, then an equation of the form

$$ax + by + c = 0 \tag{23.38}$$

represents a line in $\mathbb{R}^2$ with normal $\boldsymbol{n} = (a, b)$.

If $a, b$, and $c$ are constants that are not all zero, then an equation of the form

$$ax + by + cz + d = 0 \tag{23.39}$$

represents a plane in $\mathbb{R}^3$ with normal $\boldsymbol{n} = (a, b, c)$.

[DF*] Recall that

$$ax + by = 0$$

and

$$ax + by + cz = 0$$

are called homogeneous equations. Homogeneous equations in two or three unknowns can be written in the vector form

$$\boldsymbol{n} \cdot \boldsymbol{x} \tag{23.40}$$

where $\boldsymbol{n}$ is the vector of coefficients and $\boldsymbol{x}$ is the vector of unknowns. In $\mathbb{R}^2$ this is called the vector form of a line through the origin, and in $\mathbb{R}^3$ it is called the vector form of a plane through the origin.

[DF*] In many applications it is necessary to "decompose" a vector $\boldsymbol{u}$ into a sum of two terms, one term being a scalar multiple of a specified nonzero vector $\boldsymbol{a}$ and the other term being orthogonal to $\boldsymbol{a}$.

**Figure 23.15:** *The three different cases for orthogonal projection, when we drop the head / tip of vector $\boldsymbol{u}$ to the line through $\boldsymbol{a}$ we will obtain $\boldsymbol{w}_1$ that has the head / tip at the same x axis as vector $\boldsymbol{u}$, then we can construct vector $\boldsymbol{w}_2 = \boldsymbol{u} - \boldsymbol{w}_1$.*

**Theorem 23.29: Projection Theorem**

If $\boldsymbol{u}$ and $\boldsymbol{a}$ are vectors in $\mathbb{R}^n$, and if $\boldsymbol{a} \neq 0$, then $\boldsymbol{u}$ can be expressed in exactly one way in the form

$$\boldsymbol{u} = \boldsymbol{w}_1 + \boldsymbol{w}_2$$

where $\boldsymbol{w}_1$ is a scalar multiple of $\boldsymbol{a}$ and $\boldsymbol{w}_2$ is orthogonal to $\boldsymbol{a}$.

[DF*] The vector $\boldsymbol{w}_1$ is called the orthogonal projection of $\boldsymbol{u}$ on $\boldsymbol{a}$, and the vector $\boldsymbol{w}_2$ is called the vector component of $\boldsymbol{u}$ orthogonal to $\boldsymbol{a}$.
The vector $\boldsymbol{w}_1$ is commonly denoted by the symbol $proj_{\boldsymbol{a}}\boldsymbol{u}$

$$\boldsymbol{w}_1 = proj_{\boldsymbol{a}}\boldsymbol{u} = \frac{\boldsymbol{u} \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}\boldsymbol{a} \tag{23.41}$$

and the vector $\boldsymbol{w}_2$

$$\boldsymbol{w}_2 = \boldsymbol{u} - proj_{\boldsymbol{a}}\boldsymbol{u} = \boldsymbol{u} - \frac{\boldsymbol{u} \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}\boldsymbol{a} \tag{23.42}$$

[DF*] Sometimes we will be more interested in the norm of the vector component of $\boldsymbol{u}$ along $\boldsymbol{a}$ than in the vector component itself. A formula can be derived as follows:

$$\begin{aligned} ||proj_{\boldsymbol{a}}\boldsymbol{u}|| &= ||\frac{\boldsymbol{u} \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}\boldsymbol{a}|| \\ &= |\frac{\boldsymbol{u} \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}|||\boldsymbol{a}|| \\ &= \frac{|\boldsymbol{u} \cdot \boldsymbol{a}|}{||\boldsymbol{a}||^2}||\boldsymbol{a}|| \end{aligned}$$

From the fact we know that $||\boldsymbol{a}||^2 > 0$, thus

$$||proj_{\boldsymbol{a}}\boldsymbol{u}|| = \frac{|\boldsymbol{u} \cdot \boldsymbol{a}|}{||\boldsymbol{a}||} \tag{23.43}$$

If $\theta$ denotes the angle between $\boldsymbol{u}$ and $\boldsymbol{a}$, then $\boldsymbol{u} \cdot \boldsymbol{a} = ||\boldsymbol{u}||||\boldsymbol{a}|| \cos\ \theta$, so the equation above can also be written as

$$||proj_{\boldsymbol{a}}\boldsymbol{u}|| = ||\boldsymbol{u}||\ |\cos\ \theta| \tag{23.44}$$

**Theorem 23.30: Theorem of Pythagoras in $\mathbb{R}^n$**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal vectors in $\mathbb{R}^n$ with the Euclidean inner product, then

$$||\boldsymbol{u} + \boldsymbol{v}||^2 = ||\boldsymbol{u}||^2 + ||\boldsymbol{v}||^2 \tag{23.45}$$

**Theorem 23.31: Distance in $\mathbb{R}^n$**

In $\mathbb{R}^2$ the distance $D$ between the point $P_0(x_0, y_0)$ and the line $ax + by + c = 0$ is

$$D = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{23.46}$$

In $\mathbb{R}^3$ the distance $D$ between the point $P_0(x_0, y_0, z_0)$ and the plane $ax + by + cz + d = 0$ is

$$D = \frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}} \tag{23.47}$$

[DF*] There are other useful ways of specifying lines and planes.

A unique line in $\mathbb{R}^2$ or $\mathbb{R}^3$ is determined by a point $\boldsymbol{x}_0$ on the line and a nonzero vector $\boldsymbol{v}$ parallel to the line.

A unique plane in $\mathbb{R}^3$ is determined by a point $\boldsymbol{x}_0$ in the plane and two collinear vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ parallel to the plane. The best way to visualize this is to translate the vectors so their initial points are at $\boldsymbol{x}_0$.

**Theorem 23.32: Vector and Parametric Equations of Lines in $\mathbb{R}^2$ and $\mathbb{R}^3$**

Let $L$ be the line in $\mathbb{R}^2$ or $\mathbb{R}^3$ that contains the point $\boldsymbol{x}_0$ and is parallel to the nonzero vector $\boldsymbol{v}$. Then the equation of the line through $\boldsymbol{x}_0$ that is parallel to $\boldsymbol{v}$ is

$$\boldsymbol{x} = \boldsymbol{x}_0 + t\boldsymbol{v} \tag{23.48}$$

If $\boldsymbol{x}_0 = \boldsymbol{0}$, then the line passes through the origin and the equation has the form

$$\boldsymbol{x} = t\boldsymbol{v} \tag{23.49}$$

the variable $t$ is called a parameter, and it varies from $-\infty$ to $\infty$.

**Theorem 23.33: Vector and Parametric Equations of Planes in and $\mathbb{R}^3$**

Let $W$ be the plane in $\mathbb{R}^3$ that contains the point $\boldsymbol{x}_0$ and is parallel to the noncollinear vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Then an equation of the plane through $\boldsymbol{x}_0$ that is parallel to $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is given by

$$\boldsymbol{x} = \boldsymbol{x}_0 + t_1\boldsymbol{v}_1 + t_2\boldsymbol{v}_2 \tag{23.50}$$

If $\boldsymbol{x}_0 = \boldsymbol{0}$, then the plane passes through the origin and the equation has the form

$$\boldsymbol{x} = t_1\boldsymbol{v}_1 + t_2\boldsymbol{v}_2 \tag{23.51}$$

**Definition 23.22: Parametric Equations of Lines in $\mathbb{R}^n$**

If $\boldsymbol{x}_0$ and $\boldsymbol{v}$ are vectors in $\mathbb{R}^n$, and if $\boldsymbol{v}$ is nonzero, then the equation

$$\boldsymbol{x} = \boldsymbol{x}_0 + t\boldsymbol{v} \tag{23.52}$$

defines the line through $\boldsymbol{x}_0$ that is parallel to $\boldsymbol{v}$. In the special case where $\boldsymbol{x}_0 = \boldsymbol{0}$, the line is said to pass through the origin.

**Definition 23.23: Parametric Equations of Planes in $\mathbb{R}^n$**

If $\boldsymbol{x}_0$, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are vectors in $\mathbb{R}^n$, and if $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are not collinear, then the equation

$$\boldsymbol{x} = \boldsymbol{x}_0 + t_1\boldsymbol{v}_1 + t_2\boldsymbol{v}_2 \tag{23.53}$$

defines the plane through $\boldsymbol{x}_0$ that is parallel to $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. In the special case where $\boldsymbol{x}_0 = \boldsymbol{0}$, the plane is said to pass through the origin.

[DF*] If $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ are distinct points in $\mathbb{R}^n$, then the line determined by these points is parallel to the vector $\boldsymbol{v} = \boldsymbol{x}_1 - \boldsymbol{x}_0$, so the line can be expressed in vector form as

$$\boldsymbol{x} = \boldsymbol{x}_0 + t(\boldsymbol{x}_1 - \boldsymbol{x}_0) \tag{23.54}$$

this is called the two-point vector equations of a line in $\mathbb{R}^n$.

**Definition 23.24: Line Segment from $x_0$ to $x_1$**

If $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ are vectors in $\mathbb{R}^n$, then the equation

$$\boldsymbol{x} = \boldsymbol{x}_0 + t(\boldsymbol{x}_1 - \boldsymbol{x}_0) \quad (0 \leq t \leq 1) \tag{23.55}$$

defines the line segment from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$. When convenient, the equation above can be written as

$$\boldsymbol{x} = (1-t)\boldsymbol{x}_0 + t\boldsymbol{x}_1 \quad (0 \leq t \leq 1) \tag{23.56}$$

[DF*] Recall that a linear equation in the variables $x_1, x_2, \ldots, x_n$ has the form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

with $a_1, a_2, \ldots, a_n$ not all zero.

The corresponding homogeneous equation is

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = 0$$

with $a_1, a_2, \ldots, a_n$ not all zero.

These equations can be rewritten in vector form by letting

$$\begin{aligned} \boldsymbol{a} &= (a_1, a_2, \ldots, a_n) \\ \boldsymbol{x} &= (x_1, x_2, \ldots, x_n) \end{aligned}$$

in dot product it can be written as

$$\boldsymbol{a} \cdot \boldsymbol{x} = b \tag{23.57}$$

and for the homogeneous equation

$$\boldsymbol{a} \cdot \boldsymbol{x} = 0 \tag{23.58}$$

this equation reveals that each solution vector $\boldsymbol{x}$ of a homogeneous equation is orthogonal to the coefficient vector $\boldsymbol{a}$.

[DF*] Consider the homogeneous system

$$\begin{array}{lllll} a_{11}x_1 & + a_{12}x_2 & + \ldots & + a_{1n}x_n & = 0 \\ a_{21}x_1 & + a_{22}x_2 & + \ldots & + a_{2n}x_n & = 0 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1}x_1 & + a_{m2}x_2 & + \ldots & + a_{mn}x_n & = 0 \end{array}$$

if we denote the successive row vectors of the coefficient matrix by $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_m$, then we can rewrite this system in dot product form as

$$\begin{array}{ll} \boldsymbol{r}_1 \cdot \boldsymbol{x} & = \mathbf{0} \\ \boldsymbol{r}_2 \cdot \boldsymbol{x} & = \mathbf{0} \\ \vdots & \vdots \\ \boldsymbol{r}_m \cdot \boldsymbol{x} & = \mathbf{0} \end{array}$$

from which we see that every solution vector $\boldsymbol{x}$ is orthogonal to every row vector of the coefficient matrix.

**Theorem 23.34: Solution of Homogeneous Linear System**

If $A$ is an $m \times n$ matrix, then the solution set of the homogeneous linear system $A\boldsymbol{x} = \mathbf{0}$ consists of all vectors in $\mathbb{R}^n$ that are orthogonal to every row vector of $A$.

[DF*] Homogeneous linear system $A\boldsymbol{x} = \mathbf{0}$ and nonhomogeneous linear system $A\boldsymbol{x} = \boldsymbol{b}$ that has the same coefficient matrix are called corresponding linear systems.

**Theorem 23.35: General and Specific Solution of Linear System**

The general solution of a consistent linear system $A\boldsymbol{x} = \boldsymbol{b}$ can be obtained by adding any specific solution of $A\boldsymbol{x} = \boldsymbol{b}$ to the general solution of $A\boldsymbol{x} = \mathbf{0}$.

[DF*] The entire solution set of $A\boldsymbol{x} = \boldsymbol{b}$ can be obtained by translating the solution set of $A\boldsymbol{x} = \mathbf{0}$ by the vector $\boldsymbol{x}_0$.

[DF*] When you have matrix of size $m \times n$, you just cannot apply Gaussian elimination directly to an $m \times n$ matrix problem. If you have more equations than unknowns ($m > n$), the your problem is overdetermined and you have no solution, which means you need to use something like the least squares method. If you have $A\boldsymbol{x} = \boldsymbol{b}$, then instead of having $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ (when $n = m$), then you have to do

$$\boldsymbol{x} = (A^T A)^{-1} A^T \boldsymbol{b}$$

In the case where you have less equations then unknowns ($m < n$), then your problem is underdetermined and you have an infinity of solutions, like the general solution of homogeneous linear system. In that case, you either pick one at random (e.g. setting some of the unknowns to an arbitrary value as parameter), or you need to use regularization, which means trying to add some extra constraints.

[DF*] If you want to solve systems of equations, LU decomposition, QR decomposition (stabler than LU, but slower), Cholesky decomposition (in the case the system is symmetric) or SVD (in the case the system is not square) are almost always better choices.

**Definition 23.25: Cross Product**

If $\boldsymbol{u} = (u_1, u_2, u_3)$ and $\boldsymbol{v} = (v_1, v_2, v_3)$ are vectors in 3-space, then the cross product $\boldsymbol{u} \times \boldsymbol{v}$ is the vector defined by

$$\boldsymbol{u} \times \boldsymbol{v} = (u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2 - u_2 v_1) \tag{23.59}$$

or, in determinant notation,

$$\boldsymbol{u} \times \boldsymbol{v} = \left( \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix}, - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix}, \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \right) \tag{23.60}$$

**Theorem 23.36: Relationship Involving Cross Product and Dot Product**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in 3-space, then

(a) $\boldsymbol{u} \cdot (\boldsymbol{u} \times \boldsymbol{v}) = 0$

(b) $\boldsymbol{v} \cdot (\boldsymbol{u} \times \boldsymbol{v}) = 0$

(c) $||\boldsymbol{u} \times \boldsymbol{v}||^2 = ||\boldsymbol{u}||^2 ||\boldsymbol{v}||^2 - (\boldsymbol{u} \cdot \boldsymbol{v})^2$

(d) $\boldsymbol{u} \times (\boldsymbol{v} \times \boldsymbol{w}) = (\boldsymbol{u} \cdot \boldsymbol{w})\boldsymbol{v} - (\boldsymbol{u} \cdot \boldsymbol{v})\boldsymbol{w}$

(e) $(\boldsymbol{u} \times \boldsymbol{v}) \times \boldsymbol{w} = (\boldsymbol{u} \cdot \boldsymbol{w})\boldsymbol{v} - (\boldsymbol{v} \cdot \boldsymbol{w})\boldsymbol{u}$

**Theorem 23.37: Properties of Cross Product**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in 3-space and $k$ is any scalar, then

(a) $\boldsymbol{u} \times \boldsymbol{v} = -(\boldsymbol{v} \times \boldsymbol{u})$

(b) $\boldsymbol{u} \times (\boldsymbol{v} + \boldsymbol{w}) = (\boldsymbol{u} \times \boldsymbol{v}) + (\boldsymbol{u} \times \boldsymbol{w})$

(c) $(\boldsymbol{u} + \boldsymbol{v}) \times \boldsymbol{w} = (\boldsymbol{u} \times \boldsymbol{w}) + (\boldsymbol{v} \times \boldsymbol{w})$

(d) $k(\boldsymbol{u} \times \boldsymbol{v}) = (k\boldsymbol{u}) \times \boldsymbol{v} = \boldsymbol{u} \times (k\boldsymbol{v})$

(e) $(\boldsymbol{u} \times \boldsymbol{0}) = \boldsymbol{0} \times \boldsymbol{u} = \boldsymbol{0}$

(f) $\boldsymbol{u} \times \boldsymbol{u} = 0$

[DF*] Consider the vectors

$$\boldsymbol{i} = (1,0,0), \quad \boldsymbol{j} = (0,1,0), \quad \boldsymbol{k} = (0,0,1)$$

these vectors each have length 1 and lie along the coordinate axes. They are called the standard unit vectors in 3-space. Every vector $\boldsymbol{v} = (v_1, v_2, v_3)$ in 3-space is expressible in terms of $\boldsymbol{i}, \boldsymbol{j}$, and $\boldsymbol{k}$ since we can write

$$\begin{aligned} \boldsymbol{v} &= (v_1, v_2, v_3) \\ &= v_1(1,0,0) + v_2(0,1,0) + v_3(0,0,1) \\ &= v_1\boldsymbol{i} + v_2\boldsymbol{j} + v_3\boldsymbol{k} \end{aligned}$$

for example

$$(1,8,2) = \boldsymbol{i} + 8\boldsymbol{j} + 2\boldsymbol{k}$$

The cross product for standard unit vectors

$$\begin{array}{ccc} \boldsymbol{i} \times \boldsymbol{i} = \boldsymbol{0} & \boldsymbol{j} \times \boldsymbol{j} = \boldsymbol{0} & \boldsymbol{k} \times \boldsymbol{k} = \boldsymbol{0} \\ \boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k} & \boldsymbol{j} \times \boldsymbol{k} = \boldsymbol{i} & \boldsymbol{k} \times \boldsymbol{i} = \boldsymbol{j} \\ \boldsymbol{j} \times \boldsymbol{i} = -\boldsymbol{k} & \boldsymbol{k} \times \boldsymbol{j} = -\boldsymbol{i} & \boldsymbol{i} \times \boldsymbol{k} = -\boldsymbol{j} \end{array}$$

[DF*] The cross product can be represented symbolically in the form

$$\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix} \boldsymbol{i} - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix} \boldsymbol{j} - \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \boldsymbol{k} \tag{23.61}$$

It is not true in general that

$$\boldsymbol{u} \times (\boldsymbol{v} \times \boldsymbol{w}) = (\boldsymbol{u} \times \boldsymbol{v}) \times \boldsymbol{w}$$

for example

$$\boldsymbol{i} \times (\boldsymbol{j} \times \boldsymbol{j}) \neq (\boldsymbol{i} \times \boldsymbol{j}) \times \boldsymbol{j}$$

in cross product order matters.

[DF*] Let $\theta$ be the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$, imagine your four fingers is pointing at $\boldsymbol{u}$ with thumb orthogonal to the four fingers, and then sweep your four fingers toward the direction of $\boldsymbol{v}$, the thumb will be facing the direction of $\boldsymbol{u} \times \boldsymbol{v}$.

[DF*] If $\theta$ denotes the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$, then

$$\boldsymbol{u} \cdot \boldsymbol{v} = ||\boldsymbol{u}||||\boldsymbol{v}|| \cos\ \theta$$

so

$$\begin{aligned} ||\boldsymbol{u} \times \boldsymbol{v}||^2 &= ||\boldsymbol{u}||^2||\boldsymbol{v}||^2 - ||\boldsymbol{u}||^2||\boldsymbol{v}||^2\ \cos^2\theta \\ &= ||\boldsymbol{u}||^2||\boldsymbol{v}||^2(1-\cos^2\theta) \\ &= ||\boldsymbol{u}||^2||\boldsymbol{v}||^2\ \sin^2\theta \end{aligned}$$

since $0 \leq \theta \leq \pi$, it follows that $\sin\ \theta \geq 0$, so this can be rewritten as

$$||\boldsymbol{u} \times \boldsymbol{v}|| = ||\boldsymbol{u}||||\boldsymbol{v}||\ \sin\ \theta \tag{23.62}$$

**Figure 23.16:** $||v||\sin\ \theta$ *is the altitude of the parallelogram determined by* $\boldsymbol{u}$ *and* $\boldsymbol{v}$. *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Cross Product-2D Plot Parallelogram/main.cpp**).*

**Theorem 23.38: Area of a Parallelogram**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ in 3-space, then $||\boldsymbol{u} \times \boldsymbol{v}||$ is equal to the area of the parallelogram determined by $\boldsymbol{u}$ and $\boldsymbol{v}$.

**Definition 23.26: Scalar Triple Product**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in 3-space, then

$$\boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{w}) = \begin{vmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} \tag{23.63}$$

is called the scalar triple product of $\boldsymbol{u} = (u_1, u_2, u_3), \boldsymbol{v} = (v_1, v_2, v_3)$, and $\boldsymbol{w} = (w_1, w_2, w_3)$.

It follows that

$$\boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{w}) = \boldsymbol{w} \cdot (\boldsymbol{u} \times \boldsymbol{v}) = \boldsymbol{v} \cdot (\boldsymbol{w} \times \boldsymbol{u})$$

[DF*] **Example:**
Simplify $(\boldsymbol{u} + \boldsymbol{v}) \times (\boldsymbol{u} - \boldsymbol{v})$
**Solution:**

$$\begin{aligned} (\boldsymbol{u} + \boldsymbol{v}) \times (\boldsymbol{u} - \boldsymbol{v}) &= (\boldsymbol{u} + \boldsymbol{v}) \times \boldsymbol{u} - (\boldsymbol{u} + \boldsymbol{v}) \times \boldsymbol{v} \\ &= (\boldsymbol{u} \times \boldsymbol{u}) + (\boldsymbol{v} \times \boldsymbol{u}) - ((\boldsymbol{u} \times \boldsymbol{v}) + (\boldsymbol{v} \times \boldsymbol{v})) \\ &= 0 + (\boldsymbol{v} \times \boldsymbol{u}) - (\boldsymbol{u} \times \boldsymbol{v}) - 0 \\ &= (\boldsymbol{v} \times \boldsymbol{u}) + (\boldsymbol{v} \times \boldsymbol{u}) \\ &= 2(\boldsymbol{v} \times \boldsymbol{u}) \end{aligned}$$

[DF*] **Example:**
Prove: If $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$, and $\boldsymbol{d}$ lie in the same plane, then

$$(\boldsymbol{a} \times \boldsymbol{b}) \times (\boldsymbol{c} \times \boldsymbol{d}) = \mathbf{0}$$

**Solution:**
We know that $\boldsymbol{a} \times \boldsymbol{b}$ will be perpendicular to $\boldsymbol{a}$ and $\boldsymbol{b}$, so does $\boldsymbol{c} \times \boldsymbol{d}$ will be perpendicular to $\boldsymbol{c}$ and $\boldsymbol{d}$. When all the vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$, and $\boldsymbol{d}$ lie in the same plane it will make $\boldsymbol{a} \times \boldsymbol{b}$ and $\boldsymbol{c} \times \boldsymbol{d}$ parallel to each other.

The cross product of two parallel vectors will be $\mathbf{0}$.

**Theorem 23.39: Geometric Interpretation of Determinants**

(a) The absolute value of the determinant

$$\det \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}$$

is equal to the area of the parallelogram in 2-space determined by the vectors $\boldsymbol{u} = (u_1, u_2)$ and $\boldsymbol{v} = (v_1, v_2)$.

(b) The absolute value of the determinant

$$\det \begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix}$$

is equal to the volume of the parallelepiped in 3-dimension determined by the vectors $\boldsymbol{u} = (u_1, u_2, u_3), \boldsymbol{v} = (v_1, v_2, v_3)$, and $\boldsymbol{w} = (w_1, w_2, w_3)$.

## i. C++ Plot: Vector Addition in $\mathbb{R}^2$

We have learned that we can do some algebraic operations on vectors, we will learn how to plot vector addition in 2 dimension with Gnuplot from C++.

```
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;

        float o = 0;
        float cx = 1;
        float cy = 1;
        float bx = 1;
        float by = 0;
        // Create a vector with origin at (0,0) and terminal point at (1,0)
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_dx.push_back(bx);
        pts_B_dy.push_back(by);
        // Create a vector with origin at (0,0) and terminal point at (1,1)
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_dx.push_back(cx);
        pts_C_dy.push_back(cy);
        // Create a vector from addition of B and C
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(cx+bx);
        pts_D_dy.push_back(cy+by);
```

```
        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1:3]\nset yrange [-1:2]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "plot '-' with vectors title 'v', '-' with vectors title 'w',
             '-' with vectors title 'v+w'\n";
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
    }
```

**C++ Code 112:** *main.cpp "2D Vector Addition"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

Explanation for the codes:

- Declare the origin for each vector as floating number then we put them into the origin coordinate and terminal coordinate for each vector, as Gnuplot has the system to plot vector from $(x, y)$ to $(x + dx, y + dy)$.

```
        float o = 0;
        float cx = 1;
        float cy = 1;
        float bx = 1;
        float by = 0;
        // Create a vector with origin at (0,0) and terminal point at
            (1,0)
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_dx.push_back(bx);
        pts_B_dy.push_back(by);
        // Create a vector with origin at (0,0) and terminal point at
            (1,1)
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_dx.push_back(cx);
        pts_C_dy.push_back(cy);
        // Create a vector from addition of B and C
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(cx+bx);
        pts_D_dy.push_back(cy+by);
```

- The plotting here is being done by Gnuplot we call it as **gp**, since it is already declared as that in the first line inside **int main()**. Boost library help to coordinating the tuple for each vector from its initial point to the terminal point.

```
gp << "set xrange [-1:3]\nset yrange [-1:2]\n";

gp << "plot '-' with vectors title 'v', '-' with vectors title
     'w', '-' with vectors title 'v+w'\n";
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx,
    pts_B_dy));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx,
    pts_C_dy));
gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx,
    pts_D_dy));
```

**Figure 23.17:** *The plot of a vector $\boldsymbol{v}$, $\boldsymbol{w}$, and its addition $\boldsymbol{v}+\boldsymbol{w}$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Vector Addition/main.cpp**).*

## ii. C++ Plot: Vector Addition in $\mathbb{R}^3$

After we plot vectors addition in 2 dimension with Gnuplot from C++, now it is time to plot vectors addition in 3 dimension.

```
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;

        float o = 0;
        float cx = 1;
        float cy = 1;
        float cz = 5;
        float bx = 1;
        float by = 0;
        float bz = 0;
        // Create a vector with origin at (0,0,0) and terminal point at
            (1,0,0)
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_z .push_back(o);
        pts_B_dx.push_back(bx);
        pts_B_dy.push_back(by);
```

```
        pts_B_dz.push_back(bz);
        // Create a vector with origin at (0,0) and terminal point at
            (1,1,5)
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_z .push_back(o);
        pts_C_dx.push_back(cx);
        pts_C_dy.push_back(cy);
        pts_C_dz.push_back(cz);
        // Create a vector from addition of B and C
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_z .push_back(o);
        pts_D_dx.push_back(cx+bx);
        pts_D_dy.push_back(cy+by);
        pts_D_dz.push_back(cz+bz);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1:3]\nset yrange [-1:3]\nset zrange [-1:5]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "splot '-' with vectors title 'v', '-' with vectors title 'w
            ', '-' with vectors title 'v+w'\n";
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
    }
```

**C++ Code 113:** *main.cpp "3D Plot Addition"*

To compile and run it type:
**make**
**./main**

to compile it without Makefile / manually:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

Explanation for the codes:

- In 3 dimension / $\mathbb{R}^3$, you only need to add another coordinate, which is $z$, thus in the plot we use **splot** instead of **plot**.

  For 3D, Gnuplot has the system to plot vector from $(x, y, z)$ to $(x + dx, y + dy, z + dz)$.

```
        gp << "splot '-' with vectors title 'v', '-' with vectors
            title 'w', '-' with vectors title 'v+w'\n";
```

```
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx
                , pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx
                , pts_C_dy, pts_C_dz));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx
                , pts_D_dy, pts_D_dz));
```

**Figure 23.18:** *The plot of a vector $\boldsymbol{v}$, $\boldsymbol{w}$, and its addition $\boldsymbol{v}+\boldsymbol{w}$ in 3 dimension with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Vector Addition/main.cpp**).*

## iii. C++ Computation: Linear Combination

We are going to use exercise number 24 chapter 3.1 from [11].

Let $\boldsymbol{u} = (2, 1, 0, 1, -1)$ and $\boldsymbol{v} = (-2, 3, 1, 0, 2)$. Find scalars $a$ and $b$ so that $a\boldsymbol{u} + b\boldsymbol{v} = (-8, 8, 3, -1, 7)$.

**Solution:**

In matrix term it will be like this

$$\begin{bmatrix} 2 & -2 \\ 1 & 3 \\ 0 & 1 \\ 1 & 0 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} -8 \\ 8 \\ 3 \\ -1 \\ 7 \end{bmatrix}$$

**C++ Code 114:** *main.cpp "Linear Combination"*

Explanation for the codes:

- 

## iv. C++ Computation: Norm, Angle, Dot Product and Distance Between Vectors in $\mathbb{R}^n$

We are going to create C++ code to compute the norm, distance, dot product and the angle between two vectors $\boldsymbol{v}$ and $\boldsymbol{x}$, with

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 8 \\ 8 \\ 2 \end{bmatrix}$$

$$\boldsymbol{x} = \begin{bmatrix} 5 \\ 1 \\ 9 \\ 6 \end{bmatrix}$$

The vectors will be stored separately in a textfile called **vector1.txt** and **vector2.txt**, this could come in handy if we retrieve the data from sensor captured in a device or a satellite, so we don't need for manual input in the C++ code anymore.

```
#include <fstream>
#include <vector>
#include <iostream>
#include <cmath>

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f


const int N = 4;

using std::cout;
using std::endl;
using namespace std;

void printArray(int** arr) {

	for (int i = 0; i < N; ++i)
	{
		for (int j = 0; j < N; ++j)
		{
			cout <<arr[i][j] << ' ';
		}
		cout << endl;
	}
}

int* vec() {
	static int x[N];
	std::fstream in("vector1.txt");
	float vectortiles[N];
```

```
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

int* vec2() {
        static int v[N];
        std::fstream in("vector2.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}

int main() {
        std::vector<std::pair<double, double>> xy_pts;
        int* ptrvec;
        int* ptrvec2;
        ptrvec = vec();
        ptrvec2 = vec2();
        cout << endl;
        cout << "Vector x is: " << endl;
        for (int i = 0; i < N; ++i)
        {
                cout <<ptrvec[i]<< ' ' << endl;
        }

        cout << endl;
        cout << "Vector v is: " << endl;
        for (int i = 0; i < N; ++i)
        {
                cout <<ptrvec2[i]<< ' ' << endl;
        }

        cout << endl;

        float dot= 0;
        float distance= 0;
        float normx= 0;
        float normv= 0;

        for (int i = 0; i < N; ++i)
```

```
            {
                dot += ptrvec[i] * ptrvec2[i];
                distance += (ptrvec2[i] - ptrvec[i]) * (ptrvec2[i] - ptrvec[
                    i]);
                normx += ptrvec[i] * ptrvec[i];
                normv += ptrvec2[i] * ptrvec2[i];
            }

            float angleformula = dot/(sqrt(normx) * sqrt(normv));

            cout << endl;
            cout << "x . v = " << dot << endl;
            cout << "d(x, v) = " << sqrt(distance) << endl;
            cout << "|| x || = " << sqrt(normx) << endl;
            cout << "|| v || = " << sqrt(normv) << endl;

            cout << "theta = " << acos(angleformula*DEGTORAD) * RADTODEG <<endl;

            return 0;
        }
```

**C++ Code 115:** *main.cpp "Norm Angle Dot Product and Distance"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Norm, Dot Product, Distance ]# ./main

Vector x is:
1
8
8
2

Vector v is:
5
1
9
6


x . v = 97
d(x, v) = 9.05539
|| x || = 11.5326
|| v || = 11.9583
θ = 89.2966
```

**Figure 23.19:** *The computation of norm, dot product, distance and angle between two vectors with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Norm, Angle, Dot Product, Distance/main.cpp**).*

## v. C++ Plot: Line and Its Normal in $\mathbb{R}^2$

We will plot a line and a nonzero vector called normal that is perpenndicular to that line.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;

        // Point-normal equation of the line:
        // a(x-x0) + b(y-y0) = 0
        // we choose the value for (x0,y0) = (0,0) the origin
        float xo = 0;
        float yo = 0;
```

```
        // normal n = (a,b) = (2,3)
        float a = 2;
        float b = 1.3;

        float bx = -1;
        float by = yo + ( -(a/b)*(bx-xo) ) ;

        // Create a vector with origin at (0,0) and terminal point at (x,y)
            = (bx,by)
        pts_B_x .push_back(xo);
        pts_B_y .push_back(yo);
        pts_B_dx.push_back(bx);
        pts_B_dy.push_back(by);
        // Create a vector with origin at (0,0) and terminal point at (a,b)
            as the normal
        pts_C_x .push_back(xo);
        pts_C_y .push_back(yo);
        pts_C_dx.push_back(a);
        pts_C_dy.push_back(b);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-2:3]\nset yrange [-1:2]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = (-(2/1.3)*x)\n";
        gp << "plot '-' with vectors title 'P(x,y)', '-' with vectors title
             'n', f(x) title 'Line'\n";

        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));

        return 0;
    }
```

**C++ Code 116:** *main.cpp "Line and Its Normal"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- We choose the initial point as the origin $P_0 = (x_0, y_0) = (0, 0)$ and the vector that is passing the origin and the point $P(x, y)$ will be denoted by

$$\vec{P_0P} = (x - x_0, y - y_0)$$

with the normal that is orthogonal to $\vec{P_0P}$ will be denoted by $\boldsymbol{n} = (a, b)$ and satisfy this equation

$$\boldsymbol{n} \cdot \vec{P_0P}$$

we can freely determine or choose the value of $x$ for the point $P$ then compute the value of $y$ from point-normal equation of the line which is

$$a(x - x_0) + b(y - y_0) = 0$$

```
float xo = 0;
float yo = 0;
// normal n = (a,b) = (2,1.3)
float a = 2;
float b = 1.3;

float bx = -1;
float by = yo + ( -(a/b)*(bx-xo) ) ;
```

- We plot the function $f(x)$ as the line that is coincide with the vector $P$ and the normal $\boldsymbol{n}$.

```
gp << "set xrange [-2:3]\nset yrange [-1:2]\n";
gp << "f(x) = (-(2/1.3)*x)\n";
gp << "plot '-' with vectors title 'P(x,y)', '-' with vectors
    title 'n', f(x) title 'Line'\n";

gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx,
    pts_B_dy));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx,
    pts_C_dy));
```

**Figure 23.20:** *The plot of a line and its normal in* $\mathbb{R}^2$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Line and Normal/main.cpp**).*

## vi. C++ Plot: Plane and Its Normal in $\mathbb{R}^3$

This time we will plot a plane, a vector that is on the plane $P(x, y, z)$ and its normal $\boldsymbol{n}$ / a perpendicular vector toward vector $P(x, y, z)$ and the plane in $\mathbb{R}^3$.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;

        // Point-normal equation of the plane:
        // a(x-x0) + b(y-y0) + c(z-z0) = 0
        // we choose the value for (x0,y0,z0) = (0,0,0) the origin
        float x0 = 0;
        float y0 = 0;
        float z0 = 0;
        // normal n = (a,b,c) = (1,1,-2)
        float a = 1;
        float b = 1;
        float c = -2;
        // to determine the vector that is passing through the origin
            (0,0,0) and (x,y,z)
        float bx = 1;
        float by = 1;
        float bz = 1;

        // Create a vector with origin at (0,0,0) and terminal point at (x,y
            ,z) = (bx,by)
        pts_B_x .push_back(x0);
        pts_B_y .push_back(y0);
```

```
        pts_B_z .push_back(z0);
        pts_B_dx.push_back(bx);
        pts_B_dy.push_back(by);
        pts_B_dz.push_back(bz);

        // Create a vector with origin at (0,0,0) and terminal point at (a,b
            ,c) as the normal
        pts_C_x .push_back(x0);
        pts_C_y .push_back(y0);
        pts_C_z .push_back(z0);
        pts_C_dx.push_back(a);
        pts_C_dy.push_back(b);
        pts_C_dz.push_back(c);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-2:3]\n set yrange [-1:2]\n set zrange [-2:5]\n
            ";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 60,5,1\n"; // pitch,yaw,zoom
        gp << "g(x,y) = x \n";
        gp << "h(x,y) = y \n";
        // h(x,y) title 'z=y', g(x,y) title 'z = x'
        gp << "splot '-' with vectors title 'P(x,y,z)', '-' with vectors
            title 'n', \
        '+' using 1:(g($1,$1)):(h($1,$1)) title 'plane' with filledcurves lc
             rgb 'blue'\n";

        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));

        return 0;
    }
```

**C++ Code 117:** *main.cpp "Plane and Its Normal"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- We first determine the normal will intersect with the vector $P(x, y, z) = (bx, by, bz) = (1, 1, 1)$ at the origin $(x_0, y_0, z_0) = (0, 0, 0)$, thus based on the point-normal equation of the plane

which is

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

we can determine the normal $\boldsymbol{n} = (a, b, c)$ that is orthogonal to the vector $P(x, y, z)$ and the plane.

```
float x0 = 0;
float y0 = 0;
float z0 = 0;
// normal n = (a,b,c) = (1,1,-2)
float a = 1;
float b = 1;
float c = -2;

float bx = 1;
float by = 1;
float bz = 1;
```

- We set the view from pitch angle of 60 and yaw angle of 5. We plot the plane in 3D by using two functions of two variables $z = g(x, y) = x$ and $z = h(x, y) = y$ and use **filledcurves** to fill the area between the two functions so it will become a plane of two triangles. The rest is to plot the vector $P(x, y, z)$ and the normal $\boldsymbol{n}$.

```
gp << "set xrange [-2:3]\n set yrange [-1:2]\n set zrange
    [-2:5]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel
    'z-axis'\n";
gp << "set view 60,5,1\n"; // pitch,yaw,zoom
gp << "g(x,y) = x \n";
gp << "h(x,y) = y \n";

gp << "splot '-' with vectors title 'P(x,y,z)', '-' with
    vectors title 'n', \
'+' using 1:(g($1,$1)):(h($1,$1)) title 'plane' with
    filledcurves lc rgb 'blue'\n";

gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx
    , pts_B_dy, pts_B_dz));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx
    , pts_C_dy, pts_C_dz));
```

**Figure 23.21:** *The plot of a line and its normal in* $\mathbb{R}^2$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Plane and Normal/main.cpp**).*

## vii. C++ Plot: Distance Between a Point and the Plane in $\mathbb{R}^3$

To find the distance between a point and a plane in $\mathbb{R}^3$, we are going to use the point

$$(1, 1, 5)$$

and the plane equation is

$$z = 1$$

from $ax + by + cz + d = 0$ with $a = 0,\ b = 0,\ c = 1$, and $d = -1$. The plane of $z = 1$ is a flat plane that has the same height of $z = 1$ for all values of $x$ and $y$.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

using std::cout;
using std::endl;
using namespace std;

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;

        // Point-normal equation of the plane:
        // a(x-x0) + b(y-y0) + c(z-z0) = 0
        // we choose the value for (x0,y0,z0) = (0,0,1) the origin
        float x0 = 0;
        float y0 = 0;
        float z0 = 1;
        // normal n
        float nx = 0;
        float ny = 0;
        float nz = 5;
```

```
// to determine the vector that is passing through the origin
    (0,0,1) and the point (x,y,z)
float bx = 1;
float by = 1;
float bz = 5;

// Create a vector with origin at (0,0,1) and terminal point at
    (1,1,6)
pts_B_x .push_back(x0);
pts_B_y .push_back(y0);
pts_B_z .push_back(z0);
pts_B_dx.push_back(bx);
pts_B_dy.push_back(by);
pts_B_dz.push_back(bz);

// Create a vector with origin at (0,0,1) and terminal point at
    (0,0,6) as the normal
pts_C_x .push_back(x0);
pts_C_y .push_back(y0);
pts_C_z .push_back(z0);
pts_C_dx.push_back(nx);
pts_C_dy.push_back(ny);
pts_C_dz.push_back(nz);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-2:2]\n set yrange [-2:2]\n set zrange [-3:6]\n
    ";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
gp << "set view 100,5,1\n"; // pitch,yaw,zoom
gp << "f(x,y) = 1\n";
gp << "array point[1]\n"; // using a dummy array of length one

// pt = point type, ps = point size,
gp << "splot '-' with vectors title 'P(x,y,z)', point us (1):(1)
    :(6) title '(x,y,z,)' ps 2 lc 2 pt 6, \
'-' with vectors title 'n', f(x,y) with lines title 'f(x,y)'\n";

gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
    pts_B_dy, pts_B_dz));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
    pts_C_dy, pts_C_dz));

// to compute the distance, the plane equation is ax + by + cz + d =
     0
float a = 0;
float b = 0;
float c = 1;
```

```
            float d = -1;
            float D;
            D = abs(a*(bx+x0) + b*(by+y0) + c*(bz+z0) + d ) / (a*a + b*b + c*c);
            cout << endl;
            cout << "The distance between point ( " << bx+x0 << ", " << by+y0 <<
                 ", " << bz + z0 << ")" << endl;
            cout << "and the plane is : " << D << endl;

            return 0;
        }
```

**C++ Code 118:** *main.cpp "Distance Between a Point and the Plane in 3D"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- To plot a point in 3D with gnuplot' **splot**, we use **array point[1]**, a dummy array of length one and then plot it with command **point us (1):(1):(6)** to plot a point at $(1, 1, 6)$.

```
            gp << "set xrange [-2:2]\n set yrange [-2:2]\n set zrange
                [-3:6]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel
                'z-axis'\n";
            gp << "set view 100,5,1\n"; // pitch,yaw,zoom
            gp << "f(x,y) = 1\n";
            gp << "array point[1]\n"; // using a dummy array of length one

            // pt = point type, ps = point size,
            gp << "splot '-' with vectors title 'P(x,y,z)', point us (1)
                :(1):(6) title '(x,y,z,)' ps 2 lc 2 pt 6, \
            '-' with vectors title 'n', f(x,y) with lines title 'f(x,y)'\n
                ";

            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx
                , pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx
                , pts_C_dy, pts_C_dz));
```

- To compute the distance manually we need to input the parameters $a, b, c, d$ from the plane equation, and the point is $(bx + x0, by + y0, bz + z0) = (1, 1, 6)$, that we want to compute the distance between that point and the plane

```
            float a = 0;
            float b = 0;
```

```
float c = 1;
float d = -1;
float D;
D = abs(a*(bx+x0) + b*(by+y0) + c*(bz+z0) + d ) / (a*a + b*b +
    c*c);
cout << endl;
cout << "The distance between point ( " << bx+x0 << ", " << by+
    y0 << ", " << bz + z0 << ")" << endl;
cout << "and the plane is : " << D << endl;
```

**Figure 23.22:** *The plot of a point and the plane in $\mathbb{R}^3$ and then compute the distance between them with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Distance Between Plane and a Point/main.cpp**).*

## viii. C++ Plot: Direction Cosines in $\mathbb{R}^3$

The exercise is taken from chapter 3.3 number 41 of [11].

Let $\boldsymbol{i}, \boldsymbol{j}$, and $\boldsymbol{k}$ be unit vectors along the positive $x, y$, and $z$ axes of a rectangular coordinate system in 3-space. If $\boldsymbol{v} = (a, b, c)$ is a nonzero vector, then the angles $\alpha, \beta$, and $\gamma$ between $\boldsymbol{v}$ and the vectors $\boldsymbol{i}, \boldsymbol{j}$, and $\boldsymbol{k}$, respectively, are called the direction angles of $\boldsymbol{v}$, and the numbers $\cos\ \alpha, \cos\ \beta, \cos\ \gamma$ are called the direction cosines of $\boldsymbol{v}$.

(a) Show that $\cos\ \alpha = \frac{a}{||\boldsymbol{v}||}$

(b) Find $\cos\ \beta$ and $\cos\ \gamma$

(c) Show that $\frac{\boldsymbol{v}}{||\boldsymbol{v}||} = (\cos\ \alpha, \cos\ \beta, \cos\ \gamma)$

(d) Show that $\cos^2 \alpha + \cos^2 \beta + \cos^2 \gamma = 1$

**Solution:**

(a) $\alpha$ is the angle between $\boldsymbol{v}$ and $\boldsymbol{i}$, so

$$\begin{aligned}
\cos\ \alpha &= \frac{\boldsymbol{v} \cdot \boldsymbol{i}}{||\boldsymbol{v}||||\boldsymbol{i}||} \\
&= \frac{(a)(1) + (b)(0) + (c)(0)}{||\boldsymbol{v}|| \cdot 1} \\
&= \frac{a}{||\boldsymbol{v}||}
\end{aligned}$$

(b) Similar to part (a)

$$\begin{aligned}
\cos\ \beta &= \frac{\boldsymbol{v} \cdot \boldsymbol{j}}{||\boldsymbol{v}||||\boldsymbol{j}||} \\
&= \frac{(a)(0) + (b)(1) + (c)(0)}{||\boldsymbol{v}|| \cdot 1} \\
&= \frac{b}{||\boldsymbol{v}||}
\end{aligned}$$

$$\begin{aligned}
\cos\ \gamma &= \frac{\boldsymbol{v} \cdot \boldsymbol{k}}{||\boldsymbol{v}||||\boldsymbol{i}||} \\
&= \frac{(a)(0) + (b)(0) + (c)(1)}{||\boldsymbol{v}|| \cdot 1} \\
&= \frac{c}{||\boldsymbol{v}||}
\end{aligned}$$

(c) From the result (a) and (b)

$$\frac{\boldsymbol{v}}{||\boldsymbol{v}||} = \left( \frac{a}{||\boldsymbol{v}||}, \frac{b}{||\boldsymbol{v}||}, \frac{c}{||\boldsymbol{v}||} \right) = (\cos\ \alpha, \cos\ \beta, \cos\ \gamma)$$

(d) Since

$$\frac{\boldsymbol{v}}{||\boldsymbol{v}||}$$

is a unit vector, its magnitude is 1, so

$$\sqrt{\cos^2\alpha + \cos^2\beta + \cos^2\gamma} = 1$$
$$\cos^2\alpha + \cos^2\beta + \cos^2\gamma = 1$$

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_z;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_E_dz;

        // we choose the value for (x0,y0,z0) = (0,0,0) the origin
        float x0 = 0;
        float y0 = 0;
        float z0 = 0;
        // unit vector i
        float ix = 1;
```

```
            float iy = 0;
            float iz = 0;
            // unit vector j
            float jx = 0;
            float jy = 1;
            float jz = 0;
            // unit vector k
            float kx = 0;
            float ky = 0;
            float kz = 1;
            // vector v
            float vx = 0.5;
            float vy = 0.6;
            float vz = 1;

            // Create unit vector i
            pts_B_x .push_back(x0);
            pts_B_y .push_back(y0);
            pts_B_z .push_back(z0);
            pts_B_dx.push_back(ix);
            pts_B_dy.push_back(iy);
            pts_B_dz.push_back(iz);

            // Create unit vector j
            pts_C_x .push_back(x0);
            pts_C_y .push_back(y0);
            pts_C_z .push_back(z0);
            pts_C_dx.push_back(jx);
            pts_C_dy.push_back(jy);
            pts_C_dz.push_back(jz);

            // Create unit vector k
            pts_D_x .push_back(x0);
            pts_D_y .push_back(y0);
            pts_D_z .push_back(z0);
            pts_D_dx.push_back(kx);
            pts_D_dy.push_back(ky);
            pts_D_dz.push_back(kz);

            // Create vector v
            pts_E_x .push_back(x0);
            pts_E_y .push_back(y0);
            pts_E_z .push_back(z0);
            pts_E_dx.push_back(vx);
            pts_E_dy.push_back(vy);
            pts_E_dz.push_back(vz);

            // Don't forget to put "\n" at the end of each line!
```

```
        gp << "set xrange [0:1.5]\n set yrange [0:1.5]\n set zrange [0:1.5]\
            n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 80,70,1\n"; // pitch,yaw,zoom
        gp << "splot '-' with vectors title 'i', '-' with vectors title 'j'
            , '-' with vectors title 'k', '-' with vectors title 'v' \n";

        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
            pts_E_dy, pts_E_dz));

        return 0;
    }
```

**C++ Code 119:** *main.cpp "Direction Cosines in 3D"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

**Figure 23.23:** *The plot of unit vectors* $\boldsymbol{i}, \boldsymbol{j}$, *and* $\boldsymbol{k}$ *and* $\boldsymbol{v} = (a, b, c)$ *in* $\mathbb{R}^3$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Unit Vectors/-main.cpp**).*

## ix. C++ Plot: Parametric Equation of Plane in $\mathbb{R}^3$

We have plot a plane with implicit equation such as

$$x - y + 2z - 5 = 0$$

now we will learn to plot a plane in 3 dimension with parametric equation. An example of 2 dimensional parametric function is

$$(x, y) = (\sin(t), \cos(t))$$

which will draws a circle given only one input of independent variable, which is $t$.

For 3 dimensional parametric function, the example is

$$(x, y, z) = (r\ \cos(v)\cos(u), r\ \cos(v)\sin(u), r\ \sin(u))$$

which will draw a sphere of radius $r$, with input of two independent variables which are $u$ and $v$.

Now, consider a plane with the implicit equation above

$$x - y + 2z - 5 = 0$$

we will find the parametric equations for that. We can do this by solving the equation for any one of the variables in terms of the other two and then using those two variables as parameters.

For example, solving for $x$ in terms of $y$ and $z$ yields

$$x = 5 + y - 2z$$

and then using $y$ and $z$ as parameters $t_1$ and $t_2$, respectively, yields the parametric equations

$$x = 5 + t_1 - 2t_2, \quad y = t_1, \quad z = t_2$$

To obtain a vector equation of the plane we rewrite these parametric equations as

$$(x, y, z) = (5 + t_1 - 2t_2, t_1, t_2)$$

or, equivalently, as

$$(x, y, z) = (5, 0, 0) + t_1(1, 1, 0) + t_2(-2, 0, 1)$$

```
#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set parametric\n";
        //gp << "set xrange[-10:20]\n set yrange[-6:6]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 100,5,1\n"; // pitch,yaw,zoom
```

```
        gp << "f(x,y) = (y-x+5)/2\n";
        gp << "splot 5 + u - 2*v, u, v title 'plane'\n";
        //gp << "splot f(x,y) title 'plane',point us (1):(1):(6) title '(x,y
            ,z,)' ps 2 lc 2 pt 6\n";

        return 0;
    }
```

**C++ Code 120:** *main.cpp "Parametric Equation of Plane in 3D"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- Gnuplot plots by default $z$ as a function of $x$ and $y$. When we change into parametric mode with **set parametric**, we now have the variables $u$ and $v$.

  Remember the parametric equation is

  $$(x, y, z) = (5 + t_1 - 2t_2, t_1, t_2)$$

  In gnuplot the first variable is named as $u$ and the second variable is named $v$, thus we will have it like this

  $$(x, y, z) = (5 + u - 2v, u, v)$$

  Then put it into the **splot** command.

```
        gp << "set parametric\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel
            'z-axis'\n";
        gp << "set view 100,5,1\n"; // pitch,yaw,zoom
        gp << "array point[1]\n"; // using a dummy array of length one
        gp << "splot 5 + u - 2*v, u, v title 'plane',point us (1):(1)
            :(6) title '(x,y,z,)' ps 2 lc 2 pt 6\n";
```

- You can comment the **set parametric** and uncomment the two lines below to plot it in normal mode / implicit equation method.

```
        //gp << "set xrange[-10:20]\n set yrange[-6:6]\n";

        ...
        //gp << "splot f(x,y) title 'plane',point us (1):(1):(6) title
            '(x,y,z,)' ps 2 lc 2 pt 6\n";
```

**Figure 23.24:** *The plot of parametric equation of a plane* $(x, y, z) = (5 + u - 2v, u, v)$ *in* $\mathbb{R}^3$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Parametric Equations of Plane/main.cpp**).*

## x. C++ Computation: Numerical Orthogonal Projection

We are going to compute the vectors $w_1$ and $w_2$ that are orthogonal to each other.

Starting with the knowledge of knowing vector $u$ that is dropped / projected into nonzero vector $a$, the vector $w_1$ is a scalar multiple of $a$.

```
#include <fstream>
#include <vector>
#include <iostream>
#include <cmath>

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

const int N = 4;

using std::cout;
using std::endl;
using namespace std;

void printArray(int** arr) {

	for (int i = 0; i < N; ++i)
	{
		for (int j = 0; j < N; ++j)
		{
			cout <<arr[i][j] << ' ';
		}
		cout << endl;
	}
}

int* vec() {
	static int x[N];
	std::fstream in("vector1.txt");
	float vectortiles[N];
	for (int i = 0; i < N; ++i)
	{
		in >> vectortiles[i];
		x[i] =vectortiles[i];
	}
	return x;
}

int* vec2() {
	static int v[N];
	std::fstream in("vector2.txt");
```

```
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}

int main() {
        std::vector<std::pair<double, double>> xy_pts;
        int* ptrvec;
        int* ptrvec2;
        ptrvec = vec();
        ptrvec2 = vec2();
        cout << endl;
        cout << "Vector a is: " << endl;
        for (int i = 0; i < N; ++i)
        {
                cout <<ptrvec[i]<< ' ' << endl;
        }

        cout << endl;
        cout << "Vector u is: " << endl;
        for (int i = 0; i < N; ++i)
        {
                cout <<ptrvec2[i]<< ' ' << endl;
        }

        cout << endl;

        float dot= 0;
        float distance= 0;
        float norma= 0;
        float normu= 0;

        for (int i = 0; i < N; ++i)
        {
                dot += ptrvec[i] * ptrvec2[i];
                distance += (ptrvec2[i] - ptrvec[i]) * (ptrvec2[i] - ptrvec[
                    i]);
                norma += ptrvec[i] * ptrvec[i];
                normu += ptrvec2[i] * ptrvec2[i];
        }

        float angleformula = dot/(sqrt(norma) * sqrt(normu));
        float w1 = dot/(norma*normu);
```

```
        cout << endl;
        cout << "x . v = " << dot << endl;
        cout << "d(x, v) = " << sqrt(distance) << endl;
        cout << "|| x || = " << sqrt(norma) << endl;
        cout << "|| v || = " << sqrt(normu) << endl;

        cout << "theta = " << acos(angleformula*DEGTORAD) * RADTODEG <<endl;
        cout << endl;
        cout << "w1 = proj_{a} u = " << endl;
        for (int i = 0; i < N; ++i)
        {
            cout <<w1*ptrvec[i]<< ' ' << endl;
        }

        cout << endl;
        cout << "w2 = u - proj_{a} u = " << endl;
        for (int i = 0; i < N; ++i)
        {
            cout <<ptrvec2[i] - w1*ptrvec[i]<< ' ' << endl;
        }

        return 0;
    }
```

**C++ Code 121:** *main.cpp "Numerical Orthogonal Projection"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- The code is not too hard to be grasp, for loading vector from textfile we are using **#include <fstream>** and save it as a vector with length of $N$. The textfiles we are using are named **vector1.txt** that will be saved as vector $\vec{x}$ and **vector2.txt** that will be saved as vector $\vec{v}$.

```
int* vec() {
    static int x[N];
    std::fstream in("vector1.txt");
    float vectortiles[N];
    for (int i = 0; i < N; ++i)
    {
        in >> vectortiles[i];
        x[i] =vectortiles[i];
    }
    return x;
```

```
}

int* vec2() {
        static int v[N];
        std::fstream in("vector2.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}
```

- Under the **int main()** we initialize the value for dot product (**dot**), distance (**distance**), norm for vector $x$ (**norma**), norm for vector $v$ (**normu**) as 0 for each, then using sum formula (+=) to get the almost final result, process some of them again with square root or using the equation to obtain the orthogonal projection and angle between the vector and its orthogonal projection, then show them at the end of the code.

  We initialize **angleformula** and **w1** for the angle and orthogonal projection after we obtain (**dot, distance**), $||x||$, and $||v||$.

```
float dot= 0;
float distance= 0;
float norma= 0;
float normu= 0;

for (int i = 0; i < N; ++i)
{
        dot += ptrvec[i] * ptrvec2[i];
        distance += (ptrvec2[i] — ptrvec[i]) * (ptrvec2[i] —
            ptrvec[i]);
        norma += ptrvec[i] * ptrvec[i];
        normu += ptrvec2[i] * ptrvec2[i];
}

float angleformula = dot/(sqrt(norma) * sqrt(normu));
float w1 = dot/(norma*normu);

cout << endl;
cout << "x . v = " << dot << endl;
cout << "d(x, v) = " << sqrt(distance) << endl;
cout << "|| x || = " << sqrt(norma) << endl;
cout << "|| v || = " << sqrt(normu) << endl;

cout << "theta = " << acos(angleformula*DEGTORAD) * RADTODEG <<
    endl;
cout << endl;
```

```
        cout << "w1 = proj_{a} u = " << endl;
        for (int i = 0; i < N; ++i)
        {
            cout <<w1*ptrvec[i]<< ' ' << endl;
        }

        cout << endl;
        cout << "w2 = u - proj_{a} u = " << endl;
        for (int i = 0; i < N; ++i)
        {
            cout <<ptrvec2[i] - w1*ptrvec[i]<< ' ' << endl;
        }
```

```
Vector a is:
1
8
8
2

Vector u is:
5
1
9
6


x . v = 97
d(x, v) = 9.05539
|| x || = 11.5326
|| v || = 11.9583
θ = 89.2966

w1 = proj_{a} u =
0.00510016
0.0408013
0.0408013
0.0102003

w2 = u - proj_{a} u =
4.9949
0.959199
8.9592
5.9898
```

**Figure 23.25:** *The computation to find the vectors* $\boldsymbol{w_1}$ *and* $\boldsymbol{w_2}$*, given* $\boldsymbol{a}$ *and* $\boldsymbol{u}$ *in* $\mathbb{R}^4$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Numerical Orthogonal Projection/main.cpp**).*

## xi. C++ Computation: Symbolic Orthogonal Projection

**C++ Code 122:** *main.cpp "Orthogonal Projection"*

Explanation for the codes:

- 

## xii. C++ Plot and Computation: Cross Product in 3D

Cross product can only be conducted in $\mathbb{R}^3$ or 3 dimensional space, thus when we have two vectors ($\boldsymbol{u}$ and $\boldsymbol{v}$), the cross product of that two vectors will be orthogonal / perpendicular toward that two vectors. As a bonus we also add dot product of $\boldsymbol{u}$ and $\boldsymbol{v}$.

We get the original code from:
**//https://www.geeksforgeeks.org/program-dot-product-cross-product-two-vector/**

We modify and add the computation for area of parallelogram between two vectors and the salar triple product computation too.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>
#include <bits/stdc++.h>

#include "gnuplot-iostream.h"

#define RADTODEG 57.295779513082320876f

using namespace std;

int n = 3;

// Function that return dot product of two vector array.
int dotProduct(float vect_u[], float vect_v[])
{
	float product = 0;
	// Loop for calculate dot product
	for (int i = 0; i < n; i++)
	{
		product = product + vect_u[i] * vect_v[i];
	}
	return product;
}

// Function to find cross product of two vector array.
void crossProduct(float vect_v[], float vect_w[], float cross_P[])
{
	cross_P[0] = vect_v[1] * vect_w[2] - vect_v[2] * vect_w[1];
	cross_P[1] = vect_v[2] * vect_w[0] - vect_v[0] * vect_w[2];
	cross_P[2] = vect_v[0] * vect_w[1] - vect_v[1] * vect_w[0];
}

int main() {
	Gnuplot gp;
```

```
float u1 = 3;
float u2 = -2;
float u3 = -5;

float v1 = 1;
float v2 = 4;
float v3 = -4;

float w1 = 0;
float w2 = 3;
float w3 = 2;

float vect_u[] = { u1, u2, u3 };
float vect_v[] = { v1, v2, v3 };
float vect_w[] = { w1, w2, w3 };
float cross_P[n];
// dotProduct function call
cout << "Vector u:" << setw(15) << "vector v:" << setw(15) << "
    vector w:" << endl;

for (int i = 0; i < n; i++)
{
       cout << vect_u[i] << "\t" << "\t" << vect_v[i] << "\t" << "\t
           " << vect_w[i] ;
       cout << endl;
}
cout<<endl;

cout << "Dot product u . v : ";
cout << dotProduct(vect_u, vect_v) << endl;
cout << "Dot product v . w : ";
cout << dotProduct(vect_v, vect_w) << endl;
cout << "Dot product u . w : ";
cout << dotProduct(vect_u, vect_w) << endl;

// crossProduct function call for vector v and w
crossProduct(vect_v, vect_w, cross_P);
cout << "Cross product v x w: (";

// Loop that print cross product of two vector array.
for (int i = 0; i < n-1; i++)
{
       cout << cross_P[i] << ", ";
}
cout << cross_P[n-1] ; // for better looking display of result
cout << ")" << endl;
```

```
// compute area of parallelogram of u and v
float area_parallelogram = sqrt(cross_P[0]*cross_P[0] + cross_P[1]*
    cross_P[1] + cross_P[2]*cross_P[2]);
cout << "Area of Parallelogram from vector v and w : ";
cout << area_parallelogram << endl;

float angleformula = area_parallelogram / ( sqrt(dotProduct(vect_v,
    vect_v)) * sqrt(dotProduct(vect_w, vect_w)) );
cout << "theta between v and w (degrees) = " << asin(angleformula) *
     RADTODEG <<endl;

// Computing scalar triple product
float scalartripleproduct = 0;
// Loop for calculate dot product
for (int i = 0; i < n; i++)
{
       scalartripleproduct = scalartripleproduct + vect_u[i] *
           cross_P[i] ;
       cout << scalartripleproduct << endl;
}
cout << "Scalar triple product u . (v x w) : ";
cout << scalartripleproduct << endl;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_z;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_B_dz;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_z;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_C_dz;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
```

```
float o = 0;

// Create a vector with origin at (0,0,0) and terminal point at (u1,
    u2,u3)
pts_A_x .push_back(o);
pts_A_y .push_back(o);
pts_A_z .push_back(o);
pts_A_dx.push_back(u1);
pts_A_dy.push_back(u2);
pts_A_dz.push_back(u3);
// Create a vector with origin at (0,0,0) and terminal point at (v1,
    v2,v3)
pts_B_x .push_back(o);
pts_B_y .push_back(o);
pts_B_z .push_back(o);
pts_B_dx.push_back(v1);
pts_B_dy.push_back(v2);
pts_B_dz.push_back(v3);
// Create a vector with origin at (0,0) and terminal point at (w1,w2
    ,w3)
pts_C_x .push_back(o);
pts_C_y .push_back(o);
pts_C_z .push_back(o);
pts_C_dx.push_back(w1);
pts_C_dy.push_back(w2);
pts_C_dz.push_back(w3);
// Create a vector from cross product of vectors v and w
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(cross_P[0]);
pts_D_dy.push_back(cross_P[1]);
pts_D_dz.push_back(cross_P[2]);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-1:3]\nset yrange [-1:3]\nset zrange [-1:5]\n";
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
gp << "splot '-' with vectors title 'u', '-' with vectors title 'v
    ', '-' with vectors title 'w', '-' with vectors title 'v x w'\
    n";
gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
    pts_A_dy, pts_A_dz));
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
    pts_B_dy, pts_B_dz));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
    pts_C_dy, pts_C_dz));
```

```
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
    }
```

**C++ Code 123:** *main.cpp "Cross Product in 3D"*

To compile it with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- We define two functions at the beginning to compute dot product with **int dotProduct()** and to compute cross product with **void crossProduct()**.

```
int dotProduct(float vect_u[], float vect_v[])
{
       float product = 0;
       // Loop for calculate dot product
       for (int i = 0; i < n; i++)
       {
              product = product + vect_u[i] * vect_v[i];
       }
       return product;
}

void crossProduct(float vect_v[], float vect_w[], float cross_P
    [])
{
       cross_P[0] = vect_v[1] * vect_w[2] - vect_v[2] * vect_w
           [1];
       cross_P[1] = vect_v[2] * vect_w[0] - vect_v[0] * vect_w
           [2];
       cross_P[2] = vect_v[0] * vect_w[1] - vect_v[1] * vect_w
           [0];
}
```

- We define the vector $u, v,$ and $w$ and then call the functions **dotProduct** and **crossProduct** to compute the result and show them nicely.

```
float u1 = 3;
float u2 = -2;
float u3 = -5;

float v1 = 1;
float v2 = 4;
float v3 = -4;

float w1 = 0;
float w2 = 3;
float w3 = 2;
```

```
float vect_u[] = { u1, u2, u3 };
float vect_v[] = { v1, v2, v3 };
float vect_w[] = { w1, w2, w3 };
float cross_P[n];

cout << "Vector u:" << setw(15) << "vector v:" << setw(15) << "
    vector w:" << endl;

for (int i = 0; i < n; i++)
{
       cout << vect_u[i] << "\t" << "\t" << vect_v[i] << "\t"
           << "\t" << vect_w[i] ;
       cout << endl;
}
cout<<endl;

cout << "Dot product u . v : ";
cout << dotProduct(vect_u, vect_v) << endl;
cout << "Dot product v . w : ";
cout << dotProduct(vect_v, vect_w) << endl;
cout << "Dot product u . w : ";
cout << dotProduct(vect_u, vect_w) << endl;

crossProduct(vect_v, vect_w, cross_P);
cout << "Cross product v x w: (";

for (int i = 0; i < n-1; i++)
{
       cout << cross_P[i] << ", ";
}
cout << cross_P[n-1] ;
cout << ")" << endl;
```

- We input the vectors that will be plotted with gnuplot' **splot** from the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ and their cross product $\boldsymbol{u} \times \boldsymbol{v}$ that already computed above.

```
gp << "set xrange [-1:3]\nset yrange [-1:3]\nset zrange
    [-1:5]\n";
gp << "splot '-' with vectors title 'u', '-' with vectors
    title 'v', '-' with vectors title 'w', '-' with vectors
    title 'v x w'\n";
gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx
    , pts_A_dy, pts_A_dz));
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx
    , pts_B_dy, pts_B_dz));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx
    , pts_C_dy, pts_C_dz));
gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx
```

```
                    , pts_D_dy, pts_D_dz));
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/3D Plot Cross Product Vectors ]# ./main
Vector u:      vector v:      vector w:
3               1               0
-2              4               3
-5              -4              2

Dot product u . v : 15
Dot product v . w : 4
Dot product u . w : -16
Cross product v x w: (20, -2, 3)
Area of Parallelogram from vector v and w : 20.3224
θ between v and w (degrees) = 78.865
60
64
49
Scalar triple product u . (v x w) : 49
```

**Figure 23.26:** *The dot product and cross product computation of the vectors* $\boldsymbol{u}, \boldsymbol{v}$, *and* $\boldsymbol{w}$, *then plot all the vectors in* $\mathbb{R}^3$ *with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Cross Product Vectors/main.cpp**).*

# V. General Vector Spaces

[DF*] In Euclidean Vector Spaces we developed properties of vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$, we noticed patterns in various formulas that enabled us to extend the notion of a vector to an $n$-tuple of real numbers. Although $n$-tuple took us outside the realm of our "visual experience," it gave us a valuable tool for understanding and studying systems of linear equations.

**Definition 23.27: Vector Space Axioms**

Let $V$ be an arbitrary nonempty set of objects on which two operations are defined: addition, and multiplication by scalars.

- By addition we mean a rule for associating with each pair of objects $\boldsymbol{u}$ and $\boldsymbol{v}$ in $V$ an object $\boldsymbol{u}+\boldsymbol{v}$, called the sum of $\boldsymbol{u}$ and $\boldsymbol{v}$.
- By scalar multiplication we mean a rule for associating with each scalar $k$ and each object $\boldsymbol{u}$ in $V$ an object $k\boldsymbol{u}$, called the scalar multiple of $\boldsymbol{u}$ by $k$.

If the following axioms are satisfied by all objects $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ in $V$ and all scalars $k$ and $m$, then we call $V$ a vector space and we call the objects in $V$ vectors

1. If $\boldsymbol{u}$ and $\boldsymbol{v}$ are objects in $V$, then $\boldsymbol{u}+\boldsymbol{v}$ is in $V$
2. $\boldsymbol{u}+\boldsymbol{v}=\boldsymbol{v}+\boldsymbol{u}$
3. $\boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})=(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w}$
4. There is an object $\mathbf{0}$ in $V$, called a zero vector for $V$, such that
$$\mathbf{0}+\boldsymbol{u}=\boldsymbol{u}+\mathbf{0}=\boldsymbol{u}$$
for all $\boldsymbol{u}$ in $V$
5. For each $\boldsymbol{u}$ in $V$, there is an object $-\boldsymbol{u}$ in $V$, called a negative of $\boldsymbol{u}$, such that
$$\boldsymbol{u}+(-\boldsymbol{u})=(-\boldsymbol{u})+\boldsymbol{u}=\mathbf{0}$$
6. If $k$ is any scalar and $\boldsymbol{u}$ is any object in $V$, then $k\boldsymbol{u}$ is in $V$.
7. $k(\boldsymbol{u}+\boldsymbol{v})=k\boldsymbol{u}+k\boldsymbol{v}$
8. $(k+m)\boldsymbol{u}=k\boldsymbol{u}+m\boldsymbol{u}$
9. $k(m\boldsymbol{u})=(km)(\boldsymbol{u})$
10. $1\boldsymbol{u}=\boldsymbol{u}$

**Theorem 23.40: Properties of Vectors in Vector Space**

Let $V$ be a vector space, $\boldsymbol{u}$ a vector in $V$, and $k$ a scalar, then:

1. $0\boldsymbol{u} = \mathbf{0}$
2. $k\mathbf{0} = \mathbf{0}$
3. $(-1)\boldsymbol{u} = -\boldsymbol{u}$
4. If $k\boldsymbol{u} = \mathbf{0}$, then $k = 0$ or $\boldsymbol{u} = \mathbf{0}$

[DF*] Whenever we discover a new theorem about general vector spaces, we will at the same time be discovering a theorem about geometric vectors, vectors in $\mathbb{R}^n$, sequences, matrices, real-valued functions, and about any new kinds of vectors that we might discover.

[DF*] **Example:**
Let $V$ be the set of all ordered pairs of real numbers, and consider the following addition and scalar multiplication operations on $\boldsymbol{u} = (u_1, u_2)$ and $\boldsymbol{v} = (v_1, v_2)$:

$$\boldsymbol{u} + \boldsymbol{v} = (u_1 + v_1, u_2 + v_2), \quad k\boldsymbol{u} = (0, ku_2)$$

(a) Compute $\boldsymbol{u} + \boldsymbol{v}$ and $k\boldsymbol{u}$ for $\boldsymbol{u} = (2, 4)$, $\boldsymbol{v} = (1, -3)$, and $k = 5$.

(b) In words, explain why $V$ is closed under addition and scalar multiplication

(c) Since addition on $V$ is the standard addition operation on $\mathbb{R}^2$, certain vector space axioms hold for $V$ because they are known to hold for $\mathbb{R}^2$, which axioms are they?

(d) Show that axiom 10 fails and hence that $V$ is not a vector space under the given operations.

**Solution:**

(a) For addition

$$\begin{aligned} \boldsymbol{u} + \boldsymbol{v} &= (2, 4) + (1, -3) \\ &= (2 + 1, 4 + (-3)) \\ &= (3, 1) \end{aligned}$$

for scalar multiplication

$$k\boldsymbol{u} = k(2, 4) = 5(2, 4) = (0, 20)$$

because the rule is $k\boldsymbol{u} = (0, ku_2)$

(b) $V$ is closed under addition because the sum of any two real numbers is a real number, and it is closed under scalar multiplication because the product of two real numbers and 0 is a real number.

(c) Axioms 1-5 hold in $V$ because they also hold in $\mathbb{R}^2$.

(d) Remember that axiom 10 is

$$1\boldsymbol{u} = \boldsymbol{u}$$

Let $\boldsymbol{u} = (u_1, u_2)$ with $u_1 \neq 0$, then based on scalar multiplication rule on $V$

$$1\boldsymbol{u} = 1(u_1, u_2) = (0, u_2)$$

with

$$(0, u_2) \neq \boldsymbol{u}$$

thus axiom 10 fails to hold in $V$.

[DF*] **Example:**
Determine whether each set equipped with the given operations is a vector space. For those that are not vector spaces identify the vector space axioms that fail

(a) The set of all real numbers with the standard operations of addition and multiplication.

(b) The set of all pairs of real numbers of the form $(x, y)$, where $x \geq 0$, with the standard operations on $\mathbb{R}^2$.

(c) The set of all triples of real numbers with the standard vector addition but with scalar multiplication defined by

$$k(x, y, z) = (k^2x, k^2y, k^2z)$$

(d) The set of all $2 \times 2$ matrices of the form

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$$

with the standard matrix addition and scalar multiplication.

(e)

**Solution:** The standard operations for vectors $\boldsymbol{u} = (u_1, u_2)$ and $\boldsymbol{v} = (v_1, v_2)$ in the vector space:

$$\boldsymbol{u} + \boldsymbol{v} = (u_1 + v_1, u_2 + v_2), \quad k\boldsymbol{u} = (ku_1, ku_2)$$

(a) The set is a vector space with the standard operations.

(b) The set is not a vector space. Axiom 5: there is an object $-\boldsymbol{u}$ in the vector space, such that

$$\boldsymbol{u} + (-\boldsymbol{u}) = \boldsymbol{0}$$

fails to hold because of the restriction that $x \geq 0$.
Axiom 6:
If $k$ is any scalar and $\boldsymbol{u}$ is any object in the vector space, then $k\boldsymbol{u}$ is in the vector space.
Axiom 6 fails to hold for $k < 0$.

(c) The set is not a vector space.
Axiom 8:

$$(k + m)\boldsymbol{u} = k\boldsymbol{u} + m\boldsymbol{u}$$

we apply the axiom 8 to the scalar multiplication for the set of all triples of real numbers

$$\begin{aligned}(k + m)(x, y, z) &= k(x, y, z) + m(x, y, z) \\ &= (k^2x, k^2y, k^2z) + (m^2x, m^2y, m^2z) \\ &= (k^2)(x, y, z) + (m^2)(x, y, z) \\ &= (k^2 + m^2)(x, y, z)\end{aligned}$$

if we take $k + m$ as a scalar then it fails to hold because

$$(k + m)^2 = k^2 + 2km + m^2$$

$$(k + m)^2 \neq k^2 + m^2$$

(d) For standard matrix addition

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} + \begin{bmatrix} c & 0 \\ 0 & d \end{bmatrix} = \begin{bmatrix} a+c & 0 \\ 0 & b+d \end{bmatrix}$$

it is closed under standard matrix addition. And for the scalar multiplication, for any scalar $k$

$$k \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} = \begin{bmatrix} ka & 0 \\ 0 & kb \end{bmatrix}$$

it is closed under scalar multiplication. Thus, the set is a vector space with the given operations.

**Definition 23.28: Subspace**

A subset $W$ of a vector space $V$ is called a subspace of $V$ if $W$ is itself a vector space under the addition and scalar multiplication defined on $V$.

In general, to show that a nonempty set $W$ with two operations is a vector space one must verify the ten vector space axioms.

[DF*] It is necessary to verify that $W$ is closed under addition and scalar multiplication since it is possible that adding two vectors in $W$ or multiplying a vector in $W$ by a scalar produces a vector in $V$ that is outside of $W$.

[DF*] Some axioms are inherited from $V$ to $W$, but there are axioms that are not inherited by $W$, they are

1. Axiom 1 - Closure of $W$ under addition
2. Axiom 4 - Existence of zero vector in $W$
3. Axiom 5 - Existence of a negative in $W$ for every vector in $W$
4. Axiom 6 - Closure of $W$ under scalar multiplication

So all these must be verified to prove that $W$ is a subspace of $V$.

**Theorem 23.41: Subspace of a Vector Space**

If $W$ is a set of one or more vectors in a vector space $V$, then $W$ is a subspace of $V$ if and only if the following conditions hold

(a) If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $W$, then $\boldsymbol{u}+\boldsymbol{v}$ is in $W$.

(b) If $k$ is any scalar and $\boldsymbol{u}$ is any vector in $W$, then $k\boldsymbol{u}$ is in $W$.

[DF*] Subspaces of $\mathbb{R}^2$:

- $\{\mathbf{0}\}$
- Lines through the origin
- $\mathbb{R}^2$

Subspaces of $\mathbb{R}^3$:

- $\{\mathbf{0}\}$
- Lines through the origin
- Planes through the origin

- $\mathbb{R}^3$

Subspaces of $M_{nn}$ (symmetric matrices of size $n \times n$):

- The sets of upper triangular matrices
- The sets of lower triangular matrices
- The sets of diagonal matrices

[DF*] $C(-\infty, \infty)$, the set of continuous functions on $(-\infty, \infty)$ is a subspace of $F(-\infty, \infty)$.

A sum of continuous functions is continuous and a constant times a continuous function is continuous.

[DF*] A function with a continuous derivative is said to be continuously differentiable.

The sum of two continuously differentiable functions is continuously differentiable and a constant times a continuously differentiable function is continuously differentiable.

The functions that are continuously differentiable on $(-\infty, \infty)$ form a subspace $F(-\infty, \infty)$.

- $C^1(-\infty, \infty)$ is a subspace of $F(-\infty, \infty)$ where the superscript emphasizes that the first derivative is continuous.
- $C^m(-\infty, \infty)$ is a subspace of $F(-\infty, \infty)$ where the superscript emphasizes that the $m$-th derivative is continuous.
- $C^\infty(-\infty, \infty)$ is a subspace of $F(-\infty, \infty)$ with continous derivative of all orders in $(-\infty, \infty)$.

[DF*] A polynomial is a function that can be expressed in the form

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n \tag{23.64}$$

where $a_0, a_1, \ldots, a_n$ are constants.
The sum of two polynomials is a polynomial, and a constant times a polynomial is a polynomial.
$P_\infty$ is a subspace of $F(-\infty, \infty)$, the set of all polynomials that is closed under addition and scalar multiplication.

[DF*] The hierarchy of function spaces

$$P_n \subset C^\infty(-\infty, \infty) \subset C^m(-\infty, \infty) \subset C^1(-\infty, \infty) \subset C(-\infty, \infty) \subset F(-\infty, \infty)$$

**Theorem 23.42: Creating a New Subspace**

If $W_1, W_2, \ldots, W_r$ are subspaces of a vector space $V$, then the intersection of these subspaces is also a subspace of $V$.

**Definition 23.29: Linear Combination**

If $\boldsymbol{w}$ is a vector in a vector space $V$, then $\boldsymbol{w}$ is said to be a linear combination of the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r$ in $V$

**Theorem 23.43: Linear Combination and Subspace**

If $S = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_r\}$ is a nonempty set of vectors in a vetor space $V$, then:

1. The set $W$ of all possible linear combinations of the vectors in $S$ is a subspace of $V$.
2. The set $W$ in part (a) is the "smallest" subspace of $V$ that contains all of the vectors in $S$ in the sense that any other subspace that contains those vectors contains $W$.

**Definition 23.30: Span of a Subspace**

The subspace of a vector space $V$ that is formed from all possible linear combinations of the vectors in a nonempty set $S$ is called the span of $S$, and we say that the vectors in $S$ span that subspace. If $S = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_r\}$, then we denote the span of $S$ by

$$\text{span}\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_r\} \quad \text{or} \quad \text{span}(S)$$

[DF*] The standard unit vectors in $\mathbb{R}^n$ are

$$\boldsymbol{e}_1 = (1, 0, 0, \ldots, 0), \quad \boldsymbol{e}_2 = (0, 1, 0, \ldots, 0), \ldots, \quad \boldsymbol{e}_n = (0, 0, 0, \ldots, 1)$$

these vectors span $\mathbb{R}^n$ since every vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ in $\mathbb{R}^n$ can be expressed as

$$\boldsymbol{v} = v_1\boldsymbol{e}_1 + v_2\boldsymbol{e}_2 + \cdots + v_n\boldsymbol{e}_n$$

[DF*] The polynomials $1, x, x^2, \ldots, x^n$ span the vector space $P_n$ since each polynomial $\boldsymbol{p}$ in $P_n$ can be written as

$$\boldsymbol{p} = a_0 + a_1x + \cdots + a_nx^n$$

which is a linear combination of $1, x, x^2, \ldots, x^n$. We can denote this by writing

$$P_n = \text{span}\{1, x, x^2, \ldots, x^n\}$$

[DF*] **Example:**
Consider the vectors $\boldsymbol{u} = (1, 2, -1)$ and $\boldsymbol{v} = (6, 4, 2)$ in $\mathbb{R}^3$. Show that $\boldsymbol{w} = (9, 2, 7)$ is a linear combination of $\boldsymbol{u}$ and $\boldsymbol{v}$ and that $\boldsymbol{w'} = (4, -1, 8)$ is not a linear combination of $\boldsymbol{u}$ and $\boldsymbol{v}$.
**Solution:**
In order for $\boldsymbol{w}$ to be a linear combination of $\boldsymbol{u}$ and $\boldsymbol{v}$, there must be scalars $k_1$ and $k_2$ such that

$$\boldsymbol{w} = k_1\boldsymbol{u} + k_2\boldsymbol{v}$$

that is,

$$\begin{aligned}(9, 2, 7) &= k_1(1, 2, -1) + k_2(6, 4, 2)\\ (9, 2, 7) &= k_1(1, 2, -1) + k_2(6, 4, 2)\\ (9, 2, 7) &= (k_1 + 6k_2, 2k_1 + 4k_2, -k_1 + 2k_2)\end{aligned}$$

Equating corresponding components gives

$$\begin{array}{rrcl} k_1 & + 6k_2 & = & 9 \\ 2k_1 & + 4k_2 & = & 2 \\ -k_1 & + 2k_2 & = & 7 \end{array}$$

Solving this system using Gaussian elimination yields $k_1 = -3,\ k_2 = 2$, so

$$\boldsymbol{w} = -3\boldsymbol{u} + 2\boldsymbol{v}$$

In this problem , we have less unknowns $(k_1, k_2)$ and more equations, thus this system is called over-determined system, one of many way to solve it is by using least squares method to determine the solutions.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Over-Determined
 System with Armadillo ]# ./main
Matrix A:
   1.0000   6.0000
   2.0000   4.0000
  -1.0000   2.0000

Vector B:
   9.0000
   2.0000
   7.0000

Solution:
  -3.0000
   2.0000
```

**Figure 23.27:** *The Gaussian elimination for two rows of the linear system above, we omit the third one since it will make the matrix into singular, if you try to do it by hand the third equation will become the scalar multiple of the second equation (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Over-Determined System with Armadillo/main.cpp**).*

[DF*] **Example:**
Determine whether $\boldsymbol{v}_1 = (1,1,2),\ \boldsymbol{v}_2 = (1,0,1)$, and $\boldsymbol{v}_3 = (2,1,3)$ span the vector space in $\mathbb{R}^3$.
**Solution:**
We must determine whether an arbitrary vector $\boldsymbol{b} = (b_1, b_2, b_3)$ in $\mathbb{R}^3$ can be expressed as a linear combination

$$\boldsymbol{b} = k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + k_3\boldsymbol{v}_3$$

of the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$. Expressing this equation in terms of components gives

$$\begin{aligned}(b_1, b_2, b_3) &= k_1(1,1,2) + k_2(1,0,1) + k_3(2,1,3)\\ (b_1, b_2, b_3) &= (k_1 + k_2 + 2k3, k_1 + k_3, 2k_1 + k_2 + 3k_3)\end{aligned}$$

$$\begin{array}{rrrcl} k_1 & +\,k_2 & +\,2k_3 & = & b_1\\ k_1 & + & +\,k_3 & = & b_2\\ 2k_1 & +\,k_2 & +\,3k_3 & = & b_3\end{array}$$

our problem reduces to ascertaining whether this system is consistent for all values of $b_1, b_2$, and $b_3$. One way of doing this is to check whether the system is consistent if and only if its coefficient matrix

$$\begin{bmatrix}1 & 1 & 2\\ 1 & 0 & 1\\ 2 & 1 & 3\end{bmatrix}$$

has a nonzero determinant. The determinant of $A$ is 0, you can check it with hand or use C++ code that have already been explained in the previous section. Since $\det(A) = 0$, then the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$ do not span $\mathbb{R}^3$.

```
m Textfile ]# ./main
The matrix A is
     1       1       2
     1       0       1
     2       1       3

Determinant of the matrix is 0
```

**Figure 23.28:** *The computation to find the determinant for square matrix $A$ of size $3 \times 3$ with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Determinant of Square Matrix from Textfile/main.cpp**).*

[DF*] The solutions of a homogeneous linear system

$$A\boldsymbol{x} = \mathbf{0}$$

of $m$ equations in $n$ unknowns can be viewed as vectors in $\mathbb{R}^n$.

**Theorem 23.44: Solution of Homogeneous Linear System**

The solution set of a homogeneous linear system $A\boldsymbol{x} = \mathbf{0}$ in $n$ unknowns is a subspace of $\mathbb{R}^n$.

[DF*] **Example:**
Consider the linear system

$$\begin{bmatrix} 1 & -2 & 3 \\ 2 & -4 & 6 \\ 3 & -6 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

if we apply Gaussian elimination we will obtain infinitely many solutions, thus we will set the third variable into parameter $t$, and the second variable into parameter $s$

$$x = 2s - 3t, \quad y = s, \quad z = t$$

from which it follows that

$$x = 2y - 3z, \quad x - 2y + 3z = 0$$

this is the equation of a plane through the origin that has $\boldsymbol{n} = (1, -2, 3)$ as a normal.

[DF*] The solution set of every homogeneous system of $m$ equations in $n$ unknowns is a subspace of $\mathbb{R}^n$. But, the solution set of a nonhomogeneous system of $m$ equations in $n$ unknowns is a subspace of $\mathbb{R}^n$ is never true, since there are two possible scenarios: first, the system may not have any solution at all, and second, if there are solutions, then the solution set will not be closed under either addition or under scalar multiplication.

**Theorem 23.45: Span and Linear Combination**

If $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ and $S' = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k\}$ are nonempty sets of vectors in a vector space $V$, then

$$\text{span}\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\} = \text{span}\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k\}$$

if and only if each vector in $S$ is a linear combination of those in $S'$, and each vector in $S'$ is a linear combination of those in $S$.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Homogeneous Sys
tem ]# ./main

Matrix A:
     1      -2       3       0
     2      -4       6       0
     3      -6       9       0


The Gaussian Elimination process:
     3      -6       9       0
     0       0       0       0
     0       0       0       0

Singular Matrix.
May have infinitely many solutions.
```

**Figure 23.29:** *The computation to find the reduced echelon form of the linear system matrix with C++, it resulting in singular matrix that has infinitely many solutions or can be represented as a plane in 3-dimension (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Homogeneous System/main.cpp**).*

[DF*] **Example:**
Determine which of the following are subspaces of $\mathbb{R}^3$

(a) All vectors of the form $(a, 0, 0)$

(b) All vectors of the form $(a, 1, 0)$

(c) All vectors of the form $(a, b, c)$, where $b = a + c$

(d) All vectors of the form $(a, b, c)$, where $c = b - a$

(e) All vectors of the form $(a, -a, 0)$

**Solution:**

(a) All vectors of the form $(a, 0, 0)$ is a subspace of $\mathbb{R}^3$

(b) All vectors of the form $(a, 1, 0)$ is not a subspace of $\mathbb{R}^3$, since $(a_1, 1, 1) + (a_2, 1, 1) = (a_1 + a_2, 2, 2)$ is not in the set.

(c) All vectors of the form $(a, b, c)$, where $b = a + c$ is a subspace of $\mathbb{R}^3$

(d) All vectors of the form $(a, b, c)$, where $c = b - a$

(e) All vectors of the form $(a, -a, 0)$

[DF*] **Example:**
Determine which of the following are subspaces of $P_3$

(a) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ where $a_1 = a_2$.

(b) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ where $a_0 = 0$.

(c) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ in which $a_0, a_1, a_2$, and $a_3$ are integers.

(d) All polynomials of the form $a_0 + a_1x$, where $a_0$ and $a_1$ are real numbers.

**Solution:**

(a) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ where $a_1 = a_2$ is a subspace of $P_3$.

(b) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ where $a_0 = 0$ is a subspace of $P_3$.

(c) All polynomials of the form $a_0 + a_1x + a_2x^2 + a_3x^3$ in which $a_0, a_1, a_2$, and $a_3$ are integers is not a subspace of $P_3$ since for $k \in \mathbb{R}$

$$k(a_0 + a_1x + a_2x^2 + a_3x^3)$$

is not in the set of $P_3$ for all noninteger value of $k$.

(d) All polynomials of the form $a_0 + a_1x$, where $a_0$ and $a_1$ are real numbers is a subspace of $P_3$

[DF*] **Example:**
Determine whether the solution space of the system $A\boldsymbol{x} = \mathbf{0}$ is a line through the origin, or the origin only. If it is a plane, find an equation for it. If it is a line, find parametric equations for it.

(a)

$$A = \begin{bmatrix} 1 & -2 & 6 \\ 3 & -6 & 18 \\ -7 & 14 & -42 \end{bmatrix}$$

(b)

$$A = \begin{bmatrix} 1 & -2 & 3 \\ -3 & 6 & 9 \\ -2 & 4 & 6 \end{bmatrix}$$

(c)

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 9 & -11 & 3 \\ 3 & -4 & 1 \end{bmatrix}$$

(d)

$$A = \begin{bmatrix} 1 & 2 & -6 \\ 1 & 4 & 4 \\ 3 & 10 & 6 \end{bmatrix}$$

**Solution:**

(a) Since $\det(A) = 0$, we then reduce the matrix $A$ into reduced row echelon form. The solution set is

$$x = 2s - 6t,\ y = s,\ z = t$$

or

$$x - 2y + 6z = 0$$

which is a plane through the origin.

(b) Since $\det(A) = 0$, we then reduce the matrix $A$ into reduced row echelon form. The solution set is

$$x = 2t,\ y = t,\ z = 0$$

which is a line through the origin.

(c) Since $\det(A) = 1$, we then reduce the matrix $A$ into reduced row echelon form. We will obtain the solution as

$$x = 0,\ y = 0,\ z = 0$$

the system only has the trivial solution, which is the origin $(0, 0, 0)$.

```
trix with SymbolicC++ (copy 1) ]# ./main
A:
[ 1   -2  6 ]
[ 3   -6  18]
[ -7  14 -42]

A augmented matrix:
[ 1   -2  6   0 ]
[ 3   -6  18  0 ]
[ -7  14 -42  0 ]

det(A):
0
inv(A):
[-nan -nan -nan]
[-nan -nan -nan]
[-nan -nan -nan]


A (in reduced row form) :
[  1   -2    6    0 ]
[  0    0    0    0 ]
[  0    0  -nan -nan]

A (in row reduced echelon form) :
[  1   -2    6    0 ]
[-nan -nan -nan -nan]
[-nan -nan -nan -nan]
```

**Figure 23.30:** *The computation to find reduced row echelon form with Gaussian Elimination with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Numerical Matrix with SymbolicC++/main.cpp**).*

```
 with SymbolicC++ (copy 1) ]# ./main
A:
[ 1 -2  3]
[-3  6  9]
[-2  4 -6]

A augmented matrix:
[ 1 -2  3  0]
[-3  6  9  0]
[-2  4 -6  0]

det(A):
0
inv(A):
[-inf -nan -inf]
[-inf -nan -inf]
[-nan -nan -nan]


A (in reduced row form) :
[  1   -2    3    0 ]
[  0    0   18    0 ]
[  0    0  -nan -nan]

A (in row reduced echelon form) :
[  1   -2    3    0 ]
[-nan -nan  inf -nan]
[-nan -nan -nan -nan]
```

**Figure 23.31:** *The computation to find reduced row echelon form with Gaussian Elimination with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Numerical Matrix with SymbolicC++/main.cpp**).*

```
 with SymbolicC++ (copy 1) ]# ./main
A:
[ 1   0   0 ]
[ 9  -11  3 ]
[ 3   -4  1 ]

A augmented matrix:
[ 1   0   0   0 ]
[ 9  -11  3   0 ]
[ 3   -4  1   0 ]

det(A):
1
inv(A):
[ 1   0   0 ]
[ 0   1   -3]
[ -3  4  -11]


A (in reduced row form) :
[     1          0          0          0    ]
[     0         -11         3          0    ]
[     0          0     -0.0909091      0    ]

A (in row reduced echelon form) :
[    1         0         0         0    ]
[    -0        1     -0.272727     -0   ]
[    -0        -0        1         -0   ]
```

**Figure 23.32:** *The computation to find reduced row echelon form with Gaussian Elimination with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Numerical Matrix with SymbolicC++/main.cpp**).*

(d) Since $\det(A) = 8$, we then reduce the matrix $A$ into reduced row echelon form. We will obtain the solution as

$$x = 0,\ y = 0,\ z = 0$$

the system only has the trivial solution, which is the origin $(0, 0, 0)$.

```
 with SymbolicC++ (copy 1) ]# ./main
A:
[ 1  2 -6]
[ 1  4  4]
[ 3 10  6]

A augmented matrix:
[ 1  2 -6  0]
[ 1  4  4  0]
[ 3 10  6  0]

det(A):
8
inv(A):
[  -2    -9    4  ]
[ 0.75   3   -1.25]
[-0.25  -0.5  0.25]


A (in reduced row form) :
[ 1  2 -6  0]
[ 0  2 10  0]
[ 0  0  4  0]

A (in row reduced echelon form) :
[ 1  2 -6  0]
[ 0  1  5  0]
[ 0  0  1  0]
```

**Figure 23.33:** *The computation to find reduced row echelon form with Gaussian Elimination with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Numerical Matrix with SymbolicC++/main.cpp**).*

[DF*] **Example:**
Show that the set of continuous functions $\boldsymbol{f} = f(x)$ on $[a, b]$ such that

$$\int_a^b f(x)\ dx = 0$$

is a subspace of $C[a, b]$.
**Solution:**
Let $\boldsymbol{f} = f(x)$ and $\boldsymbol{g} = g(x)$ be the elements of the set. Then

$$\begin{aligned}\boldsymbol{f} + \boldsymbol{g} &= \int_a^b f(x) + g(x)\ dx \\ &= \int_a^b f(x)\ dx + \int_a^b g(x)\ dx \\ &= 0\end{aligned}$$

and $\boldsymbol{f} + \boldsymbol{g}$ is in the set. Let $k$ be any scalar. Then

$$\begin{aligned}k\boldsymbol{f} &= \int_a^b kf(x)\ dx \\ &= k \cdot 0 \\ &= 0\end{aligned}$$

so $k\boldsymbol{f}$ is in the set. Thus the set is a subspace of $C[a, b]$.

[DF*] In a rectangular $xy$-coordinate system every vector in the plane can be expressed in exactly one way as a linear combination of the standard unit vectors.

**Definition 23.31: Linear Independence**

If $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ is a nonempty set of vectors in a vector space $V$, then the vector equation

$$k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r = \mathbf{0}$$

has at least one solution, namely

$$k_1 = 0, \quad k_2 = 0, \quad \ldots, \quad k_r = 0$$

we call this the trivial solution. If this is the only solution, then $S$ is said to be a linearly independent set. If there are solutions in addition to the trivial solution, then $S$ is said to be a linearly dependent set.

[DF*] To show that a linear system has a nontrivial solutions you can

1. Solve it directly with Gaussian Elimination, when you have this vector equation

$$k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r = \mathbf{0}$$

   then turns it into homogeneous system and make a coefficient matrix out of it.
2. Check the determinant of the coefficient matrix, if the determinant is zero then the coefficient matrix has infinitely many solutions / nontrivial solutions. If the determinant is not zero then the system has only the trivial solution, and the vectors are linearly independent.

**Theorem 23.46: Interpretation of Linear Independence**

A set $S$ with two or more vectors is

(a) Linearly dependent if and only if at least one of the vectors in $S$ is expressible as a linear combinatino of the other vectors in $S$.

(b) Linearly independent if and only if no vector in $S$ is expressible as a linear combination of the other vector in $S$.

**Theorem 23.47: Sets With One or Two Vectors**

(a) A finite set that contains $\mathbf{0}$ is linearly dependent.

(b) A set with exactly one vector is exactly independent if and only if that vector is not $\mathbf{0}$.

(c) A set with exactly two vectors is linearly independent if and only if neither vector is a scalar multiple of the other.

**Theorem 23.48: Geometric Interpretation of Linear Independence**

Let $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ be a set of vectors in $\mathbb{R}^n$. If $r > n$, then $S$ is linearly dependent.

[DF*] Suppose that

$$\begin{aligned} \boldsymbol{v}_1 &= (v_{11}, v_{12}, \ldots, v_{1n}) \\ \boldsymbol{v}_2 &= (v_{21}, v_{22}, \ldots, v_{2n}) \\ &\vdots\vdots \\ \boldsymbol{v}_r &= (v_{r1}, v_{r2}, \ldots, v_{rn}) \end{aligned}$$

and consider the equation

$$k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r = \mathbf{0}$$

If we express both sides of this equation in terms of components and then equate the corresponding components, we obtain the system

$$\begin{array}{llllll} v_{11}k_1 & + v_{21}k_2 & + \ldots & + v_{r1}k_r & = 0 \\ v_{21}k_1 & + v_{22}k_2 & + \ldots & + v_{r2}k_r & = 0 \\ \vdots & \vdots & & \vdots & \vdots \\ v_{1n}k_1 & + v_{2n}k_2 & + \ldots & + v_{rn}k_r & = 0 \end{array}$$

This is a homogeneous system of $n$ equations in the $r$ unknowns $k_1, k_2, \ldots, k_r$. Since $r < n$, then the system has nontrivial solutions. Therefore $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ is a linearly dependent set.

[DF*] Sometimes linear dependence of functions can be deduced from known identities. But, there is no general method that can be used to determine whether a set of functions is linear independent or linear dependent.

**Definition 23.32: Wronskian**

If $\boldsymbol{f}_1 = f_1(x)$, $\boldsymbol{f}_2 = f_2(x), \ldots, \boldsymbol{f}_n = f_n(x)$ are functions that are $n-1$ times differentiable on the interval $(-\infty, \infty)$, then the determinant

$$\begin{vmatrix} f_1(x) & f_2(x) & \ldots & f_n(x) \\ f_1'(x) & f_2'(x) & \ldots & f_n'(x) \\ \vdots & \vdots & & \vdots \\ f_1^{(n-1)} & f_2^{(n-1)} & \ldots & f_n^{(n-1)} \end{vmatrix}$$

is called the Wronskian of $f_1, f_2, \ldots, f_n$.

**Theorem 23.49: Wronskian and Linear Independence of Functions**

If the functions $\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots, \boldsymbol{f}_n$ have $n-1$ continuous derivatives on the interval $(-\infty, \infty)$, and if the Wronskian of these functions is not identically zero on $(-\infty, \infty)$, then these functions form a linearly independent set of vectors in $C^{(n-1)}(-\infty, \infty)$

[DF*] **Example:**
For which real values of $\lambda$ do the following vectors form a linearly dependent set in $\mathbb{R}^3$:

$$\boldsymbol{v}_1 = \left(\lambda, -\frac{1}{2}, -\frac{1}{2}\right), \quad \boldsymbol{v}_2 = \left(-\frac{1}{2}, \lambda, -\frac{1}{2}\right), \quad \boldsymbol{v}_3 = \left(-\frac{1}{2}, -\frac{1}{2}, \lambda\right)$$

**Solution:**
The equation

$$k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + k_3\boldsymbol{v}_3 = \mathbf{0}$$

generates the homogeneous system

$$\begin{aligned}
\lambda k_1 - \frac{1}{2}k_2 - \frac{1}{2}k_3 &= 0 \\
-\frac{1}{2}k_1 + \lambda k_2 - \frac{1}{2}k_3 &= 0 \\
-\frac{1}{2}k_1 - \frac{1}{2}k_2 + \lambda k_3 &= 0
\end{aligned}$$

thus

$$\begin{aligned}
\det\begin{bmatrix} \lambda & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \lambda & -\frac{1}{2} \\ -\frac{1}{2} & -\frac{1}{2} & \lambda \end{bmatrix} &= \frac{1}{4}(4\lambda^3 - 3\lambda - 1) \\
&= \frac{1}{4}(\lambda - 1)(2\lambda + 1)^2
\end{aligned}$$

For $\lambda = 1$ and $\lambda = -\frac{1}{2}$, the determinant is zero and the vectors form a linearly dependent set.

[DF*] **Example:**
Show that if $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ is linearly independent and $\boldsymbol{v}_3$ does not lie in span $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, then $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is linearly independent.
**Solution:**
If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ were linearly dependent, there would be a nonzero solution to

$$k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + k_3\boldsymbol{v}_3 = \mathbf{0}$$

with $k_3 \neq 0$, since $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are linearly independent.
Solving for $\boldsymbol{v}_3$ gives

$$\boldsymbol{v}_3 = -\frac{k_1}{k_3}\boldsymbol{v}_1 - \frac{k_2}{k_3}\boldsymbol{v}_2$$

which contradicts that $\boldsymbol{v}_3$ is not in span $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$. Thus, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is linearly independent.

[DF*] Rectangular coordinate systems are common, but they are not essential. There are nonrectangular coordinate systems in 2-space, 3-space, to the $n$-space with equal spacing, skew axes (the major axis is not orthogonal to each other) with equal spacing or unequal spacing. Our common rectangular coordinate system has perpendicular axes.

**Definition 23.33: Basis for a Vector Space**

If $V$ is any vector space and $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a finite set of vectors in $V$, then $S$ is called a basis for $V$ if the following two conditions hold:

(a) $S$ is linearly independent.

(b) $S$ spans $V$.

[DF*] To prove the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ are linearly independent we must show that

$$c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n = \mathbf{0} \tag{23.65}$$

has only the trivial solution.
To prove that the vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ span $\mathbb{R}^n$, we must show that every vector $\boldsymbol{b} = (b_1, b_2, \ldots, b_n)$ in $\mathbb{R}^n$ can be expressed as

$$c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n = \boldsymbol{b} \tag{23.66}$$

From the two equations above we will have two linear systems, the first is homogeneous (to prove for linear independence) and the second one is nonhomogeneous system (to prove that the vectors span), both systems will have the same coefficient matrix. The nonhomogeneous system is consistent for all values of $b_1, b_2, \ldots, b_n$.

**Theorem 23.50: Uniqueness of Basis Representation**

If $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a basis for a vector space $V$, then every vector $\boldsymbol{v}$ in $V$ can be expressed in the form $\boldsymbol{v} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n$ in exactly one way.

**Definition 23.34: Coordinate Vector**

If $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a basis for a vector space $V$ (the order of the vectors in $S$ remains fixed), and

$$\boldsymbol{v} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n$$

is the expression for a vector $\boldsymbol{v}$ in terms of the basis $S$, then the scalars $c_1, c_2, \ldots, c_n$ are called the coordinates of $\boldsymbol{v}$ relative to the basis $S$. The vector $(c_1, c_2, \ldots, c_n)$ in $\mathbb{R}^n$ constructed from these coordinates is called the coordinate vector of $\boldsymbol{v}$ relative to $S$; it is denoted by

$$(\boldsymbol{v})_S = (c_1, c_2, \ldots, c_n)$$

[DF*] Sometimes it will be desirable to write a coordinate vector as a column matrix, in which case we will denote it using square brackets as

$$[\boldsymbol{v}]_S = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

we will refer to $[\boldsymbol{v}]_S$ as a coordinate matrix and reserve the terminology coordinate vector for the comma delimited form $(\boldsymbol{v})_S$.

[DF*] $(\boldsymbol{v})_S$ is a vector in $\mathbb{R}^n$, once basis $S$ is given for a vector space $V$, it establishes a one-to-one correspondence between vectors $\boldsymbol{v} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n$ in $V$ and vectors $(\boldsymbol{v})_S = (c_1, c_2, \ldots, c_n)$ in $\mathbb{R}^n$.

[DF*] The coordinate vector for the polynomial

$$\boldsymbol{p}(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx_n$$

relative to the standard basis for the vector space $P_n$ can be determined when we see the formula $\boldsymbol{p}(x)$ expresses the polynomial as a linear combination of the standard basis vector

$$S = \{1, x, x^2, \ldots, x^n\}$$

Thus, the coordinate vector for $\boldsymbol{p}$ relative to $S$ is

$$(\boldsymbol{p})_S = (c_0, c_1, c_2, \ldots, c_n)$$

[DF*] **Example:**
Find the coordinate vector $\boldsymbol{v}$ relative to the basis $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ with

$$\boldsymbol{v}_1 = (3,2,1), \quad \boldsymbol{v}_2 = (-2,1,0), \quad \boldsymbol{v}_3 = (5,0,0)$$

and $\boldsymbol{v} = (3,4,3)$.
**Solution:**
To find $(\boldsymbol{v})_S$ we must first express $\boldsymbol{v}$ as a linear combination of the vectors in $S$; that is, we must find values of $c_1, c_2$, and $c_3$ such that

$$\boldsymbol{v} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3$$

or, in terms of components,

$$(3,4,3) = c_1(3,2,1) + c_2(-2,1,0) + c_3(5,0,0)$$

Equating corresponding components gives

$$\begin{array}{rrrcl} 3c_1 & -\,2c_2 & +\,5c_3 & = & 3 \\ 2c_1 & +\,c_2 & & = & 4 \\ c_1 & & & = & 3 \end{array}$$

Solving this system we obtain $c_1 = 3, c_2 = -2, c_3 = -2$. Therefore,

$$(\boldsymbol{v})_S = (3,-2,-2)$$

```
ith Armadillo ]# ./main
Matrix A:
   3.0000  -2.0000   5.0000
   2.0000   1.0000        0
   1.0000        0        0

Vector B:
   3.0000
   4.0000
   3.0000

Solution:
   3.0000
  -2.0000
  -2.0000
```

**Figure 23.34:** *The computation to find the solution $c_1, c_2, c_3$ for the linear system above with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Square Matrix with Armadillo/main.cpp**).*

[DF*] **Example:**
Find the coordinate vector of $\boldsymbol{p}$ relative to the basis $S = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\}$ with

$$\boldsymbol{p}_1 = 1 + x, \quad \boldsymbol{p}_2 = 1 + x^2, \quad \boldsymbol{p}_3 = x + x^2$$

and $\boldsymbol{p} = 2 - x + x^2$.
**Solution:**

We must first express $\boldsymbol{p}$ as a linear combination of the basis $S$; that is, we must find values of $c_1, c_2$, and $c_3$ such that

$$\boldsymbol{p} = c_1\boldsymbol{p}_1 + c_2\boldsymbol{p}_2 + c_3\boldsymbol{p}_3$$

or, in terms of components,

$$2 - x + x^2 = c_1(1 + x) + c_2(1 + x^2) + c_3(x + x^2)$$

Equating corresponding components gives

$$\begin{array}{rrrrl} c_1 & +\,c_2 & & = & 2 \\ c_1 & & +\,c_3 & = & -1 \\ & c_2 & +\,c_3 & = & 1 \end{array}$$

Solving this system we obtain $c_1 = 0, c_2 = 2, c_3 = -1$. Therefore,

$$(\boldsymbol{p})_S = (0, 2, -1)$$

```
ith Armadillo ]# ./main
Matrix A:
   1.0000   1.0000        0
   1.0000        0   1.0000
        0   1.0000   1.0000

Vector B:
   2.0000
  -1.0000
   1.0000

Solution:
  -3.5327e-16
   2.0000e+00
  -1.0000e+00
```

**Figure 23.35:** *The computation to find the solution $c_1, c_2, c_3$ for the linear system above with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Square Matrix with Armadillo/main.cpp**).*

[DF*] **Example:**
Show that $S = \{A_1, A_2, A_3, A_4\}$ is a basis for $M_{22}$ and express $A$ as a linear combination of the basis vectors

$$A_1 \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad A_2 \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad A_3 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

$$A_4 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad A \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

**Solution:**
Solving

$$c_1A_1 + c_2A_2 + c_3A_3 + c_4A_4 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

for linear independence. Then solving for

$$c_1A_1 + c_2A_2 + c_3A_3 + c_4A_4 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

gives the systems

$$\begin{array}{lllll} c_1 & & & & = 0 \\ c_1 & + c_2 & & & = 0 \\ c_1 & + c_2 & + c_3 & & = 0 \\ c_1 & + c_2 & + c_3 & + c_4 & = 0 \end{array}$$

and

$$\begin{array}{lllll} c_1 & & & & = a \\ c_1 & + c_2 & & & = b \\ c_1 & + c_2 & + c_3 & & = c \\ c_1 & + c_2 & + c_3 & + c_4 & = d \end{array}$$

thus we need to solve the system

$$\begin{array}{lllll} c_1 & & & & = 1 \\ c_1 & + c_2 & & & = 0 \\ c_1 & + c_2 & + c_3 & & = 1 \\ c_1 & + c_2 & + c_3 & + c_4 & = 0 \end{array}$$

or in matrix form

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

check the determinant to see whether they span the vector space or not

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{vmatrix} = 1$$

since the determinant of the matrix above is not zero, so $\{A_1, A_2, A_3, A_4\}$ spans $M_{22}$. Now we will solve the linear system to show that $A$ is a linear combination of the basis vectors $\{A_1, A_2, A_3, A_4\}$, we will obtain

$$c_1 = 1,\ c_2 = -1,\ c_3 = 1,\ c_4 = -1$$

Thus

$$A = A_1 - A_2 + A_3 - A_4$$

[DF*] **Example:**
Show that $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\}$ is a basis for $P_2$, and express $\boldsymbol{p}$ as a linear combination of the basis vectors with

$$\boldsymbol{p}_1 = 1 + x + x^2, \quad \boldsymbol{p}_2 = x + x^2, \quad \boldsymbol{p}_3 = x^2$$

and $\boldsymbol{p} = 7 - x + 2x^2$.
**Solution:**
Solving the systems

$$c_1\boldsymbol{p}_1 + c_2\boldsymbol{p}_2 + c_3\boldsymbol{p}_3 = \mathbf{0}$$

$$c_1\boldsymbol{p}_1 + c_2\boldsymbol{p}_2 + c_3\boldsymbol{p}_3 = a + bx + cx^2$$

```
Matrix A:
   1.0000        0        0        0
   1.0000   1.0000        0        0
   1.0000   1.0000   1.0000        0
   1.0000   1.0000   1.0000   1.0000

Vector B:
   1.0000
        0
   1.0000
        0

Solution:
   1.0000
  -1.0000
   1.0000
  -1.0000
```

**Figure 23.36:** *The computation to find the solution $c_1, c_2, c_3, c_4$ for the linear system above with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Square Matrix with Armadillo/main.cpp**).*

$$c_1\boldsymbol{p}_1 + c_2\boldsymbol{p}_2 + c_3\boldsymbol{p}_3 = \boldsymbol{p}$$

gives the systems

$$\begin{aligned} c_1 \qquad\qquad &= 0 \\ c_1 + c_2 \qquad &= 0 \\ c_1 + c_2 + c_3 &= 0 \end{aligned}$$

$$\begin{aligned} c_1 \qquad\qquad &= a \\ c_1 + c_2 \qquad &= b \\ c_1 + c_2 + c_3 &= c \end{aligned}$$

$$\begin{aligned} c_1 \qquad\qquad &= 7 \\ c_1 + c_2 \qquad &= -1 \\ c_1 + c_2 + c_3 &= 2 \end{aligned}$$

or in matrix form

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

check the determinant to see whether they span the vector space or not

$$\begin{vmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{vmatrix} = 1$$

since the determinant is not zero, thus $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\}$ spans $P_2$. The third system' solution is

$$c_1 = 7,\ c_2 = -8,\ c_3 = 3$$

Thus we can express $\boldsymbol{p}$ with

$$\boldsymbol{p} = 7\boldsymbol{p}_1 - 8\boldsymbol{p}_2 + 3\boldsymbol{p}_3$$

[DF*] **Example:**

The accompanying figure shows a rectangular $xy$-coordinate system determined by the unit basis vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ and an $x'y'$-coordinate system determined by unit basis vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$. Find the $x'y'$ coordinates of the points whose $xy$-coordinate are given

(a) $(\sqrt{3}, 1)$
(b) $(1, 0)$
(c) $(0, 1)$
(d) $(a, b)$

**Solution:**
From the diagram,

$$\boldsymbol{j} = \boldsymbol{u}_2$$

and

$$\boldsymbol{u}_1 = (\cos\ 30^0)\boldsymbol{i} + (\sin\ 30^0)\boldsymbol{j} = \frac{\sqrt{3}}{2}\boldsymbol{i} + \frac{1}{2}\boldsymbol{j}$$

Solving $\boldsymbol{u}_1$ for $\boldsymbol{i}$ in terms of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ gives

$$\boldsymbol{i} = \frac{2}{\sqrt{3}}\boldsymbol{u}_1 - \frac{1}{\sqrt{3}}\boldsymbol{u}_2$$

(a) $(\sqrt{3}, 1)$ is $\sqrt{3}\boldsymbol{i} + \boldsymbol{j}$

$$\begin{aligned}\sqrt{3}\boldsymbol{i} + \boldsymbol{j} &= \sqrt{3}\left(\frac{2}{\sqrt{3}}\boldsymbol{u}_1 - \frac{1}{\sqrt{3}}\boldsymbol{u}_2\right) + \boldsymbol{u}_2 \\ &= 2\boldsymbol{u}_1 - \boldsymbol{u}_2 + \boldsymbol{u}_2 \\ &= 2\boldsymbol{u}_1\end{aligned}$$

the $x'y'$ coordinates are $(2, 0)$.

(b) $(1, 0)$ is $\boldsymbol{i}$ which has $x'y'$ coordinates

$$\left(\frac{2}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$

(c) $(0,1)$ is $\boldsymbol{j} = \boldsymbol{u}_2$, the $x'y'$ coordinates are $(0,1)$.

(d) $(a,b)$ is $a\boldsymbol{i} + b\boldsymbol{j}$

$$\begin{aligned} a\boldsymbol{i} + b\boldsymbol{j} &= a\left(\frac{2}{\sqrt{3}}\boldsymbol{u}_1 - \frac{1}{\sqrt{3}}\boldsymbol{u}_2\right) + b\boldsymbol{u}_2 \\ &= \frac{2}{\sqrt{3}}a\boldsymbol{u}_1 + \left(b - \frac{a}{\sqrt{3}}\right)\boldsymbol{u}_2 \end{aligned}$$

the $x'y'$ coordinates are $\left(\frac{2}{\sqrt{3}}a, b - \frac{a}{\sqrt{3}}\right)$.

**Theorem 23.51: Number of Vectors in a Basis**

All bases for a finite-dimensional vector space have the same number of vectors.

**Theorem 23.52: Number of Vectors in a Basis 2**

Let $V$ be a finite-dimensional vector space, and let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ be any basis

(a) If a set has more than $n$ vectors, then it is linearly dependent.

(b) If a set has fewer than $n$ vectors, then it does not span $V$.

**Definition 23.35: Dimension**

The dimension of a finite-dimensional vector space $V$ is denoted by $\dim(V)$ and is defined to be the number of vectors in a basis for $V$. In addition, the zero vector space is defined to have dimension zero.

Engineers often use the term degrees of freedom as a synonym for dimension.

[DF*] Dimensions of some Vector Spaces:

$$\dim(\mathbb{R}^n) = n$$

$$\dim(P_n) = n + 1$$

$$\dim(M_{mn}) = m \times n$$

e.g. the standard basis for matrix $3 \times 5$ / $M_{35}$ has 15 vectors.

[DF*] **Example:**

Find a basis for and the dimension of the solution space of the homogeneous system

$$\begin{array}{rrrrrcl} 2x_1 & +\,2x_2 & -\,x_3 & & +\,x_5 & = & 0 \\ -x_1 & -\,x_2 & +\,2x_3 & -\,3x_4 & +\,x_5 & = & 0 \\ x_1 & +\,x_2 & -\,2x_3 & & -\,x_5 & = & 0 \\ & & x_3 & +\,x_4 & +\,x_5 & = & 0 \end{array}$$

**Solution:**

We solve this system with Gauss-Jordan elimination thus the general solution is

$$x_1 = -s - t, \quad x_2 = s, \quad x_3 = -t, \quad x_4 = 0, \quad x_5 = t$$

```
     -1      -1       2      -3
      1       1      -2       0
      0       0       1       1

Final Augmented Matrix is :
      2       2      -1       0       1       0
      0       0     1.5      -3     1.5       0
      0       0    -1.5       0    -1.5       0
      0       0       1       1       1       0
      0       0       0       0       0       0

Solutions for Matrix A :
The solution/s : Infinite Solutions Exists

root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/
mbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination
```

**Figure 23.37:** *The computation to solve underdetermined system with Gauss-Jordan elimination and C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Homogeneous System/main.cpp**).*

since the solutions are infinite then we are using parameters, which can be written in vector form as

$$(x_1, x_2, x_3, x_4, x_5) = s(-1, 1, 0, 0, 0) + t(-1, 0, -1, 0, 1)$$

this shows that the vectors

$$v_1 = \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}$$

span the solution space. Since neither vector is a scalar multiple of the other, they are linearly independent and hence form a basis for the solution space. Thus, the solution space has dimension 2. The number of parameters for the solution space defines the number of dimension.

**Theorem 23.53: Plus/Minus Theorem**

Let $S$ be a nonempty set of vectors in a vector space $V$.

(a) If $S$ is a linearly independent set, and if $v$ is a vector in $V$ that is outside of $span(S$, then the set $S \cup \{v\}$ that results by inserting $v$ into $S$ is still linearly independent.

(b) If $v$ is a vector in $S$ that is expressible as a linear combination of other vectors in $S$, and if $S - \{v\}$ denotes the set obtained by removing $v$ from $S$, then $S$ and $S - \{v\}$ span the same space; that is,

$$\text{span}(S) = \text{span}(S - \{v\})$$

[DF*] In general, to show that a set of vectors $\{v_1, v_2, \ldots, v_n\}$ is a basis for a vector space $V$, we must show that the vectors are linearly independent and span $V$. If we happen to know that $V$ has dimension $n$, so that $\{v_1, v_2, \ldots, v_n\}$ contains the right number of vectors for a basis, then it suffices to check either linear independence or spanning, the remaining condition will hold automatically.

**Theorem 23.54: Basis and Dimension**

Let $V$ be an $n$-dimensional vector space, and let $S$ be a set in $V$ with exactly $n$ vectors. Then $S$ is a basis for $V$ if and only if $S$ spans $V$ or $S$ is linearly independent.

**Theorem 23.55: Basis Rules**

Let $S$ be a finite set of vectors in a finite-dimensional vector space $V$.

(a) If $S$ spans $V$ but is not a basis for $V$, then $S$ can be reduced to a basis for $V$ by removing appropriate vectors from $S$.

(b) If $S$ is linearly independent set that is not already a basis for $V$, then $S$ can be enlarged to a basis for $V$ by inserting appropriate vectors in $S$.

**Theorem 23.56: Dimension of a Vector Space and Its Subspaces**

If $W$ is a subspace of a finite-dimensional vector space $V$, then:

(a) $W$ is finite-dimensional.

(b) $\dim(W) \leq \dim(V)$.

(c) $W = V$ if and only if $\dim(W) = \dim(V)$.

[DF*] **Example:**
Find bases for the following subspaces of $\mathbb{R}^3$.

(a) The plane $2x + 4y - 3z = 0$.

(b) The plane $y + z = 0$.

(c) The line $x = 4t,\ y = 2t,\ z = -t$.

(d) All vectors of the form $(a, b, c)$ where $c = a - b$

**Solution:**

(a) Solving the equation for $x$ gives

$$x = -2y + \frac{3}{2}z$$

so the parametric equations are

$$\begin{aligned} x &= -2r + \frac{3}{2}s \\ y &= r \\ z &= s \end{aligned}$$

which can be written in vector form as

$$\begin{aligned} (x, y, z) &= (-2r + \frac{3}{2}s, r, s) \\ &= r(-2, 1, 0) + s\left(\frac{3}{2}, 0, 1\right) \end{aligned}$$

The basis is $(-2,1,0),\left(\frac{3}{2},0,1\right)$. The reason why the basis is 2 dimension because it is a plane equation, and to span a plane we only need 2-dimensional basis.

(b) Solving the equation for $y$ gives

$$y=-z$$

so the parametric equations are

$$\begin{aligned} x &= r \\ y &= -s \\ z &= s \end{aligned}$$

which can be written in vector form as

$$\begin{aligned} (x,y,z) &= (r,-s,s) \\ &= r(1,0,0)+s(0,-1,1) \end{aligned}$$

The basis is $(1,0,0),(0,-1,1)$.

(c) In vector form, the line is

$$\begin{aligned} (x,y,z) &= (4t,2t,-t) \\ &= t(4,2,-1) \end{aligned}$$

The basis is $(4,2,-1)$.

(d) The vectors can be parametrized as $a=r, b=s, c=r-s$ or

$$\begin{aligned} (a,b,c) &= (r,s,r-s) \\ &= r(1,0,1)+s(0,1,-1) \end{aligned}$$

The basis is $(1,0,1),(0,1,-1)$.

[DF*] **Example:**

Find the dimension of each of the following vector spaces

(a) The vector space of all diagonal $n\times n$ matrices.

(b) The vector space of all symmetric $n\times n$ matrices.

(c) The vector space of all upper triangular $n\times n$ matrices.

**Solution:**

(a) The dimension is $n$, since a basis is the set $[A_1,A_2,\ldots,A_n]$ where the only nonzero entry in $A_i$ is $a_{ii}=1$.

$$A_1\begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \vdots & & \vdots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix}$$

$$A_2\begin{bmatrix} 0 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & & \vdots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix}$$

$$A_n \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

(b) In a symmetric matrix, the elements above the main diagonal determine the elements below the main diagonal, so the entries on and above the main diagonal determine all the entries. There are

$$n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$$

entries on or above the main diagonal, so the space has dimension $\frac{n(n+1)}{2}$.
For example, if we have a matrix of size $3 \times 3$ like this

$$A_{33} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The basis for symmetric matrix of size $3 \times 3$ or $A_{33}$ is 6, which means this matrix has dimension $\frac{3(3+1)}{2} = 6$, we only need to determine 6 values to fill the whole entries for this matrix, they are $a_{11}, a_{12}, a_{13}, a_{22}, a_{23}, a_{33}$, since the entry of $a_{ji}$ will be the same of $a_{ij}$ for symmetric matrix.

(c) It is almost the same like in part (b), there are $\frac{n(n+1)}{2}$ entries on or above the main diagonal, so the space has dimension $\frac{n(n+1)}{2}$. For example, for matrix of size $3 \times 3$ it will be like this

$$A_{33} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix}$$

[DF*] **Example:**

(a) Show that the set $W$ of all polynomials in $P_2$ such that $p(1) = 0$ is a subspace of $P_2$.

(b) Make a conjecture about the dimension of $W$.

(c) Confirm your conjecture by finding a basis for $W$.

**Solution:**

(a) Let $\boldsymbol{p}_1 = p_1(x)$ and $\boldsymbol{p}_2 = p_2(x)$ be elements of $W$. Then

$$(p_1 + p_2)(1) = p_1(1) + p_2(1) = 0$$

so $\boldsymbol{p}_1 + \boldsymbol{p}_2$ is in $W$, closed under addition.

Then

$$(kp_1)(1) = k(p_1(1)) = k \cdot 0 = 0$$

sp $k\boldsymbol{p}_1$ is in $W$, closed under scalar multiplication.

(b) We know that polynomial $P_2$ will have degree of 2 at most, the dimension cannot be 1 since the standard basis of $P_2$

$$\{1, x, x^2\}$$

the standard basis above create the dimension of 3.

If we input only one basis to satisfy the criteria of $p(1) = 0$ with basis of either $1, x$ or $x^2$ it will never satisfy the required statement since

$$\begin{aligned} 1 &\neq 0 \\ x &\neq 0, \quad \text{for } x = 1 \\ x^2 &\neq 0, \quad \text{for } x = 1 \end{aligned}$$

Thus, it does not span to make the statement $p(1) = 0$ true, hence we add another basis, now with 2 dimension we can have the variation of basis that can cancel one another to make the statement $p(1) = 0$ true. It is sufficient, if we add another basis to become 3 dimension, then the basis would be linear dependent just to make $p(1) = 0$ true.

(c) We need to create a basis so

$$p(1) = 0$$

It will only take basis in 2 dimension that is the linear combination of the standard basis

$$\{1, x, x^2\}$$

the options are

$$\{-1 + x, -x + x^2\}$$

or

$$\{-1 + x, -1 + x^2\}$$

both options will make $p(1) = 0$, thus they are some examples of basis for $W$.

[DF*] **Example:**
Find standard basis vectors for $\mathbb{R}^4$ that can be added to the set $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ to produce a basis for $\mathbb{R}^4$.

$$\boldsymbol{v}_1 = (1, -4, 2, -3), \quad \boldsymbol{v}_2 = (-3, 8, -4, 6)$$

**Solution:**
Let

$$\begin{aligned} \boldsymbol{u}_1 &= (1, 0, 0, 0) \\ \boldsymbol{u}_2 &= (0, 1, 0, 0) \\ \boldsymbol{u}_3 &= (0, 0, 1, 0) \\ \boldsymbol{u}_4 &= (0, 0, 0, 1) \end{aligned}$$

Then the set $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3, \boldsymbol{u}_4\}$ clearly spans $\mathbb{R}^4$.
The equation

$$c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 + k_3\boldsymbol{u}_3 + k_4\boldsymbol{u}_4 = \mathbf{0}$$

leads to the system

$$\begin{array}{rrrrrrrl} c_1 & -3c_2 & +k_1 & & & & = & 0 \\ -4c_1 & +8c_2 & & +k_2 & & & = & 0 \\ 2c_1 & -4c_2 & & & +k_3 & & = & 0 \\ -3c_1 & +6c_2 & & & & +k_4 & = & 0 \end{array}$$

The matrix

$$\begin{bmatrix} 1 & -3 & 1 & 0 & 0 & 0 & 0 \\ -4 & 8 & 0 & 1 & 0 & 0 & 0 \\ 2 & -4 & 0 & 0 & 1 & 0 & 0 \\ -3 & 6 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

reduces to

$$\begin{bmatrix} 1 & 0 & -2 & 0 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 & -\frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 & 0 & -\frac{4}{3} & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{2}{3} & 0 \end{bmatrix}$$

from the reduced matrix it is clear that $\boldsymbol{u}_1$ is a linear combination of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ so it will not be in the basis, and that $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ is linearly independent so it is one possible basis.

Similarly, $\boldsymbol{u}_4$ and either $\boldsymbol{u}_2$ or $\boldsymbol{u}_3$ will produce a linearly independent set. Thus, any two of the vectors

$$(0,1,0,0), (0,0,1,0), (0,0,0,1)$$

can be used.

```
Matrix A :
     1      -3       1       0       0       0       0
    -4       8       0       1       0       0       0
     2      -4       0       0       1       0       0
    -3       6       0       0       0       1       0

Final Augmented Matrix is :
     1       0      -2   -0.75       0       0       0
     0      -4       4       1       0       0       0
     0       0       0     0.5       1       0       0
     0       0       0   -0.75       0       1       0

Solutions for Matrix A :
The solution/s : Infinite Solutions Exists
```

**Figure 23.38:** *The computation to solve underdetermined homogeneous system with Gauss-Jordan elimination and C++. The C++ computation is correct but you have to continue by hand till the very simple form of the reduced row echelon to achieve the matrix like above. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Underdetermined Homogeneous System/main.cpp**).*

[DF*] **Example:**
The vectors $\boldsymbol{v}_1 = (1, -2, 3)$ and $\boldsymbol{v}_2 = (0, 5, -3)$ are linearly independent. Enlarge $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ to a basis for $\mathbb{R}^3$.
**Solution:**
Let $\boldsymbol{v}_3 = (a, b, c)$. The equation

$$c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 = \mathbf{0}$$

leads to the system

$$\begin{array}{rrrcl} c_1 & & + ac_3 & = & 0 \\ -2c_1 & + 5c_2 & + bc_3 & = & 0 \\ 3c_1 & - 3c_2 & + cc_3 & = & 0 \end{array}$$

In matrix form

$$\begin{bmatrix} 1 & 0 & a & 0 \\ -2 & 5 & b & 0 \\ 3 & -3 & c & 0 \end{bmatrix}$$

reduces to

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & \frac{2}{5}a + \frac{1}{5}b & 0 \\ 0 & 0 & -\frac{9}{5}a + \frac{3}{5}b + c & 0 \end{bmatrix}$$

and the homogeneous system has only the trivial solution if

$$-\frac{9}{5}a + \frac{3}{5}b + c \neq 0$$

Thus, adding any vector $v_3$ with

$$9a - 3b - 5c \neq 0$$

will create a basis for $\mathbb{R}^3$.

```
The matrix A:
[ 1  0  a  0]
[-2  5  b  0]
[ 3 -3  c  0]


A (in reduced row form) :
[      1             0             a             0      ]
[      0             5           b+2*a           0      ]
[      0             0       c-9/5*a+3/5*b       0      ]
```

**Figure 23.39:** *The computation to solve symbolic underdetermined homogeneous system with C++. (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Underdetermined Symbolic Matrix with SymbolicC++/main.cpp**).*

[DF*] A basis is the vector space generalization of a coordinate system, changing bases is akin to changing coordinate axes in $\mathbb{R}^2$ and $\mathbb{R}^3$.

[DF*] If $S = \{v_1, v_2, \ldots, v_n\}$ is a basis for a finite-dimensional vector space $V$, and if

$$(v)_S = (c_1, c_2, \ldots, c_n)$$

is the coordinate vector of $v$ relative to $S$, then the mapping

$$v \to (v)_S \tag{23.67}$$

creates a connection (a one-to-one correspondence) between vectors in the general vector space $V$ and vectors in the familiar vector space $\mathbb{R}^n$. We call $v \to (v)_S$ the coordinate map from $V$ to $\mathbb{R}^n$.

[DF*] We will find it convenient to express coordinate vectors in the matrix form

$$[v]_S = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \tag{23.68}$$

[DF*] If we change the basis for a vector space $V$ from an old basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ to a new basis $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \ldots, \boldsymbol{u}'_n\}$, then for each vector $\boldsymbol{v}$ in $V$, the old coordinate vector $[\boldsymbol{v}]_B$ is related to the new coordinate vector $[\boldsymbol{v}]_{B'}$ by the equation

$$[\boldsymbol{v}]_B = P_{B' \to B}[\boldsymbol{v}]_{B'} \tag{23.69}$$

where the columns of $P$ are the coordinate vectors of the new basis vectors relative to the old basis; that is, the column vectors of $P$ are

$$[\boldsymbol{u}'_1]_B, [\boldsymbol{u}'_2]_B, \ldots, [\boldsymbol{u}'_n]_B \tag{23.70}$$

[DF*] The matrix $P$ is called the transition matrix from $B'$ to $B$. For emphasis, we will often denote it by

$$P_{B' \to B}$$

This matrix can be expressed in terms of its column vectors as

$$P_{B' \to B} = \big[[\boldsymbol{u}'_1]_B \mid [\boldsymbol{u}'_2]_B \mid \ldots \mid [\boldsymbol{u}'_n]_B\big] \tag{23.71}$$

A column vector $[\boldsymbol{u}'_1]_B$ is a column vector $\boldsymbol{u}'_1$ represented in old bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n\}$ terms.

Similarly, the transition matrix from $B$ to $B'$ can be expressed in terms of its column vectors as

$$P_{B \to B'} = [[\boldsymbol{u}_1]_{B'} \mid [\boldsymbol{u}_2]_{B'} \mid \ldots \mid [\boldsymbol{u}_n]_{B'}] \tag{23.72}$$

A column vector $[\boldsymbol{u}_1]_{B'}$ is a column vector $\boldsymbol{u}_1$ represented in new bases $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \ldots \boldsymbol{u}'_n\}$ terms.

[DF*] Consider this old bases in $\mathbb{R}^n$

$$B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n\}$$

and the new bases in in $\mathbb{R}^n$

$$B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \ldots \boldsymbol{u}'_n\}$$

We can use transition matrix $P_{B \to B'}$ to obtain old bases from multiplying $P_{B \to B'}$ with new bases

$$\begin{aligned} P_{B \to B'}[\boldsymbol{u}'_1] &= [\boldsymbol{u}_1] \\ P_{B \to B'}[\boldsymbol{u}'_2] &= [\boldsymbol{u}_2] \\ &\vdots \\ P_{B \to B'}[\boldsymbol{u}'_n] &= [\boldsymbol{u}_n] \end{aligned}$$

in converse, we can use transition matrix $P_{B' \to B}$ to obtain new bases from multiplying $P_{B' \to B}$ with old bases

$$\begin{aligned} P_{B' \to B}[\boldsymbol{u}_1] &= [\boldsymbol{u}'_1] \\ P_{B' \to B}[\boldsymbol{u}_2] &= [\boldsymbol{u}'_2] \\ &\vdots \\ P_{B' \to B}[\boldsymbol{u}_n] &= [\boldsymbol{u}'_n] \end{aligned}$$

[DF*] The columns of the transition matrix from an old basis to a new basis are the coordinate vectors of the old basis relative to the new basis.

[DF*] Let $\boldsymbol{v}$ or $[\boldsymbol{v}]_B$ be any vector in $V$, the changing of bases will change the vector $\boldsymbol{v}$ as well. We know that vector $\boldsymbol{v}$ can be represented in old bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n\}$ as

$$\boldsymbol{v} = k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 + \cdots + k_n\boldsymbol{u}_n$$

this vector $\boldsymbol{v}$ will change when we use new bases $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \ldots \boldsymbol{u}'_n\}$ into $\boldsymbol{v}'$ or $[\boldsymbol{v}]_{B'}$

$$\boldsymbol{v}' = k'_1\boldsymbol{u}'_1 + k'_2\boldsymbol{u}'_2 + \cdots + k'_n\boldsymbol{u}'_n$$

we can write them in vector notation as

$$\boldsymbol{v} = [\boldsymbol{v}]_B = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{bmatrix}, \quad \boldsymbol{v}' = [\boldsymbol{v}]_{B'} = \begin{bmatrix} k'_1 \\ k'_2 \\ \vdots \\ k'_n \end{bmatrix}$$

We are able to determine $[\boldsymbol{v}]_B$ and $[\boldsymbol{v}]_{B'}$ when we know the new and old bases by using the transition matrix

$$[\boldsymbol{v}]_B = P_{B'\to B}[\boldsymbol{v}]_{B'} \tag{23.73}$$

$$[\boldsymbol{v}]_{B'} = P_{B\to B'}[\boldsymbol{v}]_B \tag{23.74}$$

with

$$(P_{B'\to B})(P_{B\to B'}) = I_n \tag{23.75}$$

**Computational Guide 23.1: A procedure for computing $P_{B\to B'}$**

1. Form the matrix
$$[B' \mid B]$$
with $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n\}$ as old bases and $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \ldots \boldsymbol{u}'_n\}$ as new bases. The bases will be treated as column vectors when constructing the matrix above.

2. Use elementary row operations, e.g. Gaussian elimination / Gauss-Jordan elimination to reduce the matrix to reduced row echelon form.

3. The resulting matrix will be
$$[I \mid P_{B\to B'}]$$

[DF*] **Example:**
Consider the bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2\}$ for $\mathbb{R}^2$, where

$$\boldsymbol{u}_1 = (1,0), \quad \boldsymbol{u}_2 = (0,1), \quad \boldsymbol{u}'_1 = (1,1), \quad \boldsymbol{u}'_2 = (2,1)$$

(a) Find the transition matrix $P_{B'\to B}$ from $B'$ to $B$.

(b) Find the transition matrix $P_{B\to B'}$ from $B$ to $B'$.

**Solution:**

(a) The old basis vectors are $\boldsymbol{u}'_1$ and $\boldsymbol{u}'_2$ and the new basis vectors are $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$. We want to find the coordinate matrices of the old basis vectors $\boldsymbol{u}'_1$ and $\boldsymbol{u}'_2$ relative to the new basis vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$.

To do this, we observe that

$$\begin{aligned} \boldsymbol{u}_1' &= \boldsymbol{u}_1 + \boldsymbol{u}_2 \\ \boldsymbol{u}_2' &= 2\boldsymbol{u}_1 + \boldsymbol{u}_2 \end{aligned}$$

from which it follows that

$$[\boldsymbol{u}_1']_B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad [\boldsymbol{u}_2']_B = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

and hence that

$$P_{B' \to B} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}$$

(b) The old basis vectors are $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ and the new basis vectors are $\boldsymbol{u}_1'$ and $\boldsymbol{u}_2'$. We want to find the coordinate matrices of the old basis vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ relative to the new basis vectors $\boldsymbol{u}_1'$ and $\boldsymbol{u}_2'$.

To do this, we observe that

$$\begin{aligned} \boldsymbol{u}_1 &= -\boldsymbol{u}_1' + \boldsymbol{u}_2' \\ \boldsymbol{u}_2 &= 2\boldsymbol{u}_1' - \boldsymbol{u}_2' \end{aligned}$$

from which it follows that

$$[\boldsymbol{u}_1]_{B'} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \quad [\boldsymbol{u}_2]_{B'} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

and hence that

$$P_{B \to B'} = \begin{bmatrix} -1 & 2 \\ 1 & -1 \end{bmatrix}$$

[DF*] Suppose now that $B$ and $B'$ are bases for a finite-dimensional vector space $V$. Since multiplication by $P_{B' \to B}$ maps coordinate vectors relative to the basis $B'$ into coordinate vectors relative to a basis $B$, and $P_{B \to B'}$ maps coordinate vectors relative to $B$ into coordinate vectors relative to $B'$, it follows that for every vector $\boldsymbol{v}$ in $V$ we have

$$[\boldsymbol{v}]_B = P_{B' \to B}[\boldsymbol{v}]_{B'} \tag{23.76}$$

$$[\boldsymbol{v}]_{B'} = P_{B \to B'}[\boldsymbol{v}]_B \tag{23.77}$$

**Theorem 23.57: Transition to Standard Basis for $\mathbb{R}^n$**

Let $B' = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ be any basis for the vector space $\mathbb{R}^n$ and let $S = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n\}$ be the standard basis for $\mathbb{R}^n$. If the vectors in these bases are written in column form / column vector, then

$$P_{B' \to S} = [\boldsymbol{u}_1 \mid \boldsymbol{u}_2 \mid \ldots \mid \boldsymbol{u}_n] \tag{23.78}$$

[DF*] **Example:**
Find the coordinate vector for $p$ relative to the basis $S = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\}$ for $P_2$ (we are going to use $B'$ for easier notation writing for the new basis, thus $B' = S$)

(a) $\boldsymbol{p} = 5 - x + 3x^2$; $\boldsymbol{p}_1 = 5$, $\boldsymbol{p}_2 = x$, $\boldsymbol{p}_3 = x^2$

(b) $\boldsymbol{p} = 2 + x - x^2$; $\boldsymbol{p}_1 = 1 - x$, $\boldsymbol{p}_2 = x + x^2$, $\boldsymbol{p}_3 = 2 - x^2$

**Solution:**

(a) We can refer to the standard basis for $P_2$ as $B = \{1, x, x^2\}$ and the new basis as $B' = \{5, x, x^2\}$, thus we can write $\boldsymbol{p}$ in vector column form as

$$[\boldsymbol{p}]_B = \begin{bmatrix} 5 \\ -1 \\ 3 \end{bmatrix}$$

Now, we are going to do row operations on the matrix $[B'|B]$ to obtain $P_{B\to B'}$

$$[B'|B] = \left[\begin{array}{ccc|ccc} 5 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

$$[B'|B] \quad \vec{row\ operations} \quad [I|P_{B\to B'}]$$

$$[I|P_{B\to B'}] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{1}{5} & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

thus

$$\begin{aligned} [\boldsymbol{p}]_{B'} &= P_{B\to B'}[\boldsymbol{p}]_B \\ &= \begin{bmatrix} \frac{1}{5} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ -1 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix} \end{aligned}$$

the coordinate vector for $\boldsymbol{p}$ relative to the basis $S/B'$ is $[\boldsymbol{p}]_{B'} = 1 - x + 3x^2$.

(b) We can refer to the standard basis for $P_2$ as $B = \{1, x, x^2\}$ and the new basis as $B' = \{1 - x, x + x^2, 1 - x^2\}$, thus we can write $\boldsymbol{p}$ in vector column form as

$$[\boldsymbol{p}]_B = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

Now, we are going to do row operations on the matrix $[B'|B]$ to obtain $P_{B\to B'}$

$$[B'|B] = \left[\begin{array}{ccc|ccc} 1 & 0 & 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 \end{array}\right]$$

$$[B'|B] \quad \vec{row\ operations} \quad [I|P_{B\to B'}]$$

$$[I|P_{B\to B'}] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 1 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{array}\right]$$

thus

$$
\begin{aligned}
[\boldsymbol{p}]_{B'} &= P_{B\to B'}[\boldsymbol{p}]_B \\
&= \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}
\end{aligned}
$$

the coordinate vector for $\boldsymbol{p}$ relative to the basis $S/B'$ is $[\boldsymbol{p}]_{B'} = x + 2x^2$.

```
Matrix A :
     1       0       1       1       0       0
    -1       1       0       0       1       0
     0       1      -1       0       0       1

Final Augmented Matrix is :
     1       0       0     0.5    -0.5     0.5
     0       1       0     0.5     0.5     0.5
     0       0      -2      -1      -1       1
```

**Figure 23.40:** *The row operations with Gauss-Jordan elimination with C++ to find transition matrix $P_{B\to B'}$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination to Determine Matrix Transition for Changing Basis/main.cpp**).*

[DF*] **Example:**
Consider the coordinate vectors

$$
[\boldsymbol{w}]_S = \begin{bmatrix} 6 \\ -1 \\ 4 \end{bmatrix}, \quad [B]_S = \begin{bmatrix} -8 \\ 7 \\ 6 \\ 3 \end{bmatrix}
$$

(a) Find $\boldsymbol{w}$ if the basis is $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ with $\boldsymbol{v}_1 = (1,0,0), \boldsymbol{v}_2 = (2,2,0), \boldsymbol{v}_3 = (3,3,3)$.

(b) Find $B$ if the basis is $S = \{A_1, A_2, A_3, A_4\}$ with

$$
A_1 = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}
$$

$$
A_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}
$$

**Solution:**

(a) We will use a standard basis $I = \{(1,0,0),(0,1,0),(0,0,1)\}$ and the new basis $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ to find the transition matrix.

First, we can write $\boldsymbol{w}$ in vector column form as

$$
[\boldsymbol{w}]_S = \begin{bmatrix} 6 \\ -1 \\ 4 \end{bmatrix}
$$

Now, we are going to do row operations on the matrix $[I|S]$ to obtain $P_{S\to I}$

$$[I|S] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 2 & 3 \\ 0 & 1 & 0 & 0 & 2 & 3 \\ 0 & 0 & 1 & 0 & 0 & 3 \end{array}\right]$$

$$[I|S] \quad \vec{row\ operations} \quad [I|P_{S\to I}]$$

$$[I|P_{S\to I}] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 2 & 3 \\ 0 & 1 & 0 & 0 & 2 & 3 \\ 0 & 0 & 1 & 0 & 0 & 3 \end{array}\right]$$

thus

$$\begin{aligned} [\boldsymbol{w}]_I &= P_{S\to I}[\boldsymbol{w}]_S \\ &= \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 3 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 6 \\ -1 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} 16 \\ 10 \\ 12 \end{bmatrix} \end{aligned}$$

the coordinate vector for $\boldsymbol{w}$ relative to the basis $I$ is $[\boldsymbol{w}]_I = \boldsymbol{w} = (16, 10, 12)$.

(b) We will use a standard basis $I = \{I_1, I_2, I_3, I_4\}$ and the new basis $S = \{A_1, A_2, A_3, A_4\}$ to find the transition matrix.

$$I_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad I_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$I_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad I_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Now, we are going to do row operations on the matrix $[I|S]$ to obtain $P_{S\to I}$

$$[I|S] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

$$[I|S] \quad \vec{row\ operations} \quad [I|P_{S\to I}]$$

$$[I|P_{S\to I}] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

thus

$$
\begin{aligned}
[B]_I &= P_{S\to I}[B]_S \\
&= \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -8 \\ 7 \\ 6 \\ 3 \end{bmatrix} \\
&= \begin{bmatrix} 15 \\ -1 \\ 6 \\ 3 \end{bmatrix}
\end{aligned}
$$

the coordinate vector for $[B]$ relative to the basis $I$ is $[B]_I = [B] = (15, -1, 6, 3)$.

[DF*] **Example:**

Let $V$ be the space spanned by $f_1 = \sin\ x$ and $f_2 = \cos\ x$.

(a) Show that $g_1 = 2\ \sin\ x + \cos\ x$ and $g_2 = 3\ \cos\ x$ form a basis for $V$.

(b) Find the transition matrix from $B' = \{g_1, g_2\}$ to $B = \{f_1, f_2\}$.

(c) Find the transition matrix from $B$ to $B'$.

(d) Compute the coordinate vector $[\boldsymbol{h}]_B$, where $\boldsymbol{h} = 2\ \sin\ x - 5\ \cos\ x$, and then obtain $[\boldsymbol{h}]_{B'}$

**Solution:**

(a) We will have the standard basis $B$ that can be written in matrix form as

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The matrix above can be interpreted like this: the first row is to represent $\sin\ x$, the second row is to represent $\cos\ x$, the first column with Navy Blue color is to represent $f_1 = \sin\ x$, while the second column with Cyan color is to represent $f_2 = \cos\ x$.

The span of $f_1$ and $f_2$ is the set of all linear combinations

$$af_1 + bf_2 = a\ \sin\ x + b\ \cos\ x$$

and this vector can be represented by $(a, b)$.

Since $g_1 = 2f_1 + f_2$ and $g_2 = 3f_2$, it is sufficient to compute

$$\det \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix} = 6$$

Since this determinant is nonzero, $g_1$ and $g_2$ form a basis for $V$.

(b) Since $B$ can be represented as $\{(1,0), (0,1)\}$

$$[B|B'] = \left[\begin{array}{cc|cc} 1 & 0 & 2 & 0 \\ 0 & 1 & 1 & 3 \end{array}\right]$$

$$[B|B'] \quad \overrightarrow{row\ operations} \quad [I|P_{B'\to B}]$$

$$[I|P_{B'\to B}] = \left[\begin{array}{cc|cc} 1 & 0 & 2 & 0 \\ 0 & 1 & 1 & 3 \end{array}\right]$$

(c) The transition matrix from $B$ to $B'$ is the inverse of $P_{B'\to B}$, thus

$$[B'|B] = \left[\begin{array}{cc|cc} 2 & 0 & 1 & 0 \\ 1 & 3 & 0 & 1 \end{array}\right]$$

$$[B'|B] \quad \overrightarrow{row\ operations} \quad [I|P_{B\to B'}]$$

$$[I|P_{B\to B'}] = \left[\begin{array}{cc|cc} 1 & 0 & \frac{1}{2} & 0 \\ 0 & 1 & -\frac{1}{6} & \frac{1}{3} \end{array}\right]$$

so

$$P_{B\to B'} = \begin{bmatrix} \frac{1}{2} & 0 \\ -\frac{1}{6} & \frac{1}{3} \end{bmatrix}$$

(d) Since $B = \{(\sin\ x), (\cos\ x)\}$ is the standard basis for $V$. Thus $\boldsymbol{h} = 2\ \sin\ x - 5\ \cos\ x$ can be written in column vector form as

$$[\boldsymbol{h}]_B = \begin{bmatrix} 2 \\ -5 \end{bmatrix}$$

thus

$$\begin{aligned} [\boldsymbol{h}]_{B'} &= P_{B\to B'}[\boldsymbol{h}]_B \\ &= \begin{bmatrix} \frac{1}{2} & 0 \\ -\frac{1}{6} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 2 \\ -5 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -2 \end{bmatrix} \end{aligned}$$

[DF*] **Example:**
Let $S = \{\boldsymbol{e}_1, \boldsymbol{e}_2\}$ be the standard basis for $\mathbb{R}^2$, and let $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ be the basis that results when the vectors in $A$ are reflected about the line that makes an angle $\theta$ with the positive $x$-axis. Find the transition matrix $P_{B\to S}$.
**Solution:**
From the diagram, it is clear that $\boldsymbol{v}_1$ will be located in the first quadrant in $\mathbb{R}^2$, thus

$$[\boldsymbol{v}_1]_S = (\cos 2\theta, \sin 2\theta)$$

Now, we have to be careful and recognize that $\boldsymbol{v}_2$ is located below the $x$-axis or in the fourth quadrant. The angle between the positive $y$-axis and $\boldsymbol{v}_2$ is

$$2\left(\frac{\pi}{2} - \theta\right) = \pi - 2\theta$$

because the angle between $\boldsymbol{e}_2$ and the line will be the same with the angle between $\boldsymbol{v}_2$ and the line.
Thus the angle between $\boldsymbol{v}_2$ and positive $x$-axis is

$$\left((\pi - 2\theta) - \frac{\pi}{2}\right) = \frac{\pi}{2} - 2\theta$$

we are using the positive angle, so the formula will be

$$\begin{aligned} [\boldsymbol{v}_2]_S &= \left(\cos\left(\frac{\pi}{2} - 2\theta\right), -\sin\left(\frac{\pi}{2} - 2\theta\right)\right) \\ &= (\sin 2\theta, -\cos 2\theta) \end{aligned}$$

**Figure 23.41:** *The diagram of* $\boldsymbol{e}_1 = (1, 0)$ *reflected about the line that makes an angle* $\theta$ *with the positive x-axis to become* $\boldsymbol{v}_1$.

**Figure 23.42:** *The diagram of* $\boldsymbol{e}_2 = (0, 1)$ *reflected about the line that makes an angle* $\theta$ *with the positive x-axis to become* $\boldsymbol{v}_2$.

**Figure 23.43:** *The plot and computation to determine $v_1$ and $v_2$, the angle used in this code is $\theta = 35^0$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Changing Basis That Are Reflected About a Line/main.cpp**).*

and the transition matrix is

$$P_{B\to S} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix} = \begin{bmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{bmatrix}$$

[DF*] Changing basis allows you to convert a matrix from a complicated form to a simple form. It is often possible to represent a matrix in a basis where the only nonzero elements are on the diagonal, which is exceptionally simple. These diagonal elements will be the eigenvalues of the matrix. This is especially helpful in solving linear systems of differential equations. Often in physics, engineering, logistics, and probably lots of other places, you have a system of differential equations which all depend on each other. In order to solve the system directly, you would have to solve all equations at once, which is hard. We can use matrices to describe this system.

By changing basis, you may be able to make that matrix diagonal, which effectively separates the differential equations from each other, so you can solve just one at a time. This is comparatively easy. Situations this comes up are

(a) Quantum mechanics, solving the Schrodinger equation to describe the state of matter on the quantum level.

(b) Electrical engineering, understanding the time-evolution of an electrical circuit.

(c) Mechanical engineering, understanding the motion of a linear mechanical system, such as multiple spring-mass system.

(d) Solid mechanics. Computing the principal stresses/strains in a material involves a change of basis. With the principal stresses, we can compute the maximum normal stress and maximum shear stress the material experiences. And these maxima tell us whether the material will become damaged or not.

Changing basis allows you to convert a matrix from a complicated form to a simple form. Rotation is essentially a change of basis. If you ever employed a linear coordinate change, you changed bases. Let say you have a cube that is rotated arbitrarily and is not orthogonal the coordinate axes, and you're trying to determine if a point is inside of it. It is a bit tricky to determine directly. If you can change to a basis where the cube is orthogonal to the coordinate axes, determining if the point is inside becomes trivial.

[DF*] Row space, column space, and null space are some important vector spaces that are associated with matrices. Understanding them will make us know the relationships between the solutions of a linear system and properties of its coefficient matrix.

**Definition 23.36: Row and Column Vectors**

For an $m \times n$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

the vectors

$$\begin{aligned} \boldsymbol{r}_1 &= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \end{bmatrix} \\ \boldsymbol{r}_2 &= \begin{bmatrix} a_{21} & a_{22} & \dots & a_{2n} \end{bmatrix} \\ &\vdots \\ \boldsymbol{r}_m &= \begin{bmatrix} a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{aligned}$$

in $\mathbb{R}^n$ that are formed from the rows of $A$ are called the row vectors of $A$, and the vectors

$$\boldsymbol{c}_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}, \quad \boldsymbol{c}_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}, \quad \boldsymbol{c}_n = \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

in $\mathbb{R}^m$ formed from the columns of $A$ are called the column vectors of $A$.

**Definition 23.37: Row Space and Column Space**

If $A$ is an $m \times n$ matrix, then the subspace of $\mathbb{R}^n$ spanned by the row vectors of $A$ is called the row space of $A$.

The subspace of $\mathbb{R}^m$ spanned by the column vectors of $A$ is called the column space of $A$.

The solution space of the homogeneous system of equations

$$A\boldsymbol{x} = \boldsymbol{0}$$

which is a subspace of $\mathbb{R}^n$, is called the null space of $A$.

[DF*] Suppose that

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

If $\boldsymbol{c}_1, \boldsymbol{c}_2, \dots, \boldsymbol{c}_n$ denote the column vectors of $A$, then the product $A\boldsymbol{x}$ can be expressed as a linear combination of these vectors with coefficients from $\boldsymbol{x}$

$$A\boldsymbol{x} = x_1\boldsymbol{c}_1 + x_2\boldsymbol{c}_2 + \cdots + x_n\boldsymbol{c}_n \tag{23.79}$$

Thus, a linear system, $A\boldsymbol{x} = \boldsymbol{b}$, of $m$ equations in $n$ unknowns can be written as

$$x_1\boldsymbol{c}_1 + x_2\boldsymbol{c}_2 + \cdots + x_n\boldsymbol{c}_n = \boldsymbol{b} \tag{23.80}$$

from which we conclude that $A\boldsymbol{x} = \boldsymbol{b}$ is consistent if and only if $\boldsymbol{b}$ is expressible as a linear combination of the column vectors of $A$.

**Theorem 23.58: Column Space and Consistent Linear System**

A system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ is consistent if and only if $\boldsymbol{b}$ is in the column space of $A$.

**Theorem 23.59: Null Space and Consistent Linear System**

If $\boldsymbol{x}_0$ is any solution of a consistent linear system $A\boldsymbol{x} = \boldsymbol{b}$, and if $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ is a basis for the null space of $A$, then every solution of $A\boldsymbol{x} = \boldsymbol{b}$ can be expressed in the form

$$\boldsymbol{x} = \boldsymbol{x}_0 + c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k \tag{23.81}$$

Conversely, for all choices of scalars $c_1, c_2, \ldots, c_k$, the vector $\boldsymbol{x}$ in this formula is a solution of $A\boldsymbol{x} = \boldsymbol{b}$.

[DF*] The equation

$$\boldsymbol{x} = \boldsymbol{x}_0 + c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k$$

gives a formula for the general solution of $A\boldsymbol{x} = \boldsymbol{b}$.

The vector $\boldsymbol{x}_0$ is called a particular solution of $A\boldsymbol{x} = \boldsymbol{b}$.

The remaining part of the formula $c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k$ is called the general solution of $A\boldsymbol{x} = \boldsymbol{0}$.

The general solution of a consistent linear system can be expressed as the sum of a particular solution of that system and the general solution of the corresponding homogeneous system.

**Theorem 23.60: Elementary Row Operations for Null Space, Row Space, and Column Space**

Elementary row operations do not change the null space of a matrix.

Elementary row operations do not change the row space of a matrix.

Elementary row operations change the column space of a matrix.

**Theorem 23.61: Bases for the Row and Column Spaces**

If a matrix $R$ is in row echelon form, then the row vectors with the leading 1's (the nonzero row vectors) form a basis for the row space of $R$, and the column vectors with the leading 1's of the row vectors form a basis for the column space of $R$.

**Theorem 23.62: Row Equivalent Matrices and Column Vectors**

If $A$ and $B$ are row equivalent matrices, then:

(a) A given set of column vectors of $A$ is linearly independent if and only if the corresponding column vectors of $B$ are linearly independent.

(b) A given set of column vectors of $A$ forms a basis for the column space of $A$ if and only if the corresponding column vectors of $B$ form a basis for the column space of $B$.

[DF*] Vector spaces are used extensively in electrical engineering, particularly in the analysis and design of electrical systems. Here are some examples of applications of vector spaces in electrical engineering:

(a) Analysis of electric circuits: In electric circuits, currents and voltages can be represented as vectors in a vector space. By applying the principles of linear algebra to these vectors, it's possible to analyze the behavior of the circuit and determine the values of circuit variables such as voltage, current, and power.

(b) Design of control systems: Control systems are used to regulate and stabilize the behavior of complex electrical systems. Vector spaces can be used to represent the state of a control system, and linear algebra can be used to design control algorithms that stabilize the system.

(c) Signal processing: In signal processing, signals such as audio, video, and data can be represented as vectors in a vector space. By applying techniques from linear algebra, it's possible to analyze and process these signals, for example, by applying filters or compressing data.

(d) Electromagnetics: Vector spaces are used to describe the behavior of electromagnetic fields, which are represented as vector fields. Maxwell's equations, which govern the behavior of electromagnetic fields, are expressed using vector calculus.

These are just a few examples of the many applications of vector spaces in electrical engineering. Vector spaces provide a powerful mathematical framework for analyzing and designing electrical systems, and are an essential tool for any electrical engineer.

[DF*] **Example:**
Let

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(a) Show that relative to an $xyz$-coordinate system in 3-space the null space of $A$ consists of all points on the $z$-axis and that the column space consists of all points in the $xy$-plane.

(b) Find a $3 \times 3$ matrix whose null space is the $x$-axis and whose column space is the $yz$-plane.

**Solution:**

(a) The row echelon form of $A$ is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

so the general solution of the system $A\boldsymbol{x} = \mathbf{0}$ is

$$x = 0,\ y = 0,\ z = t$$

or

$$t\begin{bmatrix}0\\0\\1\end{bmatrix}$$

Thus the null space of $A$ is the $z$-axis, and the column space is the span of

$$\boldsymbol{c}_1 = \begin{bmatrix}0\\1\\0\end{bmatrix},\quad \boldsymbol{c}_2 = \begin{bmatrix}1\\0\\0\end{bmatrix}$$

which is all linear combinations of $y$ and $x$, the $xy$-plane.

(b) A $3 \times 3$ matrix that has null space as the $x$-axis and column space the $yz$-plane is

$$A = \begin{bmatrix}0 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

We can see that the matrix above will have general solution for $A\boldsymbol{x} = \mathbf{0}$ as

$$x = t,\ y = 0,\ z = 0$$

or in vector term

$$t\begin{bmatrix}1\\0\\0\end{bmatrix}$$

The column space that will span only the $yz$-plane is the span of

$$\boldsymbol{c}_1 = \begin{bmatrix}0\\1\\0\end{bmatrix},\quad \boldsymbol{c}_2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

[DF*] **Example:**
Find a $3 \times 3$ matrix whose null space is

(a) a point.

(b) a line.

(c) a plane.

**Solution:**

(a) A $3 \times 3$ matrix whose null space is a point is

$$A = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

the point is $(0, 0, 0)$.

(b) A $3 \times 3$ matrix whose null space is a line is

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

the general solution for $A\boldsymbol{x} = \mathbf{0}$ is

$$x_1 = -x_2,\ x_2 = t,\ x_3 = 0$$

or in vector form

$$t \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$$

it is a line that is passing the origin $(0, 0, 0)$ and $(-1, 1, 0)$ and this line is located in the $xy$-plane.

(c) A $3 \times 3$ matrix whose null space is a plane is

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

the general solution for $A\boldsymbol{x} = \mathbf{0}$ is

$$x_1 = s,\ x_2 = t,\ x_3 = 0$$

or in vector form

$$\boldsymbol{x} = s \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

thus the null space is the span of

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

which is all linear combination of $x$ and $y$, the $xy$-plane.

[DF*] **Example:**
Find all $2 \times 2$ matrices whose null space is the line

$$x = 2t,\ y = -3t$$

**Solution:**
The matrices $2 \times 2$ whose null space is the line $x = 2t,\ y = -3t$ are

$$A = \begin{bmatrix} -\frac{1}{2} & -\frac{1}{3} \\ 0 & 0 \end{bmatrix}$$

By inspection for the first matrix we can see that the general solution of $A\boldsymbol{x} = \mathbf{0}$ or the null space needs a parameterization of

$$-\frac{1}{3}y = t$$

thus

$$y = -3t$$

$$\begin{aligned} -\frac{1}{2}x - \frac{1}{3}y &= 0 \\ -\frac{1}{2}x &= \frac{1}{3}y \\ x &= -2\left(\frac{1}{3}y\right) \\ x &= -2\left(\frac{1}{3}(-3t)\right) \\ x &= 2t \end{aligned}$$

**Theorem 23.63: Dimension of Row Space and Column Space**

The row space and column space of a matrix $A$ have the same dimension.

[DF*] The number of nonzero rows of a reduced row echelon form matrix $R$ is the same as the number of leading 1's in matrix $R$.

**Definition 23.38: Rank and Nullity**

The common dimension of the row space and column space of a matrix $A$ is called the rank of $A$ and is denoted by $\text{rank}(A)$.

The dimension of the null space of $A$ is called the nullity of $A$ and is denoted by $\text{nullity}(A)$.

[DF*] Since the rank of a matrix $A$ of size $m \times n$ is the common dimension of its row and column space, it follows that the rank is at most the smaller of $m$ and $n$.

$$\text{rank}(A) \leq \min(m, n)$$

**Theorem 23.64: Dimension Theorem for Matrices**

If $A$ is a matrix with $n$ columns, then

$$\text{rank}(A) + \text{nullity}(A) = n$$

[DF*] The number of leading variables is the same as the number of leading 1's in the reduced row echelon form of matrix $A$, which is the rank of $A$.

The number of free variables is the same as the number of parameters in the general solution of

$$A\boldsymbol{x} = \mathbf{0}$$

which is the nullity of $A$.

**Theorem 23.65: Rank and Nullity**

If $A$ is an $m \times n$ matrix, then

(a) $\text{rank}(A)$ = the number of leading variables in the general solution of $A\boldsymbol{x} = \mathbf{0}$.

(b) $\text{nullity}(A)$ = the number of parameters in the general solution of $A\boldsymbol{x} = \mathbf{0}$.

[DF*] In many applications the equations in a linear system correspond to physical constraints or conditions that must be satisfied. In general, the most desirable systems are those that have the same number of constraints as unknowns, since such systems often have a unique solution.

Unfortunately, it is not always possible to match the number of constraints and unknowns.

When linear systems have more constraints than unknowns like this

$$\begin{array}{rrrl} x_1 & +\,x_2 & +\,x_3 & = b_1 \\ x_1 & +\,2x_2 & +\,5x_3 & = b_2 \\ -x_1 & +\,6x_2 & +\,7x_3 & = b_3 \\ -x_1 & -\,3x_2 & +\,x_3 & = b_4 \end{array}$$

it is called overdetermined systems.

When linear systems have fewer constraints than unknowns like this

$$\begin{array}{rrrl} x_1 & +\,x_2 & +\,x_3 & = b_1 \\ x_1 & +\,2x_2 & +\,5x_3 & = b_2 \end{array}$$

it is called underdetermined systems.

**Theorem 23.66: Consistent Linear System**

If $A\boldsymbol{x} = \boldsymbol{b}$ is a consistent linear system of $m$ equations in $n$ unknowns, and if $A$ has rank $r$, then the general solution of the system contains $n - r$ parameters.

**Theorem 23.67: Overdetermined and Underdetermined Systems**

Let $A$ be an $m \times n$ matrix.

(a) **Overdetermined Case**
If $m > n$, then the linear system $A\boldsymbol{x} = \boldsymbol{b}$ is inconsistent for at least one vector $\boldsymbol{b}$ in $\mathbb{R}^n$.

(b) **Underdetermined Case**
If $m < n$, then for each vector $\boldsymbol{b}$ in $\mathbb{R}^n$ the linear system $A\boldsymbol{x} = \boldsymbol{b}$ is either inconsistent or has infinitely many solutions.

[DF*] There are six important vector spaces associated with a matrix $A$ and its transpose $A^T$:

1. Row space of $A$

2. Column space of $A$
3. Null space of $A$
4. Row space of $A^T$
5. Column space of $A^T$
6. Null space of $A^T$

Transposing a matrix converts row vectors into clumn vectors and conversely, so except for a difference in notation, the row space of $A^T$ is the same as the column space of $A$, and the column space of $A^T$ is the same as the row space of $A$.

Out of the six spaces listed above, only the following four are distinct:

1. Row space of $A$
2. Column space of $A$
3. Null space of $A$
4. Null space of $A^T$

These are called the fundamental spaces of a matrix $A$.

**Theorem 23.68: Rank for Matrix and Its Transpose**

If $A$ is any matrix, then

$$\text{rank}(A) = \text{rank}(A^T)$$

[DF*] If $A$ is an $m \times n$ matrix, then

$$\text{rank}(A) + \text{nullity}(A) = m \tag{23.82}$$

It is possible to express the dimensions of all four fundamental spaces in terms of size and rank of $A$.

$$\begin{aligned} \dim[\text{row}(A)] &= r \\ \dim[\text{col}(A)] &= r \\ \dim[\text{null}(A)] &= n - r \\ \dim[\text{null}(A^T)] &= m - r \end{aligned} \tag{23.83}$$

The four formulas above provide an algebraic relationship between the size of a matrix and the dimension of its fundamental spaces.

[DF*] If $A$ is an $m \times n$ matrix, then the null space of $A$ consists of those vectors that are orthogonal to each of the row vectors of $A$.

**Definition 23.39: Orthogonal Complement**

If $W$ is a subspace of $\mathbb{R}^n$, then the set of all vectors in $\mathbb{R}^n$ that are orthogonal to every vector in $W$ is called the orthogonal complement of $W$ and is denoted by the symbol $W^\perp$.

**Theorem 23.69: Orthogonal Complement Basic**

If $W$ is a subspace of $\mathbb{R}^n$, then:

(a) $W^\perp$ is a subspace of $\mathbb{R}^n$.

(b) The only vector common to $W$ and $W^\perp$ is $\mathbf{0}$.

(c) The orthogonal complement of $W^\perp$ is $W$.

**Theorem 23.70: Orthogonal Complement for Matrix**

If $A$ is an $m \times n$ matrix, then:

(a) The null space of $A$ and the row space of $A$ are orthogonal complement in $\mathbb{R}^n$.

(b) The null space of $A^T$ and the column space of $A$ are orthogonal complements in $\mathbb{R}^m$.

**Theorem 23.71: Equivalent Statements for Square Matrix**

If $A$ is an $n \times n$ matrix, then the following statements are equivalent

(a) $A$ is invertible.

(b) $A\boldsymbol{x} = \mathbf{0}$ has only the trivial solution.

(c) The reduced row echelon form of $A$ is $I_n$.

(d) $A$ is expressible as a product of elementary matrices.

(e) $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for every $n \times 1$ matrix $\boldsymbol{b}$.

(f) $A\boldsymbol{x} = \boldsymbol{b}$ has exactly one solution for every $n \times 1$ matrix $\boldsymbol{b}$.

(g) $\det(A) \neq 0$.

(h) The column vectors of $A$ are linearly independent.

(i) The row vectors of $A$ are linearly independent.

(j) The column vectors of $A$ span $\mathbb{R}^n$.

(k) The row vectors of $A$ span $\mathbb{R}^n$.

(l) The column vectors of $A$ form a basis for $\mathbb{R}^n$.

(m) The row vectors of $A$ form a basis for $\mathbb{R}^n$.

(n) $A$ has rank $n$.

(o) $A$ has nullity 0.

(p) The orthogonal complement of the null space of $A$ is $\mathbb{R}^n$.

(q) The orthogonal complement of the row space of $A$ is $\{\mathbf{0}\}$.

[DF*] Digital data are commonly stored in matrix form, and many techniques for improving transmission speed use the rank of a matrix. Rank plays a role to measure the "redundancy" in a matrix in the sense that if $A$ is an $m \times n$ matrix of rank $k$, then $n - k$ of the column vectors and $m - k$ of the row vectors can be expressed in terms of $k$ linearly independent column or row vectors. The essential idea in many data compression schemes is to approximate the original data set by a data set with smaller rank that conveys nearly the same information, then eliminate redundant vectors in the approximating set.

[DF*] **Example:**
Suppose that $A$ is a $3 \times 3$ matrix whose null space is a line through the origin in 3-space. Can the row or column space of $A$ also be a line through the origin?
**Solution:**
When a null space is a line through the origin in 3-space, an example of an equation of such line can be represented as (but not limited to)

$$z = f(x, y) = 2x$$

the function $f(x, y)$ is a line through the origin in 3-space and it is not dependent on $y$.

The reduced row echelon form of matrix $A$ that contain the null space as a line with equation of $z = 2x$ is

$$\begin{bmatrix} 1 & 0 & -\frac{1}{2} \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

hence

$$\begin{aligned} x_1 &= \frac{1}{2}x_3 \\ x_2 &= 0 \\ x_3 &= r \end{aligned}$$

the basis for the null space of $A$ is the vector

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = r \begin{bmatrix} \frac{1}{2} \\ 0 \\ 1 \end{bmatrix}$$

with $r$ as parameter. This confirms that the line is indeed $z = f(x, y) = 2x$ in vector term.

The row space for $A$ is the span of

$$\boldsymbol{r}_1 = \begin{bmatrix} 1 \\ 0 \\ -\frac{1}{2} \end{bmatrix}, \quad \boldsymbol{r}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

thus the row space will span a plane with the vectors $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, draw these two vectors in 3-dimension and you can see that the linear combinations of these two vectors will create a plane in 3-dimensional.

The same case happens for column space for $A$

$$\boldsymbol{c}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{c}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The column space will span a plane with vectors $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, it is the $xy$ plane that become the column space for $A$.

It is proven then that both row space and column space of $A$ with size of $3 \times 3$ with nullity of 1 will be a plane in 3-dimensional space. The row space and column space of $A$ will never be a line through the origin if the null space is a line.
In short, always remember that for matrix $A$ with size $m \times n$

$$\text{rank}(A) + \text{nullity}(A) = n$$

Thus in this case, we have matrix $A$ with size of $3 \times 3$ and we will have

$$\begin{aligned} \text{nullity}(A) &= 1 \\ \text{dimension}(A) &= n = 3 \\ \text{rank}(A) &= 2 \end{aligned}$$

When the nullity$(A)$ is 1, then the row space and column space / rank$(A)$ will be 2. The number of nullity represents how many basis vector for the null space, the same for rank it represents how many basis vector for the row space and column space. A line have 1 basis only, 1 vector. A plane have 2 basis, 2 vectors will create a 3-dimension have 3 basis / 3 vectors that will create a 3-dimensional space like standard basis $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$.

$$KaingblaKaingHastaLaMeo$$

$$Prappreprappreprap...prappreprappreprap...$$

[DF*] Functions of the form

$$\boldsymbol{w} = F(\boldsymbol{x})$$

has independent variable $\boldsymbol{x}$ as a vector in $\mathbb{R}^n$ and the dependent variable $\boldsymbol{w}$ as a vector in $\mathbb{R}^m$. The special class of such functions are called matrix transformations. Such transformations are fundamental in the study of linear algebra and have important application in physics, computer graphics, engineering, social sciences, and various branches of mathematics.

**Definition 23.40: Transformation**

If $V$ and $W$ are vector spaces, and if $f$ is a function with domain $V$ and codomain $W$, then we say that $f$ is a transformation from $V$ to $W$ or that $f$ maps $V$ to $W$, which we denote by writing

$$f : V \to W$$

In the special case where $V = W$, the transformation is also called an operator on $V$.

[DF*] Suppose that $f_1, f_2, \ldots, f_m$ are real-valued functions of $n$ variables

$$\begin{aligned} w_1 &= f_1(x_1, x_2, \ldots, x_n) \\ w_2 &= f_2(x_1, x_2, \ldots, x_n) \\ &\vdots \\ w_m &= f_m(x_1, x_2, \ldots, x_n) \end{aligned} \tag{23.84}$$

These $m$ equations assign a unique point $(w_1, w_2, \ldots, w_m)$ in $\mathbb{R}^m$ to each point $(x_1, x_2, \ldots, x_n)$ in $\mathbb{R}^n$ and thus define a transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$.

If we denote this transformation by $T$, then

$$T : \mathbb{R}^n \to \mathbb{R}^m$$

and

$$T(x_1, x_2, \ldots, x_n) = (w_1, w_2, \ldots, w_m)$$

[DF*] In the special case where the equations are linear and can be expressed in the form

$$\begin{aligned} w_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ w_2 &= a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ &\vdots \\ w_m &= a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{aligned} \tag{23.85}$$

which we can write in matrix notation as

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{23.86}$$

or more briefly as

$$\boldsymbol{w} = A\boldsymbol{x} \tag{23.87}$$

Although we could view this as a linear system, we will view it instead as a transformation that maps the column vector $\boldsymbol{x}$ in $\mathbb{R}^n$ into the column vector $\boldsymbol{w}$ in $\mathbb{R}^m$.

We call this a matrix transformation (or matrix operator if $m = n$), and we denote it by

$$T_A : \mathbb{R}^n \to \mathbb{R}^m$$

thus

$$\boldsymbol{w} = T_A(\boldsymbol{x})$$

The matrix transformation $T_A$ is called multiplication by $A$, and the matrix $A$ is called the standard matrix for the transformation.

On occasion, we also use the notation

$$\boldsymbol{x}\vec{T_A}\boldsymbol{w} \tag{23.88}$$

which is read "$T_A$ maps $\boldsymbol{x}$ into $\boldsymbol{w}$."

**Theorem 23.72: Properties of Matrix Transformations**

For every matrix $A$ the amtrix transformation $T_A : \mathbb{R}^n \to \mathbb{R}^m$ has the following properties for all vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ and for every scalar $k$:

(a) $T_A(\mathbf{0}) = \mathbf{0}$

(b) $T_A(k\boldsymbol{u}) = kT_A(\boldsymbol{u})$ [Homogeneity property]

(c) $T_A(\boldsymbol{u} + \boldsymbol{v}) = T_A(\boldsymbol{u}) + T_A(\boldsymbol{v})$ [Additivity property]

(d) $T_A(\boldsymbol{u} - \boldsymbol{v}) = T_A(\boldsymbol{u}) - T_A(\boldsymbol{v})$

[DF*] A matrix transformation maps linear combinations of vectors in $\mathbb{R}^n$ into the corresponding linear combinations in $\mathbb{R}^m$

$$T_A(k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 + \cdots + k_r\boldsymbol{u}_r) = k_1T_A(\boldsymbol{u}_1) + k_2T_A(\boldsymbol{u}_2) + \cdots + k_rT_A(\boldsymbol{u}_r) \tag{23.89}$$

Depending on whether $n$-tuples and $m$-tuples are regarded as vectors or points, the geometric effect of a matrix transformation

$$T_A : \mathbb{R}^n \to \mathbb{R}^m$$

is to map each vector (point) in $\mathbb{R}^n$ into a vector (point) in $\mathbb{R}^m$.

**Theorem 23.73: Same Matrix Transformations**

If $T_A : \mathbb{R}^n \to \mathbb{R}^m$ and $T_B : \mathbb{R}^n \to \mathbb{R}^m$ are matrix transformations, and if

$$T_A(\boldsymbol{x}) = T_B(\boldsymbol{x})$$

for every vector $\boldsymbol{x}$ in $\mathbb{R}^n$, then $A = B$.

[DF*] For any matrix $A$ with size of $m \times n$ and standard basis vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$ for $\mathbb{R}^n$

$$A\boldsymbol{e}_j, \quad j = 1, 2, \ldots, n$$

will result as the $j$th column of $A$ since every entry of $\boldsymbol{e}_j$ is 0 except for the $j$th, which is 1.

[DF*] If 0 is the $m \times n$ matrix, then

$$T_0(\boldsymbol{x}) = 0\boldsymbol{x} = \boldsymbol{0}$$

so multiplication by zero maps every vector in $\mathbb{R}^n$ into the zero vector in $\mathbb{R}^m$. We call $T_0$ the zero transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$.

[DF*] If $I$ is the $n \times n$ identity matrix, then

$$T_I(\boldsymbol{x}) = I\boldsymbol{x} = \boldsymbol{x}$$

so multiplication by $I$ maps every vector in $\mathbb{R}^n$ into itself. We call $T_I$ the identity operator on $\mathbb{R}^n$.

[DF*] To find the standard matrix for a matrix transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ by considering the effect of that transformation on the standard basis vectors for $\mathbb{R}^n$.

Suppose that $A$ is unknown and that

$$\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$$

are the standard basis vectors for $\mathbb{R}^n$. Suppose also that the images of these vectors under the transformation $T_A$ are

$$T_A(\boldsymbol{e}_1) = A\boldsymbol{e}_1, \quad T_A(\boldsymbol{e}_2) = A\boldsymbol{e}_2, \ldots, \quad T_A(\boldsymbol{e}_n) = A\boldsymbol{e}_n$$

$A\boldsymbol{e}_j$ is a linear combination of the columns of $A$ in which the successive coefficients are the entries of $\boldsymbol{e}_j$, so the product of $A\boldsymbol{e}_j$ is just the $j$th column of the matrix $A$. Thus,

$$A = [T_A(\boldsymbol{e}_1) \mid T_A(\boldsymbol{e}_2) \mid \ldots \mid T_A(\boldsymbol{e}_n)] \tag{23.90}$$

is the standard matrix for a matrix transformation.

[DF*] **Reflection Operators**
Some of the most basic amtrix operators on $\mathbb{R}^2$ and $\mathbb{R}^3$ are those that map each point into its symmetric image about a fixed line or a fixed plane; these are called reflection operators.

[DF*] **Projection Operators**
Matrix operators on $\mathbb{R}^2$ and $\mathbb{R}^3$ that map each point into its orthogonal projection on a fixed line or plane are called projection operators, or more precisely, orthogonal projection operators.

[DF*] **Rotation Operators**
Matrix operators on $\mathbb{R}^2$ and $\mathbb{R}^3$ that move points along circular arcs called rotation operators.

[DF*] **Dilations and Contractions**
If $k$ is a nonnegative scalar, then the operator

$$T(\boldsymbol{x}) = k\boldsymbol{x}$$

on $\mathbb{R}^2$ or $\mathbb{R}^3$ has the effect of increasing or decreasing the length of each vector by a factor of $k$.

1. If $0 \leq k \leq 1$ the operator is called a contraction with factor $k$.
2. If $k > 1$ it is called a dilation with factor $k$.
3. If $k = 1$, then $T$ is the identity operator and can be regarded either as a contraction or a dilation.

[DF*] **Expansion and Compressions**
In a dilation or contraction of $\mathbb{R}^2$ or $\mathbb{R}^3$, all coordinates are multiplied by a factor $k$. If only one of the coordinates is multiplied by $k$, then the resulting operator is called an expansion or compression.

[DF*] **Shears**
A matrix operator of the form $T(x, y) = (x + ky, y)$ translates a point $(x, y)$ in the $xy$-plane parallel to the $x$-axis by an amount $ky$ that is proportional to the $y$-coordinate of the point.

[DF*] Now, when we see a square matrix of size $2 \times 2$ or $3 \times 3$, if it is in reduced row echelon form and fit the standard matrix we can describe it as matrix operator. For example

$$A_1 = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$$

$A_1$ is a matrix corresponds to a shear in the $x$-direction with factor 3.

$$A_2 = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}$$

$A_2$ is a matrix corresponds to a shear in the $x$-direction with factor 2, and the $y$-direction with factor $-2$.

$$A_3 = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

$A_3$ is a matrix corresponds to an expansion in the $x$-direction with factor 3.

[DF*] **Example:**
Let $\boldsymbol{x}_0$ be a nonzero column vector in $\mathbb{R}^2$, and suppose that $T : \mathbb{R}^2 \to \mathbb{R}^2$ is the transformation defined by the formula

$$T(\boldsymbol{x}) = \boldsymbol{x}_0 + R_\theta \boldsymbol{x}$$

where $R_\theta$ is the standard matrix of the rotation of $\mathbb{R}^2$ about the origin through the angle $\theta$. Give a geometric description of this transformation. Is it a matrix transformation?
**Solution:**
The result of the transformation is rotation through the angle $\theta$ followed by translation by $\boldsymbol{x}_0$. It is not a matrix transformation since

$$T(\mathbf{0}) = \boldsymbol{x}_0 \neq \mathbf{0}$$

[DF*] **Example:**
Let $\boldsymbol{x} = \boldsymbol{x}_0 + t\boldsymbol{v}$ be a line in $\mathbb{R}^n$, and let $T : \mathbb{R}^n \to \mathbb{R}^n$ be a matrix operator on $\mathbb{R}^n$. What kind

of geometric object is the image of this line under the operator $T$? Explain your reasoning.
**Solution:**
The image of the line under the operator $T$ is the line in $\mathbb{R}^n$ which has vector representation

$$\boldsymbol{x} = T(\boldsymbol{x}_0) + tT(\boldsymbol{v})$$

unless $T$ is the zero operator in which case the image is $\mathbf{0}$.

[DF*] Suppose that $T_A$ is a matrix transformation from $\mathbb{R}^n$ to $\mathbb{R}^k$ and $T_B$ is a matrix transformation from $\mathbb{R}^k$ to $\mathbb{R}^m$. If $\boldsymbol{x}$ is a vector in $\mathbb{R}^n$, then $T_A$ maps this vector into a vector $T_A(\boldsymbol{x})$ in $\mathbb{R}^k$, and $T_B$ maps that vector into the vector $T_B(T_A(\boldsymbol{x}))$ in $\mathbb{R}^m$. This process creates a transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ that we call the composition of $T_B$ with $T_A$ and denote by the symbol

$$T_B \circ T_A$$

which is read "$T_B$ circle $T_A$."

The transformation $T_A$ is performed first

$$(T_B \circ T_A)(\boldsymbol{x}) = T_B(T_A(\boldsymbol{x})) \tag{23.91}$$

this composition is itself a matrix transformation since

$$(T_B \circ T_A)(\boldsymbol{x}) = T_B(T_A(\boldsymbol{x})) = B(T_A(\boldsymbol{x})) = B(A\boldsymbol{x}) = (BA)\boldsymbol{x}$$

which shows that it is multiplication by $BA$

$$T_B \circ T_A = T_{BA} \tag{23.92}$$

[DF*] The standard matrix for a composition is the product of the standard matrices in the appropriate order.

[DF*] Composition not always commutative, when we let $T_A : \mathbb{R}^2 \to \mathbb{R}^2$ be the reflection about the line $y = x$, and let $T_B : \mathbb{R}^2 \to \mathbb{R}^2$ be the orthogonal projection on the $y$-axis. $T_B \circ T_A$ and $T_A \circ T_B$ will have different effects on a vector $\boldsymbol{x}$ graphically.

$$T_B \circ T_A \neq T_A \circ T_B$$

For another case, let $T_A : \mathbb{R}^2 \to \mathbb{R}^2$ be the reflection about the $y$-axis, and let $T_B : \mathbb{R}^2 \to \mathbb{R}^2$ be the reflection about the $x$-axis. In this case $T_B \circ T_A$ and $T_A \circ T_B$ are the same; both map every vector $\boldsymbol{x} = (x, y)$ into its negative $-\boldsymbol{x} = (-x, -y)$:

$$(T_A \circ T_B)(x, y) = T_A(x, -y) = (-x, -y)$$
$$(T_B \circ T_A)(x, y) = T_B(-x, y) = (-x, -y)$$

thus

$$T_B \circ T_A = T_A \circ T_B$$

the standard matrices for $T_A$ and $T_B$ also commute

$$[T_B \circ T_A] = [T_A \circ T_B] = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

The operator $T(\boldsymbol{x}) = -\boldsymbol{x}$ on $\mathbb{R}^2$ or $\mathbb{R}^3$ is called the reflection about the origin.

**Definition 23.41: One-to-One Matrix Transformations**

A matrix transformation $T_A : \mathbb{R}^n \to \mathbb{R}^m$ is said to be one-to-one if $T_A$ maps distinct vectors (points) in $\mathbb{R}^n$ into distinct vectors (points) in $\mathbb{R}^m$.

[DF*] Rotation operators on $\mathbb{R}^2$ are one-to-one since distinct vectors that are rotated through the same angle have distinct images.

The orthogonal projection of $\mathbb{R}^3$ on the $xy$ plane is not one-to-one, because if we maps a vertical line, the whole points in that vector / vertical line will have the same image, remember that the image of $(x, y, z)$ for orthogonal projection on $xy$ plane is $(x, y, 0)$.

**Theorem 23.74: One-to-One Matrix Transformations**

If $A$ is an $n \times n$ matrix and $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is the corresponding matrix operator, then the following statements are equivalent.

(a) $A$ is invertible.

(b) The range of $T_A$ is in $\mathbb{R}^n$.

(c) $T_A$ is one-to-one.

[DF*] **Properties of Rotation Operator**
The operator $T : \mathbb{R}^n \to \mathbb{R}^n$ that rotates vectors in $\mathbb{R}^n$ through an angle $\theta$ is one-to-one. Confirm that $[T]$ is invertible.

**Solution:**
The standard matrix for $T$ is

$$[T] = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

This matrix is invertible because

$$\det[T] = \begin{vmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{vmatrix} = \cos^2\theta + \sin^2\theta = 1$$

since $\det[T] \neq 0$, the matrix is invertible and the operator $T$ is one-to-one.

[DF*] **Properties of Projection Operator**
The operator $T : \mathbb{R}^n \to \mathbb{R}^n$ that projects each vector in $\mathbb{R}^3$ orthogonally on the $xy$-plane is not one-to-one. Confirm that $[T]$ is not invertible.

**Solution:**
The standard matrix for $T$ is

$$[T] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

since $\det[T] = 0$, the matrix is not invertible and the operator $T$ is not one-to-one.

[DF*] If $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is a one-to-one matrix operator, then it follows that $A$ is invertible. The matrix operator

$$T_{A^{-1}} : \mathbb{R}^n \to \mathbb{R}^n$$

that corresponds to $A^{-1}$ is called the inverse operator or the inverse of $T_A$. This terminology is appropriate because $T_A$ and $T_{A^{-1}}$ cancel the effect of each other in the sense that if $\boldsymbol{x}$ is any vector in $\mathbb{R}^n$, then

$$T_A(T_{A^{-1}}(\boldsymbol{x})) = AA^{-1}\boldsymbol{x} = 1\boldsymbol{x} = \boldsymbol{x}$$
$$T_{A^{-1}}(T_A(\boldsymbol{x})) = A^{-1}A\boldsymbol{x} = 1\boldsymbol{x} = \boldsymbol{x}$$

If $T_A$ maps $\boldsymbol{x}$ to $\boldsymbol{w}$, then $T_{A^{-1}}$ maps $\boldsymbol{w}$ to $\boldsymbol{x}$.

[DF*] If $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is a one-to-one matrix operator, and if $T_{A^{-1}} : \mathbb{R}^n \to \mathbb{R}^n$ is its inverse, then the standard matrices for these operators are related by the equation

$$T_{A^{-1}} = T_A^{-1} \tag{23.93}$$

[DF*] **Standard Matrix for $T^{-1}$**

Let $T : \mathbb{R}^n \to \mathbb{R}^n$ be the operator that rotates each vector in $\mathbb{R}^2$ through the angle $\theta$, so the standard matrix is

$$[T] = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

it is evident geometrically that to undo the effect of $T$, one must rotate each vector in $\mathbb{R}^2$ through the angle $-\theta$. But this is exactly what the operator $T^{-1}$ does, since the standard matrix for $T^{-1}$ is

$$[T^{-1}] = [T]^{-1} = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix} = \begin{bmatrix} \cos\ (-\theta) & -\sin\ (-\theta) \\ \sin\ (-\theta) & \cos\ (-\theta) \end{bmatrix}$$

which is the standard matrix for a rotation through the angle $-\theta$.

[DF*] If $f_1, f_2, \ldots, f_m$ are any functions of the $n$ variables $x_1, x_2, \ldots, x_n$, then the equations

$$\begin{aligned} w_1 &= f_1(x_1, x_2, \ldots, x_n) \\ w_2 &= f_2(x_1, x_2, \ldots, x_n) \\ &\vdots \\ w_m &= f_m(x_1, x_2, \ldots, x_n) \end{aligned}$$

define a transformation $T : \mathbb{R}^n \to \mathbb{R}^m$ that maps the vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ into the vector $(w_1, w_2, \ldots, w_m)$. But it is only the case where these equations are linear that $T$ is a matrix transformation.

**Theorem 23.75: Algebraic Properties of Transformation $T : \mathbb{R}^n \to \mathbb{R}^m$**

$T : \mathbb{R}^n \to \mathbb{R}^m$ is a matrix transformation if and only if the following relationships hold for all vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ and for every scalar $k$:

(a) $T(\boldsymbol{u} + \boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$ (additivity property)

(b) $T(k\boldsymbol{u}) = kT(\boldsymbol{u})$ (homogeneity property)

**Theorem 23.76: Linear Transformation and Matrix Transformation**

Every linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ is a matrix transformation, and conversely, every matrix transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ is a linear transformation.

[DF*] Below is a renewal of equivalent statements, like renewing baptize, we add the last two points from this composition section.

**Theorem 23.77: Equivalent Statements for Square Matrix**

If $A$ is an $n \times n$ matrix, then the following statements are equivalent

(a) $A$ is invertible.

(b) $A\boldsymbol{x} = \mathbf{0}$ has only the trivial solution.

(c) The reduced row echelon form of $A$ is $I_n$.

(d) $A$ is expressible as a product of elementary matrices.

(e) $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for every $n \times 1$ matrix $\boldsymbol{b}$.

(f) $A\boldsymbol{x} = \boldsymbol{b}$ has exactly one solution for every $n \times 1$ matrix $\boldsymbol{b}$.

(g) $\det(A) \neq 0$.

(h) The column vectors of $A$ are linearly independent.

(i) The row vectors of $A$ are linearly independent.

(j) The column vectors of $A$ span $\mathbb{R}^n$.

(k) The row vectors of $A$ span $\mathbb{R}^n$.

(l) The column vectors of $A$ form a basis for $\mathbb{R}^n$.

(m) The row vectors of $A$ form a basis for $\mathbb{R}^n$.

(n) $A$ has rank $n$.

(o) $A$ has nullity 0.

(p) The orthogonal complement of the null space of $A$ is $\mathbb{R}^n$.

(q) The orthogonal complement of the row space of $A$ is $\{\mathbf{0}\}$.

(r) The range of $T_A$ is $\mathbb{R}^n$.

(s) $T_A$ is one-to-one.

[DF*] The inverse of reflection in $\mathbb{R}^2$ and $\mathbb{R}^3$ is the reflection itself toward the same axis / plane. The inverse of dilation is contraction. The inverse of rotation through an angle $\theta$ is the rotation through an angle $-\theta$ about the same axis (in $\mathbb{R}^2$ or $\mathbb{R}^3$).

[DF*] By inspection

1. The orthogonal projection on the $x$-axis in $\mathbb{R}^2$ is not one-to-one because the second row is all 0's, so the determinant is 0.
2. The reflection about the $y$-axis in $\mathbb{R}^2$ is one-to-one because the determinant is $-1$.
3. The reflection about the line $y = x$ in $\mathbb{R}^2$ is one-to-one because the determinant is $-1$.
4. The contraction with factor $k > 0$ in $\mathbb{R}^2$ is one-to-one because the determinant is $k^2 \neq 0$.
5. The rotation about the positive $z$-axis in $\mathbb{R}^3$ is one-to-one because the determinant is 1.
6. The reflection about the $xy$-plane in $\mathbb{R}^3$ is one-to-one because the determinant is $-1$.

7. The dilation with factor $k > 0$ in $\mathbb{R}^3$ is one-to-one because the determinant is $k^3 \neq 0$.

If today we can see the application of matrix transformation in photo editing with computer, in the future the automatic matrix transformation in $\mathbb{R}^3$ will be able to create a better and faster optimal engine, machine, computer' hardware, by resizing it to make it smaller but still with great proportion in a system as a whole, or to create a waste plant that convert trashes to electricity that need compact engine, smaller but able to do bigger job, that perhaps can take 10 million tons of trashes a day with a centralized machine as small as 24 inch CRT TV that able to control the gigantic medium to process the trashes.

[DF*] **Example:**

Determine whether $T : \mathbb{R}^2 \to \mathbb{R}^2$ is a matrix operator

(a) $T(x, y) = (2x, y)$

(b) $T(x, y) = (x^2, y)$

**Solution:**

(a) Let $\boldsymbol{u} = (x_1, y_1)$ and $\boldsymbol{v} = (x_2, y_2)$ thus

$$\begin{aligned} T(\boldsymbol{u}+\boldsymbol{v}) &= T((x_1, y_1) + (x_2, y_2)) \\ &= T(x_1 + x_2, y_1 + y_2) \\ &= (2(x_1 + x_2), y_1 + y_2) \\ &= (2x_1, y_1) + (2x_2, y_2) \\ T(\boldsymbol{u}+\boldsymbol{v}) &= T(\boldsymbol{u}) + T(\boldsymbol{v}) \end{aligned}$$

now for the homogeneity property, for any scalar $k$

$$\begin{aligned} T(k\boldsymbol{u}) &= T(k(x_1, y_1)) \\ &= T(kx_1, ky_1) \\ &= (2kx_1, ky_1) \\ &= k(2x_1, y_1) \\ T(k\boldsymbol{u}) &= kT(\boldsymbol{u}) \end{aligned}$$

$T$ is a matrix operator.

(b) Let $\boldsymbol{u} = (x_1, y_1)$ and $\boldsymbol{v} = (x_2, y_2)$ thus

$$\begin{aligned} T(\boldsymbol{u}+\boldsymbol{v}) &= T((x_1, y_1) + (x_2, y_2)) \\ &= T(x_1 + x_2, y_1 + y_2) \\ T(\boldsymbol{u}+\boldsymbol{v}) &= ((x_1^2 + 2x_1x_2 + x_2^2), y_1 + y_2) \end{aligned}$$

we know that

$$\begin{aligned} T(\boldsymbol{u}) + T(\boldsymbol{v}) &= T(x_1, y_1) + T(x_2, y_2) \\ &= (x_1^2, y_1) + (x_2^2, y_2) \end{aligned}$$

hence

$$T(\boldsymbol{u}+\boldsymbol{v}) \neq T(\boldsymbol{u}) + T(\boldsymbol{v})$$

$T$ is not a matrix operator. This only need to be proven from either the additivity property or homogeneity property. For the homogeneity property it can be shown like

this

$$\begin{aligned} T(k\boldsymbol{u}) &= T(k(x_1, y_1)) \\ &= T(kx_1, kx_2) \\ &= (k^2x_1^2, kx_2) \\ T(k\boldsymbol{u}) &= k(kx_1^2, y_1) \end{aligned}$$

and

$$kT(\boldsymbol{u}) = k(x_1^2, y_1)$$

thus

$$T(k\boldsymbol{u}) \neq kT(\boldsymbol{u})$$

[DF*] The overall effect of a matrix operator on $\mathbb{R}^2$ can often be ascertained by graphing the images of the vertices $(0,0), (1,0), (0,1)$, and $(1,1)$ of the unit square.

With this knowledge, now we need to know the transformed vertices of the image by multiplying the standard matrix with the original vertices, we will obtain

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

for the reflection about the $y$-axis case we will have

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

with $(x, y)$ are points / vertices of the original image, and $(x', y')$ are points / vertices of the transformed image. In the reflection about the $y$-axis case, the point $(1,1)$ will be transformed into $(-1,1)$.

[DF*] The standard matrix for each operator:

(a) Reflection about the $y$-axis

$$[T] = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

(b) Reflection about the $x$-axis

$$[T] = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

(c) Reflection about the line $y = x$

$$[T] = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

(d) Counterclockwise rotation through an angle $\theta$

$$[R_\theta] = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

(e) Expansion in the $x$-direction by a factor $k(k > 1)$

$$[T] = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

(f) Compression in the $x$-direction by a factor $k(0 < k < 1)$

$$[T] = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

(g) Expansion in the $y$-direction by a factor $k(k > 1)$

$$[T] = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

(h) Compression in the $y$-direction by a factor $k(0 < k < 1)$

$$[T] = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

(i) Shear in the $x$-direction by a factor $k(k > 0)$

$$[T] = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

(j) Shear in the $x$-direction by a factor $k(k < 0)$

$$[T] = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

(k) Shear in the $y$-direction by a factor $k(k > 0)$

$$[T] = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

(l) Shear in the $y$-direction by a factor $k(k < 0)$

$$[T] = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

[DF*] A matrix transformation $T_A$ is one-to-one if and only if $A$ can be expressed as a product of elementary matrices. Thus, we can analyze the effect of any one-to-one transformation $T_A$ by first factoring the matrix $A$ into a product of elementary matrices

$$A = E_1 E_2 \dots E_r$$

and then expressing $T_A$ as the composition

$$T_A = T_{E_1 E_2 \dots E_r} = T_{E_1} \circ T_{E_2} \circ \dots \circ T_{E_r}$$

**Theorem 23.78: Geometry of One-to-One Matrix Operators**

If $E$ is an elementary matrix, then $T_E : \mathbb{R}^2 \to \mathbb{R}^2$ is one of the following:

(a) A shear along a coordinate axis.

(b) A reflection about $y = x$.

(c) A compression along a coordinate axis.

(d) An expansion along a coordinate axis.

(e) A reflection about a coordinate axis.

(f) A compression or expansion along a coordinate axis followed by a reflection about a coordinate axis.

**Theorem 23.79: Geometric Effect of $T_A$**

If $T_A : \mathbb{R}^2 \to \mathbb{R}^2$ is multiplication by an invertible matrix $A$, then the geometric effect of $T_A$ is the same as an appropriate succession of shears, compressions, expansions, and reflections.

[DF*] **Example:**
Express

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

as a product of elementary matrices, and then describe the geometric effect of multiplication by $A$ in terms of shears, compressions, expansions, and reflections.
**Solution:**
$A$ can be reduced to $I$ as follows:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \to \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix} \to \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \to \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

with the elementary row operations as follows:

1. Add $-3$ times the first row to the second row.
2. Multiply the second row by $-\frac{1}{2}$.
3. Add $-2$ times the second row to the first row.

Now each elementary row operations will have its corresponding elementary matrices, from identity matrix of size $2 \times 2$ we perform the elementary row operation above to it, thus it will create 3 elementary matrices below

$$E_1 = \begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 1 & 0 \\ 0 & -\frac{1}{2} \end{bmatrix}, \quad E_3 = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$$

Inverting these elementary matrices yields

$$A = E_1^{-1} E_2^{-1} E_3^{-1}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

The first transformation is the one at outer right, reading from right to left. Also note that

$$\begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

It follows that the effect of multiplying by $A$ is equivalent to

1. Shearing by a factor of 2 in the $x$-direction

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

2. Then, expanding by a factor of 2 in the $y$-direction

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

3. Then, reflecting about the $x$-axis

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

4. Then shearing by a factor of 3 in the $y$-direction

$$\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}$$

**Theorem 23.80: Images of Lines Under Matrix Operators**

If $T : \mathbb{R}^2 \to \mathbb{R}^2$ is multiplication by an invertible matrix, then:

(a) The image of a straight line is a straight line.

(b) The image of a straight line through the origin is a straight line through the origin.

(c) The images of parallel straight lines are parallel straight lines.

(d) The image of the line segment joining points $P$ and $Q$ is the line segment joining the images of $P$ and $Q$.

(e) The images of three points lie on a line if and only if the points themselves lie on a line.

[DF*] **Example:**

(a) Show that multiplication by

$$A = \begin{bmatrix} 4 & 2 \\ 1 & 1 \end{bmatrix}$$

maps every point in the plane onto the line $y = \frac{1}{2}x$.

(b) The noncollinear points $(1,0), (0,1), (-1,0)$ are mapped onto a line. Does this violate the theorem that stated "If $T : \mathbb{R}^2 \to \mathbb{R}^2$ is multiplication by an invertible matrix, then the images of three points lie on a line if and only if the points themselves lie on a line." ?

**Solution:**

(a) Let $(x, y)$ be a point on a plane, the image under multiplication by $A$ is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

thus

$$x' = 4x + 2y$$
$$y' = 2x + y$$

from there we will obtain

$$y' = \frac{1}{2}x'$$

Thus $(x', y')$ satisfies

$$y = \frac{1}{2}x$$

it is proven that every point in the plane will be mapped onto the line $y = \frac{1}{2}x$.

(b) The three points $(1, 0), (0, 1), (-1, 0)$ are not located in a straight line, hence we need to check the matrix $A$ whether it violates the theorem

$$\det(A) = \frac{1}{(4)(1) - (2)(2)} \begin{bmatrix} 1 & -2 \\ -2 & 4 \end{bmatrix}$$

we have seen that matrix $A$ has no inverse / non-invertible matrix, thus it does not violate the theorem since the theorem will remain true as long as the matrix for the transformation is invertible. Whenever we see matrix with linear dependent row, then that matrix is non-invertible.

[DF*] **Example:**

Prove the theorem that states "The image of a straight line is a straight line."
In this case a line in the plane has an equation of the form $Ax + By + C = 0$, where $A$ and $B$ are not both zero. Show that the image of this line under multiplication by the invertible matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

has the equation $A'x + B'y + C = 0$, where

$$A' = \frac{dA - cB}{ad - bc}$$

and

$$B' = \frac{-bA + aB}{ad - bc}$$

Then show that $A'$ and $B'$ are not both zero to conclude that the image is a line.

**Solution:**

Let $(x, y)$ be a point on the line $Ax + By + C = 0$, and let $(x', y')$ be its image under multiplication by matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{ad-bc} \begin{bmatrix} dx' - by' \\ -cx' + ay' \end{bmatrix}$$

Substituting in $Ax + By + C = 0$ yields

$$\begin{aligned} Ax + By + C &= 0 \\ A\frac{1}{ad-bc}(dx' - by') + B\frac{1}{ad-bc}(-cx' + ay') + C &= 0 \\ \frac{1}{ad-bc}\left[A(dx' - by') + B(-cx' + ay')\right] + C &= 0 \\ \frac{1}{ad-bc}\left[(dAx' - cBx') + (aBy' - bAy')\right] + C &= 0 \\ \frac{1}{ad-bc}\left[dA - cB\right]x' + \frac{1}{ad-bc}[aB - bA]y' + C &= 0 \end{aligned}$$

Thus

$$\begin{aligned} Ax' + By' + C &= 0 \\ \frac{dA - cB}{ad-bc}x' + \frac{aB - bA}{ad-bc}y' + C &= 0 \end{aligned}$$

with

$$\begin{aligned} A &= \frac{dA - cB}{ad-bc} \\ B &= \frac{aB - bA}{ad-bc} \end{aligned}$$

We will show that $A'$ and $B'$ are not both zero, from the line equation we know that an invertible matrix will have

$$ad - bc \neq 0$$

then we will prove it with contradiction, assume that both $A'$ and $B'$ are both zero

$$\begin{aligned} A' &= 0 \\ dA - cB &= 0 \\ dA &= cB \\ \frac{A}{B} &= \frac{c}{d} \end{aligned}$$

$$\begin{aligned} B' &= 0 \\ aB - bA &= 0 \\ aB &= bA \\ \frac{A}{B} &= \frac{a}{b} \end{aligned}$$

then

$$\begin{aligned} \frac{c}{d} &= \frac{a}{b} \\ bc &= ad \\ ad - bc &= 0 \end{aligned}$$

contradiction, since an invertible matrix will have $ad - bc \neq 0$, this proves that $A'$ and $B'$ are not both zero, and the image is a line.

[DF*] A dynamical system is a finite set of variables whose values change with time. The value of a variable at a point in time is called the state of the variable at that time, and the vector formed from these states is called the state of the dynamical system at that time.

[DF*] In many dynamical systems the states of the variables are not known with certainty but can be expressed as probabilities; such dynamical systems are called stochastic processes.
If an experiment or observation has $n$ possible outcomes, then the probabilities of those outcomes must be nonnegative fractions whose sum is 1.

In a stochastic process with $n$ possible states, the state vector at each time $t$ has the form

$$\boldsymbol{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix}$$

with $x_i(t)$ is the probability that the system is in state $i$ with $i = 1, 2, \ldots, n$. The entries in this vector must add up to 1 since they account for all $n$ possibilities. In general, a vector with nonnegative entries that add up to 1 is called a probability vector.

[DF*] A square matrix, each of whose columns is a probability vector, is called a stochastic matrix. Such matrices commonly occur in formulas that relate successive states of a stochastic process.

For example, the state vectors $\boldsymbol{x}(k+1)$ and $\boldsymbol{x}(k)$ are related by an equation of the form

$$\boldsymbol{x}(k+1) = P\boldsymbol{x}(k) \tag{23.94}$$

in which

$$\begin{bmatrix} P_A & (1 - P_B) \\ (1 - P_A) & P_B \end{bmatrix}$$

is a stochastic matrix with:
$P_A$ is the probability that event $A$ will occur again at time $k+1$ after it occurs at time $k$.
$1 - P_A$ is the probability that event $A$ will not occur at time $k+1$ (event $B$ occurs at $k+1$) after event A occurs at time $k$.
$P_B$ is the probability that event $B$ will occur again at time $k+1$ after it occurs at time $k$.
$1 - P_B$ is the probability that event $B$ will not occur at time $k+1$ (event $A$ occurs at $k+1$) after event B occurs at time $k$.

**Definition 23.42: Markov Chain**

A Markov chain is a dynamical system whose state vectors at a succession of time interval are probability vectors and for which the state vectors at successive time intervals are related by an equation of the form

$$\boldsymbol{x}(k+1) = P\boldsymbol{x}(k)$$

in which $P = [p_{ij}]$ is a stochastic matrix and $p_{ij}$ is the probability that the system will be in state $i$ at time $t = k+1$ if it is in state $j$ at time $t = k$. The matrix $P$ is called the transition matrix for the system.

[DF*] In a Markov chain with an initial state of $\boldsymbol{x}(0)$, the successive state vectors are

$$\boldsymbol{x}(1) = P\boldsymbol{x}(0), \quad \boldsymbol{x}(2) = P\boldsymbol{x}(1), \quad \boldsymbol{x}(3) = P\boldsymbol{x}(2), \quad \boldsymbol{x}(4) = P\boldsymbol{x}(3), \ldots$$

For brevity, it is common to denote $\boldsymbol{x}(k) = \boldsymbol{x}_k$, which allows us to write the successive state vectors mroe briefly as

$$\boldsymbol{x}_1 = P\boldsymbol{x}_0, \quad \boldsymbol{x}_2 = P\boldsymbol{x}_1, \quad \boldsymbol{x}_3 = P\boldsymbol{x}_2, \quad \boldsymbol{x}_4 = P\boldsymbol{x}_3, \ldots \tag{23.95}$$

Alternatively, these state vectors can be expressed in terms of the initial state vector $\boldsymbol{x}_0$ as

$$\begin{aligned}
\boldsymbol{x}_1 &= P\boldsymbol{x}_0 \\
\boldsymbol{x}_2 &= P(P\boldsymbol{x}_0) = P^2\boldsymbol{x}_0 \\
\boldsymbol{x}_3 &= P(P^2\boldsymbol{x}_0) = P^3\boldsymbol{x}_0 \\
\boldsymbol{x}_4 &= P(P^3\boldsymbol{x}_0) = P^4\boldsymbol{x}_0
\end{aligned}$$

from which it follows that

$$\boldsymbol{x}_k = P^k\boldsymbol{x}_0 \tag{23.96}$$

[DF*] The matrix

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

is stochastic and hence can be regarded as the transition matrix for a Markov chain. A real life example for this transition matrix is the changing of day and night. It will always happens like that with probability 1, after day we will have night, and vice versa.
A simple calculation shows that

$$P^2 = I$$

from which it follows that

$$I = P^2 = P^4 = P^6 = \ldots$$

and

$$P = P^3 = P^5 = P^7 = \ldots$$

Thus, the successive states in the Markov chain with initial vector $\boldsymbol{x}_0$ are

$$\boldsymbol{x}_0, \ P\boldsymbol{x}_0, \ \boldsymbol{x}_0, \ P\boldsymbol{x}_0, \ \boldsymbol{x}_0, \ldots$$

which oscillate between $\boldsymbol{x}_0$ and $P\boldsymbol{x}_0$. Thus, the Markov chain does not stabilize unless both components of $\boldsymbol{x}_0$ are $\frac{1}{2}$.

**Definition 23.43: Regular Markov Chain**

A stochastic matrix $P$ is said to be regular if $P$ or some positive power of $P$ has all positive entries, and a Markov chain whose transition matrix is regular is said to be a regular Markov chain.

**Theorem 23.81: Long-term Behavior of Markov Chains: Steady-state Vector**

If $P$ is the transition matrix for a regular Markov chain, then:

(a) There is a unique probability vector $\boldsymbol{q}$ such that $P\boldsymbol{q} = \boldsymbol{q}$.

(b) For any initial probability vector $\boldsymbol{x}_0$, the sequence of the state vectors

$$\boldsymbol{x}_0,\ P\boldsymbol{x}_0, \ldots,\ P^k\boldsymbol{x}_0$$

converges to $\boldsymbol{q}$.

The vector $\boldsymbol{q}$ is called the steady-state vector of the Markov chain. It can be found by rewriting the equation as

$$(I - P)\boldsymbol{q} = \mathbf{0}$$

and then solving this equation for $\boldsymbol{q}$ subject to the requirement that $\boldsymbol{q}$ be a probability vector.

[DF*] **Example:**
Suppose that at some initial point in time $100,000$ people live in a certain city and $25,000$ people live in a village on the mountain. The Regional Planning Commission determines that each year 10% of the city population moves to the village for retirement or choose a healthier and more meaningful lifestyle and 3% of the village population moves to the city for study or work purpose.

(a) Assuming that the total population remains constant, make a table that shows the populations of the city and the village over a five-year period (round to the nearest integer).

(b) Over the long term, how will the population be distributed between the city and the village?

**Solution:**

(a) Let state 1 be living in the city, and state 2 be living in the village on a mountain. Then the transition matrix is

$$P = \begin{bmatrix} 0.9 & 0.03 \\ 0.1 & 0.97 \end{bmatrix}$$

the initial state vector is

$$\boldsymbol{x}_0 = \begin{bmatrix} 100,000 \\ 25,000 \end{bmatrix}$$

The populations over five-year period will be

$$Px_0 = \begin{bmatrix} 90,750 \\ 34,250 \end{bmatrix}, \quad P^2x_0 = \begin{bmatrix} 82,702 \\ 42,298 \end{bmatrix}, \quad P^3x_0 = \begin{bmatrix} 75,701 \\ 49,299 \end{bmatrix}$$

$$P^4x_0 = \begin{bmatrix} 69,610 \\ 55,390 \end{bmatrix}, \quad P^5x_0 = \begin{bmatrix} 64,311 \\ 60,689 \end{bmatrix}$$

(b) The system

$$(I - P)\boldsymbol{q} = \mathbf{0}$$

| *Year* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *City* | 90.750 | 82,702 | 75,701 | 69,610 | 64,311 |
| *Village* | 34,250 | 42,298 | 49,299 | 55,390 | 60,689 |

**Table 23.1:** *The populations of the city and the village over a five-year period.*

$$\begin{bmatrix} 0.1 & -0.03 \\ -0.1 & 0.03 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The general solution is

$$q_1 = \frac{3}{10}s, \quad q_2 = s$$

since

$$q_1 + q_2 = 1$$

we will have

$$\begin{aligned} \frac{3}{10}s + s &= 1 \\ \frac{13}{10}s &= 1 \\ s &= \frac{10}{13} = 0.7692308 \end{aligned}$$

and

$$q_1 = \frac{3}{13} = 0.23076922$$

the steady-state vector is

$$\boldsymbol{q} = \begin{bmatrix} \frac{3}{13} \\ \frac{10}{13} \end{bmatrix} = \begin{bmatrix} 0.23076922 \\ 0.7692308 \end{bmatrix}$$

Over the long term, the population of 125,000 will be distributed with $\frac{3}{13}(125,000) = 28,846$ in the city and $\frac{10}{13}(125,000) = 96,154$ in the village.

[DF*] **Example:**
Physical traits are determined by the genes that an offspring receives from its parents. In the simplest case a trait in the offspring is determined by one pair of genes, one member of the pair inherited from the male parent and the other from the female parent. Typically, each gene in a pair can assume one of two forms, called alleles, denoted by $A$ and $a$. This leads to three possible pairings

$$AA, \quad Aa, \quad aa$$

called genotypes (the pairs $Aa$ and $aA$ determine the same trait and hence are not distinguished from one another). It is shown in the study of heredity that if a parent of known genotype is crossed with a random parent of unknown genotype probabilities given in the following table, which can be viewed as a transition matrix for a Markov process: Thus, for example, the offspring of a parent of genotype $AA$ that is crossed at random with a parent of unknown genotype will have a 50% chance of being $AA$, a 50% chance of being $Aa$, and no chance of being $aa$.

| | **AA** | **Aa** | **aa** |
|---|---|---|---|
| **AA** | $\frac{1}{2}$ | $\frac{1}{4}$ | 0 |
| **Aa** | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ |
| **aa** | 0 | $\frac{1}{4}$ | $\frac{1}{2}$ |

**Table 23.2:** *The row represents the genotype of offspring and the column represents the genotype of parent.*

(a) Show that the transition matrix is regular

(b) Find the steady-state vector, and discuss its physical interpretation.

**Solution:**

(a) Let state 1 be the genotype being $AA$, state 2 be the genotype being $Aa$, and state 3 be the genotype being $aa$. Then the transition matrix is

$$P = \begin{bmatrix} \frac{1}{2} & \frac{1}{4} & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{4} & \frac{1}{2} \end{bmatrix}$$

the transition matrix is said to be regular if $P$ or some positive power of $P$ has all positive entries, thus

$$P^2 = \begin{bmatrix} \frac{3}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{8} & \frac{1}{4} & \frac{3}{8} \end{bmatrix}$$

since $P^2$ has positive entries, then the transition matrix $P$ is regular.

(b) Now to find the steady-state vector $\boldsymbol{q}$ we will use the formula

$$(I - P)\boldsymbol{q} = \mathbf{0}$$

then solving this equation for $\boldsymbol{q}$ we will have

$$\begin{bmatrix} \frac{1}{2} & -\frac{1}{4} & 0 \\ -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ 0 & -\frac{1}{4} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

the general solution of this system is

$$q_1 = s, \quad q_2 = 2s, \quad q_3 = s$$

for $\boldsymbol{q}$ to be a probability vector, we must have

$$\begin{aligned} q_1 + q_2 + q_3 &= 1 \\ s + 2s + s &= 1 \\ 4s &= 1 \\ s &= \frac{1}{4} \end{aligned}$$

the steady-state vector $\boldsymbol{q}$ is

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{2} \\ \frac{1}{4} \end{bmatrix}$$

over long-term, there is a 50% chance the offspring will have genotype $Aa$, there will be more offspring with genotype being $Aa$ compared to genotype being $AA$ or $aa$.

## i. C++ Computation: Wronskian for Symbolic Matrix

Taken from exercise set 4.3 number 24 from [11].

Use Wronski's test to show that the functions $f_1(x) = e^x, f_2(x) = xe^x$, and $f_3(x) = x^2e^x$ are linearly independent vectors in $F(-\infty, \infty)$.

**Solution:**
The Wronskian is

$$W(x) = \begin{vmatrix} e^x & xe^x & x^2e^x \\ e^x & e^x + xe^x & 2xe^x + x^2e^x \\ e^x & 2e^x + xe^x & 2e^x + 4xe^x + x^2e^x \end{vmatrix} = 2e^{3x}$$

Since $W(x) = 2e^{3x}$ is nonzero for all $x$, the vectors are linearly independent. Hence, $f_1(x) = e^x, f_2(x) = xe^x$, and $f_3(x) = x^2e^x$ span three-dimensional subspace of $F(-\infty, \infty)$ of $F(-\infty, \infty)$.

For the computation, we are going to use **SymbolicC++** so we are able to input symbolic function and determine the derivative till $n$-th terms as well. We also include the Gaussian Elimination algorithm to compute reduced row echelon form for the Wronskian matrix.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 3 // number of rows
#define C 3 // number of columns

// Driver program
int main()
{
        Symbolic x("x");
        Symbolic y1, dy1, ddy1;
        Symbolic y2, dy2, ddy2;
        Symbolic y3, dy3, ddy3;

        y1 = exp(x);
        dy1 = df(y1, x);
        ddy1 = df(dy1, x);

        y2 = x*exp(x);
        dy2 = df(y2, x);
        ddy2 = df(dy2, x);

        y3 = x*x*exp(x);
        dy3 = df(y3, x);
        ddy3 = df(dy3, x);

        // Construct a symbolic matrix of size 3 X 3
```

```
Matrix<Symbolic> W_mat(3,3);
W_mat[0][0] = y1;  W_mat[0][1] = y2;  W_mat[0][2] = y3;
W_mat[1][0] = dy1; W_mat[1][1] = dy2; W_mat[1][2] = dy3;
W_mat[2][0] = ddy1; W_mat[2][1] = ddy2; W_mat[2][2] = ddy3;
cout << "W:\n" << W_mat <<endl;
cout << "det(W):\n" << W_mat.determinant() <<endl;
//cout << "inv(B):\n" << B_mat.inverse() <<endl;
cout << endl;

// Perform Gaussian Elimination / Forward elimination
for (int k=0; k<R; k++)
{
       int i_max = k;
       Symbolic v_max = W_mat[i_max][k];

       if (i_max != k)

       for (int m=0; m<C; k++)
       {
              Symbolic temp = W_mat[k][m];
              W_mat[k][m] = W_mat[i_max][m];
              W_mat[i_max][m] = temp;
       }

       for (int i=k+1; i<R; i++)
       {
              // factor f to set current row kth element to 0 and
                  subsequently remaining kth column to 0
              Symbolic f = W_mat[i][k]/W_mat[k][k];

              // subtract fth multiple of corresponding kth row
                  element
              for (int j=k+1; j<C; j++)
              {
                     W_mat[i][j] -= W_mat[k][j]*f;
                     // filling lower triangular matrix with zeros
              }
              W_mat[i][k] = 0;
       }
}

//cout << "W (in reduced row form) :\n" << W_mat <<endl;

for (int i = 0; i < R; i++)
{
       Symbolic f2 = W_mat[i][i];
       for (int j = 0; j < C; j++)
       {
```

```
                        W_mat[i][j] = W_mat[i][j]/f2;
                    }
                }
                cout << "W (in row reduced echelon form) :\n" << W_mat <<endl;
                cout<<endl;
                return 0;
        }
```

**C++ Code 124:** *main.cpp "Wronskian for Symbolic Matrix"*

To compile it, type:
**make**
**./main**

Explanation for the codes:

- To be able to compute the derivative of each functions $f_1(x), f_2(x)$, and $f_3(x)$ we need to first declare them as **Symbolic**.

```
Symbolic x("x");
Symbolic y1, dy1, ddy1;
Symbolic y2, dy2, ddy2;
Symbolic y3, dy3, ddy3;

y1 = exp(x);
dy1 = df(y1, x);
ddy1 = df(dy1, x);

y2 = x*exp(x);
dy2 = df(y2, x);
ddy2 = df(dy2, x);

y3 = x*x*exp(x);
dy3 = df(y3, x);
ddy3 = df(dy3, x);
```

```
mbolicC++/ch23-Numerical Linear Algebra/Wronskian for Symbolic Matrix with Sym
licC++ ]# ./main
W:
[          e^x                    x*e^x                 x^(2)*e^x        ]
[          e^x                 e^x+x*e^x            2*x*e^x+x^(2)*e^x    ]
[          e^x                2*e^x+x*e^x        2*e^x+4*x*e^x+x^(2)*e^x]

det(W):
2*e^(3*x)

W (in row reduced echelon form) :
[  1     x   x^(2)]
[  0     1    2*x ]
[  0     0     1  ]
```

**Figure 23.44:** *The Wronskian computation with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Wronskian for Symbolic Matrix with SymbolicC++/main.cpp**).*

## ii. C++ Plot and Computation: 3D Plot of Standard Basis Reflected toward a Line into New Bases

Taken from a modified exercise set 4.6 number 19 from [11].

The exercise is concerning on $\mathbb{R}^2$, then Sentinel asked to make the plot for $\mathbb{R}^3$. The computation that are done by hand still need to be revised and any input is very much welcome and needed.

Now, let $S = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$ be the standard basis, and let $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ be the basis that results when the vectors in $S$ are reflected about the line that makes an angle $\theta$ with the positive $x$-axis on $xy$ plane, and an angle of elevation $\phi$ with the $xy$-plane. Assuming that $0 < \theta < \frac{\pi}{2}$ and $0 < \phi < \frac{\pi}{2}$.

The main point to comprehend this is to know that when a point $(1, 0)$ is rotated at an angle of $\theta$ toward the positive $x$-axis the new point will be located at $(\cos\theta, \sin\theta)$. Reflection toward a line with angle of $\theta$ toward the positive $x$-axis is the same as doing the rotation twice.

We will take a look at each of the standard basis and their reflection, trigonometry and geometry play an important part to be able to understand this.

In $\mathbb{R}^3$ we have three standard basis, which are

$$\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The first basis $\boldsymbol{e}_1 = (1, 0, 0)$ when reflected to the line it will first be reflected in the $xy$ plane thus we will find the coordinate for the $x$ and $y$-axis for the new basis in $xy$ plane by using the angle $\theta$, then it will be reflected with the elevation angle of $\phi$ so we will obtain the last $x, y, z$ coordinate for the new basis $\boldsymbol{v}_1$ which is

$$\boldsymbol{v}_1 = \begin{bmatrix} \cos 2\theta \ \cos 2\phi \\ \sin 2\theta \ \cos 2\phi \\ \sin 2\phi \end{bmatrix}$$

For the second basis $\boldsymbol{e}_2$ it is done in 2 reflections like the first basis, hence

$$\boldsymbol{v}_2 = \begin{bmatrix} \sin 2\theta \ \cos 2\phi \\ -\cos 2\theta \ \cos 2\phi \\ \sin 2\phi \end{bmatrix}$$

The last standard basis $\boldsymbol{e}_3$ to become the new basis $\boldsymbol{v}_3$.

The first reflection is from $\boldsymbol{e}_3$ on the $xz$ plane, reflect the standard basis toward the line that is making the angle of elevation of $\phi$, afterward the second reflection is rotating in the $xy$ plane with angle of $\theta$ (counterclockwise rotation).

Then the new basis will have $x$ coordinate as $\sin 2\phi \ \cos \ \theta$, $y$ coordinate as $\sin 2\phi \ \sin \ \theta$ and $z$ coordinate as $-\cos 2\phi$ (see the related diagram to be able to comprehend).

**Figure 23.45:** *The diagram of $\boldsymbol{e}_1 = (1, 0, 0)$ reflected about the line that makes an angle $\theta$ with the positive x-axis to become $\boldsymbol{v}_1$. This is the first part of reflection for $\boldsymbol{e}_1$ to gain the x and y coordinate in the xy plane before we will reflect again to find the new $x, y, z$ coordinate for $\boldsymbol{v}_1$.*

**Figure 23.46:** *The diagram of $\boldsymbol{v}_1$ that already reflected in the xy plane and now will be reflected about an angle $\phi$ toward the xy plane to become $\boldsymbol{v}_1'$ (the final basis).*

**Figure 23.47:** *The diagram of $\boldsymbol{e}_2 = (0, 1, 0)$ reflected about the line that makes an angle $\theta$ with the positive x-axis to become $\boldsymbol{v}_2$. This is the first part of reflection for $\boldsymbol{e}_2$ to gain the x and y coordinate in the xy plane before we will reflect again to find the new $x, y, z$ coordinate for $\boldsymbol{v}_2$.*

**Figure 23.48:** *The diagram of $\boldsymbol{v}_2$ that already reflected in the xy plane and now will be reflected about an angle $\phi$ toward the xy plane to become $\boldsymbol{v}_2'$ (the final basis).*

$$\boldsymbol{v}_3 = \begin{bmatrix} \sin 2\phi \\ \sin 2\phi \\ -\cos 2\phi \end{bmatrix}$$

The transition matrix $P_{B\to S}$ is

$$P_{B\to S} = \begin{bmatrix} \cos 2\theta\ \cos 2\phi & \sin 2\theta\ \cos 2\phi & \sin 2\phi\ \cos\theta \\ \sin 2\theta\ \cos 2\phi & -\cos 2\theta\ \cos 2\phi & \sin 2\phi\ \sin\theta \\ \sin 2\phi & \sin 2\phi & -\cos 2\phi \end{bmatrix}$$

**Figure 23.49:** *The diagram of* $\boldsymbol{e}_3 = (0, 0, 1)$ *reflected toward the line that makes an angle* $\phi$ *with the* $xy$ *plane to become* $\boldsymbol{v}_3$.

For the C++ we are using Gnuplot, the computation for the new coordinate for the new bases is done by hand, we just input the formula directly. We choose the angle $\theta = 35^0$ and $\phi = 30^0$.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>
#include <bits/stdc++.h>

#include "gnuplot-iostream.h"

#define RADTODEG 57.295779513082320876f
#define DEGTORAD 0.0174532925199432957f

using namespace std;

int n = 3;
float phi = 30; // Angle of elevation between positive z-axis and xy-
    plane
float theta = 35; // Angle between the line and positive x-axis

int main() {
       Gnuplot gp;

       float u1 = 1;
       float u2 = 0;
       float u3 = 0;

       float v1 = 0;
       float v2 = 1;
       float v3 = 0;

       float w1 = 0;
       float w2 = 0;
       float w3 = 1;
       // Define the line that makes an angle theta with the positive x-
           axis and angle of elevation of phi with the xy-plane
       float l1 = cos(DEGTORAD*theta);
       float l2 = sin(DEGTORAD*theta);
       float l3 = sin(DEGTORAD*phi);

       // The new basis that reflected about the line that makes an angle
           theta with the positive x-axis
       // and angle of elevation of phi with the xy-plane
       float r1 = cos(2*DEGTORAD*theta) * cos(2*DEGTORAD*phi);
       float r2 = sin(2*DEGTORAD*theta) * cos(2*DEGTORAD*phi);
       float r3 = sin(2*DEGTORAD*phi);
```

```
float s1 = sin(2*DEGTORAD*theta) * cos(2*DEGTORAD*phi);
float s2 = -cos(2*DEGTORAD*theta) * cos(2*DEGTORAD*phi);
float s3 = sin(2*DEGTORAD*phi);

float t1 = sin(2*DEGTORAD*phi) * cos(DEGTORAD*theta);
float t2 = sin(2*DEGTORAD*phi) * sin(DEGTORAD*theta);
float t3 = -cos(2*DEGTORAD*phi);

float vect_u[] = { u1, u2, u3 };
float vect_v[] = { v1, v2, v3 };
float vect_w[] = { w1, w2, w3 };
float vect_s[] = { r1, r2, r3 };
float vect_t[] = { s1, s2, s3 };
float vect_r[] = { t1, t2, t3 };

cout << "Vector e1:" << setw(15) << "vector e2:" << setw(15) << "
    vector e3:" << endl;

for (int i = 0; i < n; i++)
{
       cout << vect_u[i] << "\t" << "\t" << vect_v[i] << "\t" << "\t
           " << vect_w[i] ;
       cout << endl;
}
cout << endl;
cout << "Vector v1:" << setw(15) << "vector v2:" << setw(15) << "
    vector v3:" << endl;

for (int i = 0; i < n; i++)
{
       cout << setprecision(4) << vect_s[i] << "\t" << "\t" <<
           vect_t[i] << "\t" << "\t" << vect_r[i] ;
       cout << endl;
}
cout<<endl;

cout << "Angle between the line and positive x-axis (degrees) = "
    << theta <<endl;
cout << "Angle of elevation between the line and the xy-plane (
    degrees) = " << phi <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
```

```
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_z;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_E_dz;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_z;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;
        std::vector<double> pts_F_dz;
        std::vector<double> pts_G_x;
        std::vector<double> pts_G_y;
        std::vector<double> pts_G_z;
        std::vector<double> pts_G_dx;
        std::vector<double> pts_G_dy;
        std::vector<double> pts_G_dz;

        float o = 0;

        pts_A_x .push_back(o);
        pts_A_y .push_back(o);
        pts_A_z .push_back(o);
        pts_A_dx.push_back(u1);
        pts_A_dy.push_back(u2);
        pts_A_dz.push_back(u3);

        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
```

```
            pts_B_z .push_back(o);
            pts_B_dx.push_back(v1);
            pts_B_dy.push_back(v2);
            pts_B_dz.push_back(v3);

            pts_C_x .push_back(o);
            pts_C_y .push_back(o);
            pts_C_z .push_back(o);
            pts_C_dx.push_back(w1);
            pts_C_dy.push_back(w2);
            pts_C_dz.push_back(w3);

            pts_D_x .push_back(o);
            pts_D_y .push_back(o);
            pts_D_z .push_back(o);
            pts_D_dx.push_back(r1);
            pts_D_dy.push_back(r2);
            pts_D_dz.push_back(r3);

            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(s1);
            pts_E_dy.push_back(s2);
            pts_E_dz.push_back(s3);

            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(t1);
            pts_F_dy.push_back(t2);
            pts_F_dz.push_back(t3);
            // Create the line
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(l1);
            pts_G_dy.push_back(l2);
            pts_G_dz.push_back(l3);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-1:2]\nset yrange [-1:2]\nset zrange [-1:2]\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "set view 70,10,1.2\n"; // pitch,yaw,zoom
            gp << "splot '-' with vectors title 'e_{1}', '-' with vectors title
                 'e_{2}', '-' with vectors title 'e_{3}', '-' with vectors
                title 'v_{1}', '-' with vectors title 'v_{2}','-' with vectors
```

```
                 title 'v_{3}','—' with vectors title 'line' \n";
            gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
                pts_A_dy, pts_A_dz));
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
                pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
                pts_C_dy, pts_C_dz));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
    }
```

**C++ Code 125:** *main.cpp "3D Plot of Standard Basis Reflected toward a Line into New Bases"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.50:** *The 3D plot of the standard basis and the new basis that resulted from reflecting the standard basis toward a line with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Changing Basis That Are Reflected About a Line/main.cpp**).*

### iii. C++ Computation: Particular Solution of a Nonhomogeneous Linear System $A\boldsymbol{x} = \boldsymbol{b}$ and General Solution of a Homogeneous Linear System $A\boldsymbol{x} = \boldsymbol{0}$

In this section, we will compute particular solution of a linear system

$$\begin{bmatrix} 1 & 3 & -2 & 0 & 2 & 0 \\ 2 & 6 & -5 & -2 & 4 & -3 \\ 0 & 0 & 5 & 10 & 0 & 15 \\ 2 & 6 & 0 & 8 & 4 & 18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

and a general solution of a homogeneous linear system

$$\begin{bmatrix} 1 & 3 & -2 & 0 & 2 & 0 \\ 2 & 6 & -5 & -2 & 4 & -3 \\ 0 & 0 & 5 & 10 & 0 & 15 \\ 2 & 6 & 0 & 8 & 4 & 18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The algorithm for the C++:

- For the nonhomogeneous solution, we will use **Armadillo** library, this library is amazing that it can compute the solution with great precision.
- For the homogeneous solution, since it will have many / infinite solutions, We will perform Gaussian elimination that we define in the function in the C++ code to reduce matrix to reduced row echelon form, then assigning parameters to the corresponding variables.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

#define M 10

// Function to print the matrix
void PrintMatrix(float a[][M])
{
        int r = 4;
        int c = 7;
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
```

```
                cout <<setw(6) << a[i][j] << "\t";
                cout << endl;
        }
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][M], int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pro = 0;

        // Performing elementary operations
        for (i = 0; i < n; i++)
        {
                if (a[i][i] == 0)
                {
                        c = 1;
                        while ((i + c) < n && a[i + c][i] == 0)
                        c++;
                        if ((i + c) == n)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

// Function to print the desired result
// if unique solutions exists, otherwise
// prints no solution or infinite solutions
// depending upon the input given.
```

```
void PrintResult(float a[][M], int n, int flag)
{
        cout << "The solution/s : ";

        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "x(" << i << ") = "<< a[i][n] / a[i][i] << ", ";
        }
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float a[][M], int n, int flag)
{
        int i, j;
        float sum;

        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < n; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        mat B;
```

```
B.load("vectorB.txt");
mat X;
X = solve(A,B,solve_opts::force_approx);

cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
cout <<"Solution:" << "\n" << X <<endl;

float b[M][M] = {{ 1,3, -2,0,2,0,0 },
       { 2,6,-5,-2,4,-3,0 },
       { 0,0,5,10,0,15,0 },
       { 2,6,0,8,4,18,0 }};

cout << "Homogeneous System : " << endl;
int r = 4;
int c = 7;
for (int i = 0; i < r; ++i)
{
       for (int j = 0; j < c; ++j)
       {
              cout <<setw(6) << b[i][j]<< "\t";
       }
       cout << endl;
}

// Order of Matrix(n)
int n = 7, flag = 0;

// Performing Matrix transformation
flag = PerformOperation(b, n);

if (flag == 1)
flag = CheckConsistency(b, n, flag);

// Printing Final Matrix
cout << endl;
cout << "Gaussian Elimination for Homogeneous System: " << endl;
PrintMatrix(b);
cout << endl;

// Printing Solutions for homogeneous system
PrintResult(b,n, flag);
cout << endl;

for (int i = 0; i < r; i++)
{
       float f2 = b[i][i];
       for (int j = 0; j < c; j++)
```

```
                {
                        b[i][j] = b[i][j]/f2;
                }
        }
        cout << "Row Echelon Form for Homogeneous System : " << endl;
        PrintMatrix(b);
}
```

**C++ Code 126:** *main.cpp "Particular and General Solution of a Linear System"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A (Coefficient Matrix):
    1.0000    3.0000   -2.0000         0    2.0000         0
    2.0000    6.0000   -5.0000   -2.0000    4.0000   -3.0000
         0         0    5.0000   10.0000         0   15.0000
    2.0000    6.0000         0    8.0000    4.0000   18.0000

Vector B (Nonhomogeneous System):
         0
   -1.0000
    5.0000
    6.0000

Solution:
   -2.2432e-16
    6.9438e-16
   -2.5938e-16
    2.2204e-16
    4.2071e-16
    3.3333e-01

Homogeneous System :
     1       3      -2       0       2       0       0
     2       6      -5      -2       4      -3       0
     0       0       5      10       0      15       0
     2       6       0       8       4      18       0

Gaussian Elimination for Homogeneous System:
     1       3      -2       0       2       0       0
     0       0      -1      -2       0      -3       0
     0       0       5      10       0      15       0
     0       0       4       8       0      18       0

The solution/s : Infinite Solutions Exists

Row Echelon Form for Homogeneous System :
     1       3      -2       0       2       0       0
   -nan    -nan    -inf    -inf    -nan    -inf    -nan
     0       0       1       2       0       3       0
     0       0     0.5       1       0    2.25       0
```

**Figure 23.51:** *The computation to determine the particular solution of a nonhomogeneous linear system $A\boldsymbol{x} = \boldsymbol{b}$ and general solution of a $A\boldsymbol{x} = \boldsymbol{0}$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/General and Particular Solution for Underdetermined System with Armadillo/main.cpp**).*

From the computation we will obtain the particular solution of $A\boldsymbol{x} = \boldsymbol{b}$ as follow

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3} \end{bmatrix}$$

then the general solution of $A\boldsymbol{x} = \boldsymbol{0}$ are parametrized first

$$\begin{aligned} x_2 &= r \\ x_4 &= s \\ x_5 &= t \\ x_6 &= 0 \end{aligned}$$

then

$$\begin{aligned} x_1 &= -3x_2 + 2x_3 - 2x_5 \\ x_3 &= -2x_4 \end{aligned}$$

thus after substitution, we will obtain the solution in parameters term

$$\begin{aligned} x_1 &= -3r - 4s - 2t \\ x_2 &= r \\ x_3 &= -2s \\ x_4 &= s \\ x_5 &= t \\ x_6 &= 0 \end{aligned}$$

The particular solution $\boldsymbol{x}_0$ of the nonhomegenous system and the general solution $\boldsymbol{x}_h$ of the corresponding homogeneous system are related by

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} -3r - 4s - 2t \\ r \\ -2s \\ s \\ t \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3} \end{bmatrix} + r \begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -4 \\ 0 \\ -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

with

$$\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{x}_h$$

$\boldsymbol{x}$ is the general solution of $A\boldsymbol{x} = \boldsymbol{b}$, the vector $\boldsymbol{x}_0$ is the particular solution of $A\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{x}_h$ is the general solution of $A\boldsymbol{x} = \boldsymbol{0}$.

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3} \end{bmatrix}$$

$$\boldsymbol{x}_h = r \begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -4 \\ 0 \\ -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

the vectors

$$\begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -4 \\ 0 \\ -2 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

form a basis for the solution space $A\boldsymbol{x} = \mathbf{0}$ / the null space of $A$.

Another option is to try another algorithm in C++ that applying the row operations several times, the result itself is better:

```
Matrix A (Coefficient Matrix):
    1.0000    3.0000   -2.0000         0    2.0000         0
    2.0000    6.0000   -5.0000   -2.0000    4.0000   -3.0000
         0         0    5.0000   10.0000         0   15.0000
    2.0000    6.0000         0    8.0000    4.0000   18.0000

Vector B (Nonhomogeneous System):
         0
  -1.0000
   5.0000
   6.0000

Solution:
  -2.2432e-16
   6.9438e-16
  -2.5938e-16
   2.2204e-16
   4.2071e-16
   3.3333e-01

Matrix for the linear system Ax=b :
     1       3      -2       0       2       0       0
     2       6      -5      -2       4      -3      -1
     0       0       5      10       0      15       5
     2       6       0       8       4      18       6
Matrix for the linear system Ax=0 :
     1       3      -2       0       2       0       0
     2       6      -5      -2       4      -3       0
     0       0       5      10       0      15       0
     2       6       0       8       4      18       0

G****************************Freya*****************

Row Echelon Form for linear system Ax=b :
     1       3      -2       0       2       0       0
    -0      -0       1       2      -0       3       1
     0       0       0       0       0       1  0.3333
     0       0       0       0       0       0       0

G****************************Freya*****************

Row Echelon Form for linear system Ax=0 :
     1       3      -2       0       2       0       0
    -0      -0       1       2      -0       3      -0
     0       0       0       0       0       1       0
     0       0       0       0       0       0       0
```

**Figure 23.52:** *The computation with better result that is similar with the next section to determine the particular solution of a nonhomogeneous linear system $A\boldsymbol{x} = \boldsymbol{b}$ and general solution of a $A\boldsymbol{x} = \boldsymbol{0}$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/General Solution for Underdetermined System Ax=b and Ax=0 with Armadillo Case 1/main.cpp**).*

## iv. C++ Computation: Vector Form of the General Solution of a Homogeneous Linear System $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{x} = \boldsymbol{0}$

The method for this section is almost the same as the previous section. But the C++ code will be different, and the problem is stated in linear system instead of matrix form.

Find the vector form of the general solution of the given linear system $A\boldsymbol{x} = \boldsymbol{b}$; then use the result to find the vector form of the general solution of $A\boldsymbol{x} = \boldsymbol{0}$.

$$\begin{bmatrix} x_1 & -2x_2 & +x_3 & +2x_4 & = & -1 \\ 2x_1 & -4x_2 & +2x_3 & +4x_4 & = & -2 \\ -x_1 & +2x_2 & -x_3 & -2x_4 & = & 1 \\ 3x_1 & -6x_2 & +3x_3 & +6x_4 & = & -3 \end{bmatrix}$$

**Solution:**

The linear system above in matrix form will be

$$\begin{bmatrix} 1 & -2 & 1 & 2 & & \\ 2 & -4 & 2 & 4 & & \\ 0 & 0 & 5 & 10 & 0 & 15 \\ 2 & 6 & 0 & 8 & 4 & 18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 1 \\ -3 \end{bmatrix}$$

Thus the linear system in matrix form is

$$\begin{bmatrix} 1 & -2 & 1 & 2 & -1 \\ 2 & -4 & 2 & 4 & -2 \\ -1 & 2 & -1 & -2 & 1 \\ 3 & -6 & 3 & 6 & -3 \end{bmatrix}$$

the reduced row echelon form for the linear system

$$\begin{bmatrix} 1 & -2 & 1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

the solution of $A\boldsymbol{x} = \boldsymbol{b}$ is

$$x_1 = -1 + 2x_2 - x_3 - 2x_4$$

we will parametrize the other variables as

$$x_2 = r$$
$$x_3 = s$$
$$x_4 = t$$

thus

$$x_1 = -1 + 2r - s - 2t$$

the solution in vector form for $A\boldsymbol{x} = \boldsymbol{b}$

$$\boldsymbol{x} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + r \begin{bmatrix} 2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} -2 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

the general solution in vector form for $A\boldsymbol{x} = \mathbf{0}$

$$\boldsymbol{x} = r \begin{bmatrix} 2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} -2 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The C++ code is going to be different than the previous section, even if the goal is the same to perform Gaussian elimination in order to obtain reduced row echelon form and find the general solution of linear system $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{x} = \mathbf{0}$.

In this section, we load the matrix $A$ (the coefficient matrix) from textfile **matrixA.txt** and the vector $\boldsymbol{b}$ from textfile **vectorB.txt** so no need to define or write the matrix entries from inside the C++ code.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

#define C_Mat 5 // Define the number of column for the linear system Ax=b
#define R_Mat 4 // Define the number of row for the linear system Ax=b
#define C_MatA 4 // Define the number of column for the matrix A
#define R_MatA 4 // Define the number of row for the matrix A

// Function to print the matrix
void PrintMatrix(float a[][C_Mat])
{
        int r = R_Mat;
        int c = C_Mat;
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(4) << a[i][j] << "\t";
                cout << endl;
        }
}
```

```
// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][C_Mat], int r, int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < r; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < r && a[i + c][i] == 0)
			c++;
			if ((i + c) == r)
			{
				flag = 1;
				break;
			}
			for (j = i, k = 0; k <= n; k++)
			swap(a[j][k], a[j+c][k]);
		}

		for (j = 0; j < n; j++)
		{
			// Excluding all i == j
			if (i != j)
			{
				// Converting Matrix to reduced row echelon
				    form(diagonal matrix)
				float pro = a[j][i] / a[i][i];
				for (k = 0; k <= n; k++)
				a[j][k] = a[j][k] - (a[i][k]) * pro;
			}
		}
	}
	return flag;
}

void PrintResult(float a[][C_Mat], int r, int flag)
{
	cout << "The solution/s : ";

	if (flag == 2)
	{
		cout << "Infinite Solutions Exists" << endl;
	}
	else if (flag == 3)
```

```
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < r; i++)
                cout << "x(" << i << ") = "<< a[i][r] / a[i][i] << ", ";
        }
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float a[][C_Mat], int r, int n, int flag)
{
        int i, j;
        float sum;

        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < r; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}

// Driver code
int main(int argc, char** argv)
{
        arma::mat A(R_MatA,C_MatA,fill::zeros);
        A.load("matrixA.txt");
        mat B;
        B.load("vectorB.txt");
        mat X;
        X = solve(A,B,solve_opts::force_approx);

        cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
        cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
        cout <<"Solution:" << "\n" << X <<endl;

        float matrixc[R_Mat][C_Mat] = {};
```

```
        float matrixd[R_Mat][C_Mat] = {};

        for (int i = 0; i < R_Mat; ++i)
        {
               for(int j = 0; j<C_Mat-1; ++j)
               {
                      matrixc[i][j] = A[i+j*R_Mat] ;
                      matrixd[i][j] = A[i+j*R_Mat] ;
               }
        }
        for (int i = 0; i < R_Mat; ++i)
        {
               for(int j = C_Mat-1; j<C_Mat; ++j)
               {
                      matrixc[i][j] = B[i] ;
                      matrixd[i][j] = 0 ;
               }
        }

        cout <<"Matrix for the linear system Ax=b :" <<endl;
        PrintMatrix(matrixc);
        cout <<"Matrix for the linear system Ax=0 :" <<endl;
        PrintMatrix(matrixd);

        // Order of Matrix(n)
        int n = 7, flag = 0;

        // Performing Matrix transformation
        flag = PerformOperation(matrixc, R_Mat, C_Mat);

        if (flag == 1)
        flag = CheckConsistency(matrixc, R_Mat, C_Mat, flag);

        cout << endl;
        cout << "G*****************************Freya*****************" <<
             endl;
        cout << endl;

        // Printing Solutions for homogeneous system
        PrintResult(matrixc,R_Mat, flag);
        cout << endl;

        cout << "Row Echelon Form for linear system Ax=b : " << endl;
        PrintMatrix(matrixc);

        cout << endl;
        cout << "G*****************************Freya*****************" <<
             endl;
```

```
            cout << endl;

            PerformOperation(matrixd, R_Mat, C_Mat);
            cout <<"Row Echelon Form for linear system Ax=0 :" <<endl;
            PrintMatrix(matrixd);

      }
```

**C++ Code 127:** *main.cpp "General Solution of a Linear System Ax*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We define the matrix $A$ with Armadillo to load the textfile **MatrixA.txt** along with the vector $\boldsymbol{b}$ to load from **vectorB.txt**, and then declare two matrices *matrixc* and *matrixd* that we will use to perform the row operation / Gaussian elimination.

$$matrixc \;\rightarrow\; A\boldsymbol{x} = \boldsymbol{b}$$

$$matrixd \;\rightarrow\; A\boldsymbol{x} = \boldsymbol{0}$$

```
            arma::mat A(R_MatA,C_MatA,fill::zeros); //declare matrix A with
                 size of R X C all have 0 entries with Armadillo.
            A.load("matrixA.txt");
            mat B;
            B.load("vectorB.txt");
            mat X;
            X = solve(A,B,solve_opts::force_approx);

            cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
            cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
            cout <<"Solution:" << "\n" << X <<endl;

            float matrixc[R_Mat][C_Mat] = {};
            float matrixd[R_Mat][C_Mat] = {};

            for (int i = 0; i < R_Mat; ++i)
            {
                  for(int j = 0; j<C_Mat-1; ++j)
                  {
                        matrixc[i][j] = A[i+j*R_Mat] ;
                        matrixd[i][j] = A[i+j*R_Mat] ;
                  }
```

```
        }
        for (int i = 0; i < R_Mat; ++i)
        {
                for(int j = C_Mat-1; j<C_Mat; ++j)
                {
                        matrixc[i][j] = B[i] ;
                        matrixd[i][j] = 0 ;
                }
        }
```

```
Matrix A (Coefficient Matrix):
   1.0000  -2.0000   1.0000   2.0000
   2.0000  -4.0000   2.0000   4.0000
  -1.0000   2.0000  -1.0000  -2.0000
   3.0000  -6.0000   3.0000   6.0000

Vector B (Nonhomogeneous System):
  -1.0000
  -2.0000
   1.0000
  -3.0000

Solution:
  -0.1000
   0.2000
  -0.1000
  -0.2000

Matrix for the linear system Ax=b :
     1      -2       1       2      -1
     2      -4       2       4      -2
    -1       2      -1      -2       1
     3      -6       3       6      -3
Matrix for the linear system Ax=0 :
     1      -2       1       2       0
     2      -4       2       4       0
    -1       2      -1      -2       0
     3      -6       3       6       0

G****************************Freya*****************

The solution/s : Infinite Solutions Exists

Row Echelon Form for linear system Ax=b :
     1      -2       1       2      -1
     0       0       0       0       0
     0       0       0       0       0
     0       0       0       0       0

G****************************Freya*****************

Row Echelon Form for linear system Ax=0 :
     1      -2       1       2       0
     0       0       0       0       0
     0       0       0       0       0
     0       0       0       0       0
```

**Figure 23.53:** *The computation to determine the general solution of a nonhomogeneous linear system $A\boldsymbol{x} = \boldsymbol{b}$ and general solution of a $A\boldsymbol{x} = \boldsymbol{0}$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/General Solution for Underdetermined System Ax=b and Ax=0 with Armadillo /main.cpp**).*

## v. C++ Computation: Compute Basis for a Column Space by Row Reduction

We will use C++ to find a basis for the column space of the matrix

$$A = \begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 2 & -6 & 9 & -1 & 8 & 2 \\ 2 & -6 & 9 & -1 & 9 & 7 \\ -1 & 3 & -4 & 2 & -5 & -4 \end{bmatrix}$$

with Gaussian elimination we can reduce the matrix above into the row echelon form

$$R = \begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 0 & 0 & 1 & 3 & -2 & -6 \\ 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Remember that $A$ and $R$ can have different column spaces, thus we cannot find a basis for the column space of $A$ directly from the column vectors of $R$.

But, if we can find a set of column vectors of $R$ that forms a basis for the column space of $R$, then the corresponding column vectors of $A$ will form a basis for the column space of $A$.

Now, let's examine the matrix $R$, since the first, third and fifth columns of $R$ contain the leading 1's of the row vectors, the vectors

$$\boldsymbol{c}_1' = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{c}_3' = \begin{bmatrix} 4 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{c}_5' = \begin{bmatrix} -5 \\ -2 \\ 1 \\ 0 \end{bmatrix}$$

Thus, the corresponding column vectors of $A$, which are

$$\boldsymbol{c}_1 = \begin{bmatrix} 1 \\ 2 \\ 2 \\ -1 \end{bmatrix}, \quad \boldsymbol{c}_3 = \begin{bmatrix} 4 \\ 9 \\ 9 \\ -4 \end{bmatrix}, \quad \boldsymbol{c}_5 = \begin{bmatrix} 5 \\ 8 \\ 9 \\ -5 \end{bmatrix}$$

form a basis for the column space of $A$ that span the subspace $\mathbb{R}^4$.

We take the matrix from textfile **matrixA.txt**, so it will be easier for future development. Besides the main problem, we also add at the beginning if the matrix is paired with vector $\boldsymbol{b}$ to make a nonhomogeneous linear system $A\boldsymbol{x} = \boldsymbol{b}$, so we can also determine the solution.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;
```

```
#define M 10

// Function to print the matrix
void PrintMatrix(float a[][M])
{
	int r = 4;
	int c = 7;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << a[i][j] << "\t";
		cout << endl;
	}
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][M], int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < n; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < n && a[i + c][i] == 0)
			c++;
			if ((i + c) == n)
			{
				flag = 1;
				break;
			}
			for (j = i, k = 0; k <= n; k++)
			swap(a[j][k], a[j+c][k]);
		}

		for (j = 0; j < n; j++)
		{
			// Excluding all i == j
			if (i != j)
			{
				// Converting Matrix to reduced row echelon
				    form(diagonal matrix)
				float pro = a[j][i] / a[i][i];
				for (k = 0; k <= n; k++)
```

```
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

int PerformOperation2(float a[][M], int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again
        for (i = 2; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 0 && a[i-1][j] !=0)
                        {
                                float pivot = a[i-1][j] / a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}

// Function to print the desired result
// if unique solutions exists, otherwise
// prints no solution or infinite solutions
// depending upon the input given.
void PrintResult(float a[][M], int n, int flag)
{
        cout << "The solution/s : ";

        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }
```

```
        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "x(" << i << ") = "<< a[i][n] / a[i][i] << ", ";
        }
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float a[][M], int n, int flag)
{
        int i, j;
        float sum;

        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < n; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        mat B;
        B.load("vectorB.txt");
        mat X;
        X = solve(A,B,solve_opts::force_approx);

        cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
        cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
        cout <<"Solution:" << "\n" << X <<endl;

        float b[M][M] = {{ 1,-3, 4,-2,5,4,0 },
                { 2,-6,9,-1,8,2,0 },
                { 2,-6,9,-1,9,7,0 },
                { -1,3,-4,2,-5,-4,0 }};
```

```
cout << "Homogeneous System : " << endl;
int r = 4;
int c = 7;
for (int i = 0; i < r; ++i)
{
       for (int j = 0; j < c; ++j)
       {
              cout <<setw(6) << b[i][j]<< "\t";
       }
       cout << endl;
}

// Order of Matrix(n)
int n = 7, flag = 0;

// Performing Matrix transformation
flag = PerformOperation(b, n);

if (flag == 1)
flag = CheckConsistency(b, n, flag);

// Printing Final Matrix
cout << endl;
cout << "Gaussian Elimination for Homogeneous System: " << endl;
PrintMatrix(b);
cout << endl;

// Printing Solutions for homogeneous system
PrintResult(b,n, flag);
cout << endl;

cout << "Row Echelon Form for Homogeneous System : " << endl;
flag = PerformOperation2(b, n);
PrintMatrix(b);

cout << "Basis for Column Space of Matrix A: " << endl;

for (int i = 0; i < r; ++i)
{
       for (int j = 0; j < c; ++j)
       {
              if ( b[i][j] !=0 )
              {
                     cout << "A.col(" << j << ") : " << endl << A.
                         col(j) << endl;
                     j = c-1;
              }
```

```
                }
                cout << endl;
            }
        }
```

**C++ Code 128:** *main.cpp "Basis for a Column Space by Row Reduction"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The code contains the computation if the matrix is a linear system of form $A\boldsymbol{x} = \boldsymbol{b}$ for convenient and future modification, but the main focus shall be located in the part where we make the matrix into a homogenenous linear system of form $A\boldsymbol{x} = \mathbf{0}$ where we will reduce the matrix $A$ into the row echelon form.

  We first create a new function **PerformOperation2** to perform another Gaussian elimination to make a new leading 1 at farther right at the new column after the first Gaussian elimination from the function **PerformOperation**.

  We put the command **j = n-1** to stop the computation once we have found entry in matrix that is nonzero at the same column between the current row and the previous row, then do the row operation on the current row; so there will be only 0 below the leading 1 of each row.

```
int PerformOperation2(float a[][M], int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again
        for (i = 2; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 0 && a[i-1][j] !=0)
                        {
                                float pivot = a[i-1][j] / a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][
                                            k] * pivot);
                                }
                                j = n-1;
```

```
                        }
                }
        }
        return flag;
}
```

- To show the basis for column space of matrix $A$, we simply search for the first entry that become the leading 1's at each row of matrix $R$ and then take the column number and show it with function in Armadillo **A.col(j)** to show column $j$-th from matrix $A$ that correspond to the basis for the column space of $R$.

  We end the loop with $j = c - 1$ after we find the leading one at each row of matrix $R$, this is the method to find the basis for the column space of $R$. We only show the basis for the column space of $A$, you can modify it a bit to show the basis for the column space of $R$.

```
cout << "Row Echelon Form for Homogeneous System : " << endl;
flag = PerformOperation2(b, n);
PrintMatrix(b);

cout << "Basis for Column Space of Matrix A: " << endl;

for (int i = 0; i < r; ++i)
{
        for (int j = 0; j < c; ++j)
        {
                if ( b[i][j] !=0 )
                {
                        cout << "A.col(" << j << ") : " << endl <<
                            A.col(j) << endl;
                        j = c-1;
                }
        }
        cout << endl;
}
```

```
Homogeneous System :
     1      -3       4      -2       5       4       0
     2      -6       9      -1       8       2       0
     2      -6       9      -1       9       7       0
    -1       3      -4       2      -5      -4       0

Gaussian Elimination for Homogeneous System:
     1      -3       4      -2       5       4       0
     0       0       1       3      -2      -6       0
     0       0       1       3      -1      -1       0
     0       0       0       0       0       0       0

The solution/s : Infinite Solutions Exists

Row Echelon Form for Homogeneous System :
     1      -3       4      -2       5       4       0
     0       0       1       3      -2      -6       0
     0       0       0       0       1       5       0
     0       0       0       0       0       0       0
Basis for Column Space of Matrix A:
A.col(0) :
   1.0000
   2.0000
   2.0000
  -1.0000


A.col(2) :
   4.0000
   9.0000
   9.0000
  -4.0000


A.col(4) :
   5.0000
   8.0000
   9.0000
  -5.0000
```

**Figure 23.54:** *The computation to compute and find basis for a column space of matrix A with C++ (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Basis for a Column Space by Row Reduction with Armadillo/main.cpp**).*

## vi. C++ Computation: Compute Basis for the Row and Column Space by Row Reduction

At the previous section we are able to compute the basis for the column space of the matrix $A$. Now, we will use C++ to find a basis for the row space of the matrix

$$A = \begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 2 & -6 & 9 & -1 & 8 & 2 \\ 2 & -6 & 9 & -1 & 9 & 7 \\ -1 & 3 & -4 & 2 & -5 & -4 \end{bmatrix}$$

with Gaussian elimination we can reduce the matrix above into the row echelon form

$$R = \begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 0 & 0 & 1 & 3 & -2 & -6 \\ 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The nonzero row vectors of $R$ form a basis for the row space of both $R$ and $A$. These basis vectors are

$$\boldsymbol{r}_1 = \begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \end{bmatrix}$$
$$\boldsymbol{r}_2 = \begin{bmatrix} 0 & 0 & 1 & 3 & -2 & -6 \end{bmatrix}$$
$$\boldsymbol{r}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 5 \end{bmatrix}$$

it is easier to find basis for the row space for matrix $A$, since elementary row operations do not alter its row space.

For the C++ code, we are supplying both the column vectors that form a basis for the column space of $A$ and the row vectors that form a basis for the row space of $A$. We still use **Armadillo** combined with function to do elementary row operations to obtain the reduced row echelon form of matrix $A$.

If the previous section needs the matrix $A$ to be manually typed, in this section another adding is we are taking the inputs as row vectors $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4$ that are named **vectorV.txt**, **vectorW.txt**, **vectorX.txt**, and **vectorY.txt** respectively.

At the end of **int main()**, we also try to do row operations on $A^T$ for this case, but does not result in the best final result, still need final touch to make a perfect reduced row echelon form for $A^T$ here.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
```

```
using namespace arma;

const int N = 6; // Define the number of column for A, the amount of data
    in each row vector

#define C_Mattranspose 4 // Define the number of column for A^T
#define R_Mattranspose 6 // Define the number of row for A^T
#define C_Mat 6 // Define the number of column for A
#define R_Mat 4 // Define the number of row for A

// Function to print the matrix

void PrintMatrixC(float a[][6])
{
       int r = 4;
       int c = 6;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << a[i][j] << "\t";
              cout << endl;
       }
}
void PrintMatrixTranspose(float a[][C_Mattranspose])
{
       int r = 6;
       int c = 4;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << a[i][j] << "\t";
              cout << endl;
       }
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][C_Mat], int r, int n)
{
       int i, j, k = 0, c, flag = 0, m = 0;
       float pro = 0;

       // Performing elementary operations
       for (i = 0; i < r; i++)
       {
              if (a[i][i] == 0)
              {
                     c = 1;
                     while ((i + c) < r && a[i + c][i] == 0)
```

```
                        c++;
                        if ((i + c) == r)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row echelon
                                    form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

int PerformOperation2(float a[][C_Mat], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 0 && a[i-1][j] !=0)
                        {
                                float pivot = a[i-1][j] / a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                j = n-1;
                        }
                }
        }
```

```
        return flag;
}

int PerformOperationAT(float a[][C_Mattranspose], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pro = 0;

        // Performing elementary operations
        for (i = 0; i <= r; i++)
        {
                if (a[i][i] == 0)
                {
                        c = 1;
                        while ((i + c) < r && a[i + c][i] == 0)
                        c++;
                        if ((i + c) == r)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j <= n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

int PerformOperationAT2(float a[][C_Mattranspose], int m, int n)
{
        int flag = 0;
        float pivot = 0;

        // Performing elementary operations again to make leading 1 in every
             rows
```

```
        for (int i = 1; i < m; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        if (a[i][j] !=1 && a[i][j] !=0)
                        {
                                float pivot = a[i][j];
                                for (int k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] / pivot;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}

int PerformOperationAT3(float a[][C_Mattranspose], int m, int n)
{
        int flag = 0;
        float pivot = 0;

        // Performing elementary operations again to delete linearly
            dependent row
        for (int i = 0; i < m; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        if (a[i-1][j] == 1 && a[i][j] !=0 )
                        {
                                float pivot = a[i-1][j]/a[i][j];
                                for (int k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}

float* vecv() {
        static float v[N];
        std::fstream in("vectorV.txt");
        float vectortiles[N];
```

```
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}

float* vecw() {
        static float w[N];
        std::fstream in("vectorW.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                w[i] =vectortiles[i];
        }
        return w;
}

float* vecx() {
        static float x[N];
        std::fstream in("vectorX.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

float* vecy() {
        static float y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}

// Driver code
int main(int argc, char** argv)
{
        float* ptrvecv;
```

```
ptrvecv = vecv();
float* ptrvecw;
ptrvecw = vecw();
float* ptrvecx;
ptrvecx = vecx();
float* ptrvecy;
ptrvecy = vecy();

arma::mat A(R_Mat,C_Mat,fill::zeros); //declare matrix A with size
    of R X C all have 0 entries with Armadillo.
arma::mat AGauss(R_Mat,C_Mat,fill::zeros); //declare matrix AGauss
    with size of R X C all have 0 entries with Armadillo.
arma::mat A_transpose(C_Mat,R_Mat,fill::zeros); //declare matrix
    A_transpose with size of C X R all have 0 entries with Armadillo
    .

float a[R_Mat][C_Mat] = {};
float c[R_Mat][C_Mat] = {};
for (int j = 0; j < C_Mat; ++j)
{
       for(int i = 0; i < R_Mat; ++i)
       {
              if (i==0)
              {
                     c[i][j] = ptrvecv[j];
                     a[i][j] = ptrvecv[j];
              }
              if (i==1)
              {
                     c[i][j] = ptrvecw[j];
                     a[i][j] = ptrvecw[j];
              }
              if (i==2)
              {
                     c[i][j] = ptrvecx[j];
                     a[i][j] = ptrvecx[j];
              }
              if (i==3)
              {
                     c[i][j] = ptrvecy[j];
                     a[i][j] = ptrvecy[j];
              }
       }
}
cout << " A : " << endl;
PrintMatrixC(c);

// Fill matrix A from Armadillo with matrix c[4][6] from loading
```

```
    textfile of row vectors v,w,x,y
for (int i = 0; i < R_Mat; ++i)
{
       for(int j = 0; j<C_Mat; ++j)
       {
              A[i+j*R_Mat] = c[i][j];
       }
}

A_transpose = A.t();
A_transpose.print("A^T: (with Armadillo)");
cout << endl;
cout << endl;

float a_transpose[C_Mat][R_Mat] = {};
for (int i = 0; i < C_Mat; ++i)
{
       for(int j = 0; j<R_Mat; ++j)
       {
              a_transpose[i][j] = a[j][i];
       }
}

int flag = 0;

// Performing row operations to reduced row echelon for A
PerformOperation(c, R_Mat, C_Mat);
// Performing row operations again to make leading 1's in all rows
PerformOperation2(c, R_Mat, C_Mat);
// Performing row operations again to delete linearly dependent rows
//PerformOperationA3(a, R_Mat, C_Mat);

cout << " A in row echelon form: " << endl;
PrintMatrixC(c);
cout <<endl;

// Save the matrix A that has become reduced row echelon into
    Armadillo Matrix A_gauss
for (int i = 0; i < R_Mat; ++i)
{
       for(int j = 0; j<C_Mat; ++j)
       {
              AGauss[i+j*R_Mat] = c[i][j];
       }
}
cout <<endl;
AGauss.print("A in row echelon form with Armadillo:");
cout << endl;
```

```
arma::mat BC(4,0, fill::ones); // Declare matrix B to store the
    basis vectors with 4 rows and 0 columns
arma::mat BR(0,6, fill::ones); // Declare matrix B to store the
    basis vectors with 0 rows and 6 columns

cout << " Basis for the row space of A : " << endl;
for (int i = 0; i < R_Mat; ++i)
{
       for (int j = 0; j < C_Mat; ++j)
       {
              if ( c[i][j] !=0 )
              {
                     cout << "A.row(" << i << ") : " << endl <<
                         AGauss.row(i) << endl;
                     BR.insert_rows(i,AGauss.row(i)); // AGauss.row(
                         i) will be added to row i in matrix BR
                     j = C_Mat-1;
              }
       }
       cout << endl;
}
cout << " Basis for the column space of A : " << endl;
for (int i = 0; i < R_Mat; ++i)
{
       for (int j = 0; j < C_Mat; ++j)
       {
              if ( c[i][j] !=0 )
              {
                     cout << "A.col(" << j << ") : " << endl << A.
                         col(j) << endl;
                     BC.insert_cols(i,A.col(j)); // A.col(j) will be
                          added to column i in matrix BC
                     j = C_Mat-1;
              }
       }
       cout << endl;
}

BR.print("BR (Matrix with Basis Vectors for Row Space of A):");
cout << endl;
BC.print("BC (Matrix with Basis Vectors for Column Space of A):");
cout << endl;

// Performing row operations to reduced row echelon for A^T
// Still not successful to become reduced row echelon

//PrintMatrixTranspose(a_transpose); // Working really great.
```

```
            cout << endl;
            PerformOperationAT(a_transpose, R_Mattranspose, C_Mattranspose);
            // Performing row operations again to make leading 1's in all rows
            PerformOperationAT2(a_transpose, R_Mattranspose, C_Mattranspose);
            // Performing row operations again to delete linearly dependent rows
            PerformOperationAT3(a_transpose, R_Mattranspose, C_Mattranspose);

            //PrintMatrixTranspose(a_transpose);
            cout << endl;
        }
```

**C++ Code 129:** *main.cpp "Basis for the Row and Column Space by Row Reduction"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
 A :
     1      -3       4      -2       5       4
     2      -6       9      -1       8       2
     2      -6       9      -1       9       7
    -1       3      -4       2      -5      -4
A^T: (with Armadillo)
   1.0000   2.0000   2.0000  -1.0000
  -3.0000  -6.0000  -6.0000   3.0000
   4.0000   9.0000   9.0000  -4.0000
  -2.0000  -1.0000  -1.0000   2.0000
   5.0000   8.0000   9.0000  -5.0000
   4.0000   2.0000   7.0000  -4.0000


 A in row echelon form:
     1      -3       4      -2       5       4
     0       0       1       3      -2      -6
     0       0       0       0       1       5
     0       0       0       0       0       0


A in row echelon form with Armadillo:
   1.0000  -3.0000   4.0000  -2.0000   5.0000   4.0000
        0        0   1.0000   3.0000  -2.0000  -6.0000
        0        0        0        0   1.0000   5.0000
        0        0        0        0        0        0

 Basis for the row space of A :
A.row(0) :
   1.0000  -3.0000   4.0000  -2.0000   5.0000   4.0000


A.row(1) :
        0        0   1.0000   3.0000  -2.0000  -6.0000


A.row(2) :
        0        0        0        0   1.0000   5.0000
```

```
 Basis for the column space of A :
A.col(0) :
   1.0000
   2.0000
   2.0000
  -1.0000


A.col(2) :
   4.0000
   9.0000
   9.0000
  -4.0000


A.col(4) :
   5.0000
   8.0000
   9.0000
  -5.0000



BR (Matrix with Basis Vectors for Row Space of A):
   1.0000  -3.0000   4.0000  -2.0000   5.0000   4.0000
        0        0   1.0000   3.0000  -2.0000  -6.0000
        0        0        0        0   1.0000   5.0000

BC (Matrix with Basis Vectors for Column Space of A):
   1.0000   4.0000   5.0000
   2.0000   9.0000   8.0000
   2.0000   9.0000   9.0000
  -1.0000  -4.0000  -5.0000
```

**Figure 23.55:** *The computation to compute and find basis for the row and column space of matrix A with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Basis for a Column Space by Row Reduction with Armadillo/main.cpp**).*

## vii. C++ Computation: Compute Basis for a Row Space by Row Reduction and Transposing Matrix

Suppose we have a matrix $A$ of size $r \times c$, in order to find the basis vectors for the row space we can do row operations to obtain reduced row echelon form of matrix $A$ and then take the row with leading 1's and the corresponding row on matrix $A$ are the basis vectors for the row space.

Now we will try to transpose $A$ and then the basis for the column space of $A^T$ are the basis for the row space of $A$. Why we need to transpose a matrix and find the basis for column space instead of just make the reduced row echelon of $A$ directly? Sometimes it is easier and the computation is cheaper to make a reduced row echelon out of $A^T$ compared to make a reduced row echelon out of $A$. So this trick can be handy sometimes.

Find a basis for the row space of

$$A = \begin{bmatrix} 1 & -2 & 0 & 0 & 3 \\ 2 & -5 & -3 & -2 & 6 \\ 0 & 5 & 15 & 10 & 0 \\ 2 & 6 & 18 & 8 & 6 \end{bmatrix}$$

consisting entirely of row vectors from $A$.

**Solution:**

We will transpose $A$, thereby converting the row space of $A$ into the column space of $A^T$ to find a basis for the column space of $A^T$; and then we will transpose again to convert column vectors back to row vectors. Transposing $A$ yields

$$A^T = \begin{bmatrix} 1 & 2 & 0 & 2 \\ -2 & -5 & 5 & 6 \\ 0 & -3 & 15 & 18 \\ 0 & -2 & 10 & 8 \\ 3 & 6 & 0 & 6 \end{bmatrix}$$

Reducing this matrix to row echelon form yields

$$\begin{bmatrix} 1 & 0 & 10 & 22 \\ 0 & 1 & -5 & -10 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The first, second and fourth columns contain the leading 1's, so the corresponding column vectors in $A^T$ form a basis for the column space of $A^T$; these are

$$\boldsymbol{c}_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ 0 \\ 3 \end{bmatrix}, \quad \boldsymbol{c}_2 = \begin{bmatrix} 2 \\ -5 \\ -3 \\ -2 \\ 6 \end{bmatrix}, \quad \boldsymbol{c}_4 = \begin{bmatrix} 2 \\ 6 \\ 18 \\ 8 \\ 6 \end{bmatrix}$$

Transposing again and adjusting the notation appropriately yields the basis vectors

$$\boldsymbol{r}_1 = \begin{bmatrix} 1 & -2 & 0 & 0 & 3 \end{bmatrix}, \quad \boldsymbol{r}_2 = \begin{bmatrix} 2 & -5 & -3 & -2 & 6 \end{bmatrix}$$

$$\boldsymbol{r}_4 = \begin{bmatrix} 2 & 6 & 18 & 8 & 6 \end{bmatrix}$$

for the row space of $A$.

For the C++ code, we are using **Armadillo** to help for the computation and showing the matrix nicely, the inputs are row vectors $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4$ that are named **vectorV.txt**, **vectorW.txt**, **vectorX.txt**, and **vectorY.txt** respectively. For the row operations to become reduced row echelon we still depend on our own function that we create **PerformOperation**, **PerformOperation2**, and **PerformOperation3**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

const int N = 5; // Define the number of column for A, the amount of data
    in each row vector
const int R = 4; // number of rows for A
const int C = 5; // number of column for A

#define C_Mattranspose 4 // Define the number of column for A^T
#define R_Mattranspose 5 // Define the number of row for A^T
#define N_Mat 5 // Define the number of column for A

// Function to print the matrix
void PrintMatrix(float a[][N_Mat])
{
       int r = 4;
       int c = 5;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << a[i][j] << "\t";
              cout << endl;
       }
}
void PrintMatrixTranspose(float a[][C_Mattranspose])
{
       int r = 5;
       int c = 4;
       for (int i = 0; i < r; i++)
```

```
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << a[i][j] << "\t";
                cout << endl;
        }
}

int PerformOperation(float a[][C_Mattranspose], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pro = 0;

        // Performing elementary operations
        for (i = 0; i <= r; i++)
        {
                if (a[i][i] == 0)
                {
                        c = 1;
                        while ((i + c) < r && a[i + c][i] == 0)
                        c++;
                        if ((i + c) == r)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j <= n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

int PerformOperation2(float a[][C_Mattranspose], int m, int n)
{
        int flag = 0;
```

```
        float pivot = 0;

        // Performing elementary operations again to make leading 1 in every
             rows
        for (int i = 1; i < m; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        if (a[i][j] !=1 && a[i][j] !=0)
                        {
                                float pivot = a[i][j];
                                for (int k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] / pivot;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}

int PerformOperation3(float a[][C_Mattranspose], int m, int n)
{
        int flag = 0;
        float pivot = 0;

        // Performing elementary operations again to delete linearly
            dependent row
        for (int i = 0; i < m; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        if (a[i-1][j] == 1 && a[i][j] !=0 )
                        {
                                float pivot = a[i-1][j]/a[i][j];
                                for (int k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}
```

```
float* vecv() {
        static float v[N];
        std::fstream in("vectorV.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}

float* vecw() {
        static float w[N];
        std::fstream in("vectorW.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                w[i] =vectortiles[i];
        }
        return w;
}

float* vecx() {
        static float x[N];
        std::fstream in("vectorX.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

float* vecy() {
        static float y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}
```

```
// Driver code
int main(int argc, char** argv)
{
        float* ptrvecv;
        ptrvecv = vecv();
        float* ptrvecw;
        ptrvecw = vecw();
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat A(R,C,fill::zeros);
        arma::mat A_transpose(C,R,fill::zeros);
        arma::mat A_gauss(C,R,fill::zeros);

        float a[R][C] = {};
        for (int j = 0; j < C; ++j)
        {
                for(int i = 0; i < R; ++i)
                {
                        if (i==0)
                        {
                                a[i][j] = ptrvecv[j];
                        }
                        if (i==1)
                        {
                                a[i][j] = ptrvecw[j];
                        }
                        if (i==2)
                        {
                                a[i][j] = ptrvecx[j];
                        }
                        if (i==3)
                        {
                                a[i][j] = ptrvecy[j];
                        }
                }
        }

        // Fill matrix A from Armadillo with matrix a[R][C] from loading
            textfile of row vectors v,w,x,y
        for (int i = 0; i < R; ++i)
        {
                for(int j = 0; j<C; ++j)
                {
                        A[i+j*R] = a[i][j];
                }
```

```
        }

        A_transpose = A.t();
        float a_transpose[C][R] = {};
        A.print("A:");
        cout << endl;
        cout <<"A (in simpler form): " << endl;
        PrintMatrix(a);
        cout << endl;
        A_transpose.print("A^T:");
        cout << endl;
        cout << endl;

        int flag = 0;

        // Transposing matrix A manually
        for (int i = 0; i < C; ++i)
        {
               for(int j = 0; j<R; ++j)
               {
                      a_transpose[i][j] = a[j][i];
               }
        }
        cout << " A^T (in simpler form): " << endl;
        PrintMatrixTranspose(a_transpose);
        cout << endl;

        // Performing row operations to reduced row echelon
        PerformOperation(a_transpose, R_Mattranspose, C_Mattranspose);
        // Performing row operations again to make leading 1's in all rows
        PerformOperation2(a_transpose, R_Mattranspose, C_Mattranspose);
        // Performing row operations again to delete linearly dependent rows
        PerformOperation3(a_transpose, R_Mattranspose, C_Mattranspose);

        // Save the transposed matrix that has become reduced row echelon
            into Armadillo Matrix A_gauss
        for (int i = 0; i < C; ++i)
        {
               for(int j = 0; j<R; ++j)
               {
                      A_gauss[i+j*C] = a_transpose[i][j];
               }
        }
        cout << " A^T in row echelon form: " << endl;
        PrintMatrixTranspose(a_transpose);
        cout <<endl;
        A_gauss.print("A^T in row echelon form with Armadillo:");
        cout << endl;
```

```
        arma::mat B(5,0, fill::ones);

        cout << " Basis for the column space of A^T : " << endl;
        for (int i = 0; i < R_Mattranspose; ++i)
        {
            for (int j = 0; j < C_Mattranspose; ++j)
            {
                if ( a_transpose[i][j] !=0 )
                {
                    cout << "A^T.col(" << j << ") : " << endl <<
                        A_transpose.col(j) << endl;
                    B.insert_cols(i,A_transpose.col(j));
                    j = C_Mattranspose-1;
                }
            }
            cout << endl;
        }
        B.print("B (Matrix with Basis Vectors for row Space of A):");
    }
```

**C++ Code 130:** *main.cpp "Basis for a Row Space by Row Reduction and Matrix Transpose"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Define the matrix that we are going to manipulate with Armadillo, the matrix $A$, $A^T$, and $A_gauss$. Then we input manually matrix of size $R \times C$ from the textfiles as row vectors into $a[R][C]$. The matrix $A$ we define as Armadillo' matrix can then get input of entries from $a[R][C]$, with this we can easily use Armadillo capabilities and function to do computation and manipulate $A$.

```
        arma::mat A(R,C,fill::zeros);
        arma::mat A_transpose(C,R,fill::zeros);
        arma::mat A_gauss(C,R,fill::zeros);

        float a[R][C] = {};
        for (int j = 0; j < C; ++j)
        {
            for(int i = 0; i < R; ++i)
            {
                if (i==0)
                {
```

```
                        a[i][j] = ptrvecv[j];
                }
                if (i==1)
                {
                        a[i][j] = ptrvecw[j];
                }
                if (i==2)
                {
                        a[i][j] = ptrvecx[j];
                }
                if (i==3)
                {
                        a[i][j] = ptrvecy[j];
                }
        }
}
for (int i = 0; i < R; ++i)
{
        for(int j = 0; j<C; ++j)
        {
                A[i+j*R] = a[i][j];
        }
}
```

- To create a matrix that consisting of the basis vectors for the column space of $A^T$ we will first declare a matrix $B$ of size $5 \times 0$, why 0? Because we want to add the first column till the last column from the basis vectors of $A^T$. The first time we input a column from **B.insert_cols(i,A_transpose.col(j))** will goes to the first column index that is the reason we set it to 0.

```
arma::mat B(5,0, fill::ones);

cout << " Basis for the column space of A^T : " << endl;
for (int i = 0; i < R_Mattranspose; ++i)
{
        for (int j = 0; j < C_Mattranspose; ++j)
        {
                if ( a_transpose[i][j] !=0 )
                {
                        cout << "A^T.col(" << j << ") : " << endl
                            << A_transpose.col(j) << endl;
                        B.insert_cols(i,A_transpose.col(j));
                        j = C_Mattranspose-1;
                }
        }
        cout << endl;
}
B.print("B (Matrix with Basis Vectors for row Space of A):");
```

```
ranspose with Armadillo ]# ./main
A:
    1.0000   -2.0000         0         0    3.0000
    2.0000   -5.0000   -3.0000   -2.0000    6.0000
         0    5.0000   15.0000   10.0000         0
    2.0000    6.0000   18.0000    8.0000    6.0000

A (in simpler form):
     1      -2       0       0       3
     2      -5      -3      -2       6
     0       5      15      10       0
     2       6      18       8       6

A^T:
    1.0000    2.0000         0    2.0000
   -2.0000   -5.0000    5.0000    6.0000
         0   -3.0000   15.0000   18.0000
         0   -2.0000   10.0000    8.0000
    3.0000    6.0000         0    6.0000


 A^T (in simpler form):
     1       2       0       2
    -2      -5       5       6
     0      -3      15      18
     0      -2      10       8
     3       6       0       6

 A^T in row echelon form:
     1       0      10      22
    -0       1      -5     -10
    -0      -0      -0       1
     0       0       0       0
     0       0       0       0
```

```
A^T in row echelon form with Armadillo:
    1.0000         0   10.0000   22.0000
         0    1.0000   -5.0000  -10.0000
         0         0         0    1.0000
         0         0         0         0
         0         0         0         0

 Basis for the column space of A^T :
A^T.col(0) :
   1.0000
  -2.0000
        0
        0
   3.0000


A^T.col(1) :
   2.0000
  -5.0000
  -3.0000
  -2.0000
   6.0000


A^T.col(3) :
    2.0000
    6.0000
   18.0000
    8.0000
    6.0000
```

**Figure 23.56:** *The computation to compute and find basis for a column space of matrix $A^T$ that is equivalent as the basis vectors for the row space of $A$ with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Basis for the Row Space of a Matrix by Transpose with Armadillo/main.cpp**).*

## viii. C++ Computation: Row Space and Column Space Application in Material Science

We depend on material to create things that are around us today, like computer, smartphone, sensors, space shuttle, processor, vehicles, airplane, etc. Suppose we have to be realistic and know that we have limited ore and resources, thus we need to find a replacement for gold, for example, in the processor we want to construct. To find a material that have the same properties / attributes like gold will need a really nice technique in row space and column space department.

Let

$$A = \begin{bmatrix} \boldsymbol{c}_1 & \boldsymbol{c}_2 & \dots & \boldsymbol{c}_n \end{bmatrix}$$

the column vector $\boldsymbol{c}_i$ represent condition / goodness of material number $i$.

While

$$A = \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \vdots \\ \boldsymbol{r}_n \end{bmatrix}$$

the row vector $\boldsymbol{r}_i$ represents the treatment number $i$ given to all materials, for example $\boldsymbol{r}_1$ represents the heat given is $10^0$, $\boldsymbol{r}_2$ represents the heat given is $20^0$, etc.

Not only heat given, we can also give treatment such as giving pressure at the materials at constant increasing rate, or give treatment of changing humidity.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

thus, we can then define that $a_{ij}$ is treatment number $i$ given to material $j$. From this we can find the pattern, do interpolation, and able to decide which material can be a good replacement for gold with the power of Row Space and Column Space.

This can be of use too in Civil Engineering, when we want to create a building or a house, with limited budget, we can find a nice replacement for the paint that can last long / has high durability with lower price, or a replacement for a roof that can sustain bad weather for years.

The algorithm for testing:

1. Prepare the materials you want to test
2. Use a sensor that able to tell the quality of the material
3. Give treatment to all materials and record it with the sensor, you can capture the data and save it into textfile
4. The captured data can then become an input, you can choose to input as row vector or column vector, be consistent with your input.
5. You can try to manipulate the matrix, row vector or column vector, find bases of row spaces or column spaces, do elementary row operations, or do least square fitting to find what you need.

We prepare 4 textfiles, with dummy data of 4 rows, the row vectors represent:

$$A = \begin{bmatrix} \boldsymbol{r}_1 \rightarrow T = 20^0 \\ \boldsymbol{r}_2 \rightarrow T = 40^0 \\ \boldsymbol{r}_3 \rightarrow T = 60^0 \\ \boldsymbol{r}_4 \rightarrow T = 80^0 \end{bmatrix}$$

while the column vector:

1. $\boldsymbol{c}_1$ that is saved as **vectorV.txt** is material superconducting aluminum.
2. $\boldsymbol{c}_2$ that is saved as **vectorW.txt** is material gold.
3. $\boldsymbol{c}_3$ that is saved as **vectorX.txt** is material copper.
4. $\boldsymbol{c}_4$ that is saved as **vectorY.txt** is material bronze.

it is not real life data, just an example.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

const int N = 4;
const int R = 4; // number of rows
const int C = 4; // number of column

#define MN 4

// Function to print the matrix
void PrintMatrix(float a[][MN])
{
	int r = 4;
	int c = 4;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << a[i][j] << "\t";
		cout << endl;
	}
}

int PerformOperation(float a[][MN], int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < n; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < n && a[i + c][i] == 0)
			c++;
			if ((i + c) == n)
			{
				flag = 1;
				break;
			}
```

```
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] — (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}

float* vecv() {
        static float v[N];
        std::fstream in("vectorV.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                v[i] =vectortiles[i];
        }
        return v;
}

float* vecw() {
        static float w[N];
        std::fstream in("vectorW.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                w[i] =vectortiles[i];
        }
        return w;
}

float* vecx() {
        static float x[N];
        std::fstream in("vectorX.txt");
        float vectortiles[N];
```

```
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

float* vecy() {
        static float y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}

// Driver code
int main(int argc, char** argv)
{
        float* ptrvecv;
        ptrvecv = vecv();
        float* ptrvecw;
        ptrvecw = vecw();
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat A(R,C,fill::zeros); //declare matrix A with size of R X C
            all have 0 entries with Armadillo.
        arma::mat A_transpose(C,R,fill::zeros); //declare matrix A_transpose
             with size of C X R all have 0 entries with Armadillo.
        arma::mat A_gauss(R,C,fill::zeros); //declare matrix A after
            performing Gaussian elimination with size of R X C

        float a[R][C] = {};
        for (int i = 0; i < R; ++i)
        {
                for(int j = 0; j<C; ++j)
                {
                        if (j==0)
                        {
                                a[i][j] = ptrvecv[i];
                        }
```

```
                        if (j==1)
                        {
                                a[i][j] = ptrvecw[i];
                        }
                        if (j==2)
                        {
                                a[i][j] = ptrvecx[i];
                        }
                        if (j==3)
                        {
                                a[i][j] = ptrvecy[i];
                        }
                }
        }

        // Fill matrix A from Armadillo with matrix a[R][C] from loading
            textfile of vectors v,w,x,y
        for (int i = 0; i < R; ++i)
        {
                for(int j = 0; j<C; ++j)
                {
                        A[i+j*R] = a[i][j];
                }
        }

        A_transpose = A.t();

        A.print("A:");
        cout << endl;
        cout <<"A (in simpler form): " << endl;
        PrintMatrix(a);
        cout << endl;
        A_transpose.print("A^T:");
        cout << endl;
        cout << endl;

        int flag = 0;

        // Performing Matrix transformation
        PerformOperation(a, N);
        for (int i = 0; i < R; ++i)
        {
                for(int j = 0; j<C; ++j)
                {
                        A_gauss[i+j*R] = a[i][j];
                }
        }
        A_gauss.print("A in row echelon form:");
```

```
        cout << endl;

        // determinant and inverse from Armadillo
        cout << "det(A^T * A): " << det(A_transpose*A) << endl;
        cout << endl;
        cout << "inv(A): " << endl << arma::inv(A) << endl;

        cout <<endl;
    }
```

**C++ Code 131:** *main.cpp "Row Space and Column Space Application in Material Science"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

We can modify the code and add something to manipulate the matrix, row space, column space to achieve what we want to find out. The benefit of using **Armadillo** is that it can show the matrix tidier than using the function **PrintMatrix**, we are able to do computation for determinant and inverse with great precision when using **Armadillo** if we want to proceed to find curve fitting for the data.

```
A:
   1.0000e+02   1.0000e+02   1.0000e+02   1.0000e+02
   8.0000e+01   8.9000e+01   7.0600e+01   8.2000e+01
   7.1000e+01   8.1000e+01   5.8700e+01   4.5000e+01
   4.9000e+01   7.2000e+01   1.0400e+01   2.6000e+01

A (in simpler form):
   100     100     100     100
    80      89    70.6      82
    71      81    58.7      45
    49      72    10.4      26

A^T:
   1.0000e+02   8.0000e+01   7.1000e+01   4.9000e+01
   1.0000e+02   8.9000e+01   8.1000e+01   7.2000e+01
   1.0000e+02   7.0600e+01   5.8700e+01   1.0400e+01
   1.0000e+02   8.2000e+01   4.5000e+01   2.6000e+01


A in row echelon form:
   1.0000e+02            0            0            0
   8.8282e-44   9.0000e+00            0            0
  -4.2039e-45            0  -1.8556e+00   1.9073e-06
   1.4013e-45            0            0   1.9361e+02

det(A^T * A): 1.04542e+11

inv(A):
   0.3193  -0.3686  -0.1284   0.1566
  -0.2020   0.2229   0.0907  -0.0832
  -0.1081   0.1138   0.0782  -0.0786
   0.0008   0.0319  -0.0406   0.0052
```

**Figure 23.57:** *The manipulation of a matrix $A$ from input of column vectors $v, w, x, y$ that represent different materials condition, the row vector itself represent different heat given (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Row Space and Column Space Application/main.cpp**).*

## ix. C++ Computation: Rank and Nullity of a Matrix

Find the rank and nullity of the matrix

$$A = \begin{bmatrix} -1 & 2 & 0 & 4 & 5 & -3 \\ 3 & -7 & 2 & 0 & 1 & 4 \\ 2 & -5 & 2 & 4 & 6 & 1 \\ 4 & -9 & 2 & -4 & -4 & 7 \end{bmatrix}$$

The reduced row echelon form of $A$ is

$$R = \begin{bmatrix} 1 & 0 & -4 & -28 & -37 & 13 \\ 0 & 1 & -2 & -12 & -16 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since this matrix has two leading 1's, its row and column spaces are two dimensional and

$$\text{rank}(A) = 2$$

To find the nullity of $A$, we must find the dimension of the solution space of the linear system $A\boldsymbol{x} = \mathbf{0}$. This system can be solved by reducing its augmented matrix to reduced row echelon form. The resulting matrix will be identical to $R$ and the corresponding system of equations will be

$$\begin{aligned} x_1 - 4x_3 - 28x_4 - 37x_5 + 13x_6 &= 0 \\ x_2 - 2x_3 - 12x_4 - 16x_5 + 5x_6 &= 0 \end{aligned}$$

Solving these equations for the leading variables yields

$$\begin{aligned} x_1 &= 4x_3 + 28x_4 + 37x_5 - 13x_6 \\ x_2 &= 2x_3 + 12x_4 + 16x_5 - 5x_6 \end{aligned}$$

from which we obtain the general solution

$$\begin{aligned} x_1 &= 4r + 28s + 37t - 13u \\ x_2 &= 2r + 12s + 16t - 5u \\ x_3 &= r \\ x_4 &= s \\ x_5 &= t \\ x_6 &= u \end{aligned}$$

or in column vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = r \begin{bmatrix} 4 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} 28 \\ 12 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 37 \\ 16 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + u \begin{bmatrix} -13 \\ -5 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Because the four vectors on the right side form a basis for the solution space of $A\boldsymbol{x} = \mathbf{0}$ then

$$\text{nullity}(A) = 4$$

The C++ code able to compute till the we obtain the basis for the null space. The computation of rank can use Armadillo or manually by counting how many leading 1's in the reduced row echelon form of matrix $A$. The conversion into reduced row echelon form is a must, since this code depends heavily on a reduced row echelon form of matrix $A$ to be able to compute its' rank, nullity and basis for null space.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

#define C_MatA 6 // Define the number of column for the matrix A
#define R_MatA 4 // Define the number of row for the matrix A

// Function to print the matrix
void PrintMatrix(float a[][C_MatA])
{
	int r = R_MatA;
	int c = C_MatA;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(4) << a[i][j] << "\t";
		cout << endl;
	}
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][C_MatA], int r, int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < r; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < r && a[i + c][i] == 0)
```

```
                        c++;
                        if ((i + c) == r)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row echelon
                                    form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}
int PerformOperation2(float a[][C_MatA], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again to make leading 1's
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 1 && a[i][j] !=0)
                        {
                                float pivot = a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] / pivot;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}
```

```
int PerformOperation3(float a[][C_MatA], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again to make reduced row
            echelon form
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 0 && a[i-1][j] !=0)
                        {
                                if ( a[i-1][j] == -1 )
                                {
                                        for (k = 0; k <= n; k++)
                                        {
                                                a[i-1][k] = -1*(a[i-1][k]);
                                        }
                                }
                                float pivot = a[i-1][j]/a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                        }
                }
        }
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 0 && a[i-2][j] !=0 && a[i-1][j] == 0)
                        {
                                for (k = 0; k <= n; k++)
                                {
                                        swap(a[i][k], a[i-1][k]); // swap row
                                            that has zero entry in between two
                                            nonzero entries downward
                                }
                        }
                }
        }
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
```

```
                        if (a[i][j] != 0 && a[i-1][j] !=0)
                        {
                                float pivot = a[i-1][j]/a[i][j];
                                for (k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                //break;
                                j = n-1;
                                i = r-1;
                        }
                }

        }

        return flag;
}

int PerformOperation4(float a[][C_MatA], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;

        // Performing elementary operations again if the leading 1's is
            negative, turns it into positive
        for (i = 0; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] == -1)
                        {
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = -1*a[i][k];
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float a[][C_MatA], int r, int n, int flag)
{
        int i, j;
        float sum;
```

```
        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < r; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}


// Driver code
int main(int argc, char** argv)
{
        arma::mat A(R_MatA,C_MatA,fill::zeros); //declare matrix A with size
             of R_MatA X C_MatA all have 0 entries with Armadillo.
        A.load("matrixA.txt");

        cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;

        cout << endl;

        cout << "G****************Freya*****************" << endl;
        cout << endl;

        float matrixc[R_MatA][C_MatA] = {};
        for (int i = 0; i < R_MatA; ++i)
        {
                for(int j = 0; j<C_MatA; ++j)
                {
                        matrixc[i][j] = A[i+j*R_MatA] ;
                }
        }


        // Order of Matrix(n)
        int n = 6, flag = 0;

        // Performing Matrix transformation
        flag = PerformOperation(matrixc, R_MatA, C_MatA);

        if (flag == 1)
        flag = CheckConsistency(matrixc, R_MatA, C_MatA, flag);
```

```
        cout << endl;
        cout << "****************Row Echelon Form for matrix A
            ****************" << endl;
        cout << endl;

        PerformOperation2(matrixc, R_MatA, C_MatA);
        PerformOperation3(matrixc, R_MatA, C_MatA);
        PerformOperation2(matrixc, R_MatA, C_MatA);
        PerformOperation4(matrixc, R_MatA, C_MatA);

        PrintMatrix(matrixc);
        cout << endl;

        // To compute rank, dimension and nullity.
        int rankA = 0;
        for (int i = 0; i < R_MatA; ++i)
        {
               for(int j = 0; j<C_MatA; ++j)
               {
                      if (matrixc[i][j] == 1)
                      {
                             rankA = rankA + 1;
                      }
                      else
                      {
                             rankA = rankA;
                      }
               }
        }
        int nullity = C_MatA — rankA;

        cout << "rank: " << rankA <<endl;
        cout << endl;
        cout << "dimension: " << C_MatA <<endl;
        cout << endl;
        cout << "nullity: " << nullity <<endl;
        cout << endl;
        cout << "****************Basis for the Null Space of Matrix A
            ****************" << endl;
        cout << endl;

        // To compute the Basis for the Null Space of matrix A
        float matrixd[C_MatA][4] = {};

        for (int i = 0; i < rankA; i++)
        {
               for (int j = 0; j < nullity; j++)
               {
```

```
                matrixd[i][j] = -1*matrixc[i][j + rankA];
        }
    }
    for (int i = rankA; i < C_MatA; i++)
    {
        for (int j = 0; j < nullity; j++)
        {
            if (i-j == rankA)
            {
                matrixd[i][j] = 1;
            }
            else
            {
                matrixd[i][j] = 0;
            }
        }
    }

    for (int i = 0; i < C_MatA; i++)
    {
        for (int j = 0; j < nullity; j++)
        cout <<setw(6) << setprecision(4) << matrixd[i][j] << "\t";
        cout << endl;
    }

    arma::mat ANull(C_MatA,nullity,fill::zeros); //declare matrix ANull
        with size of C_MatA X nullity all have 0 entries with Armadillo.
    // Copy from matrixd to ANull
    for (int i = 0; i < C_MatA; ++i)
    {
        for(int j = 0; j<nullity; ++j)
        {
            ANull[i+j*C_MatA] = matrixd[i][j] ;
        }
    }
    //ANull.print("Null Basis :\n");
    cout << endl;
    cout << "Basis for Null Space of Matrix A: " << endl;

    for (int j = 0; j < nullity; ++j)
    {
        cout << "column(" << j << ") : " << endl << ANull.col(j) <<
            endl;
    }
    cout << endl;

    // Optional: Armadillo' computation
    /*
```

```
            cout <<"Rank of matrix A :" << arma::rank(A) << endl;
            cout << endl;

            cout <<"Nullity of matrix A :" << A.n_cols - arma::rank(A) << endl;
            cout << endl;
            cout <<"Orthonormal basis of Null space of matrix A :\n" << arma::
                null(A) << endl;
            cout << endl;
            */
      }
```

**C++ Code 132:** *main.cpp "Rank Nullity and Basis for Null Space"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- To determine / compute the basis for the null space, first we declare another 2-dimensional array / matrix with name **matrixd** with row of 6 and column of 4, we cannot put the column as variable of **nullity** it will trigger an error thus we write it this way **float matrixd[C_MatA][4]**.

  For the leading variables part: to adjust with the general solution we assign the negative of the reduced row echelon form of matrix $A$ with column being transitioned as much as the number fo the rank to this new **matrixd** with **matrixd[i][j] = -1*matrixc[i][j + rankA]**. This computation only for the first row till the row that has the last leading 1's / the number of **rank(**$A$**)**.

  The last thing we need to do concerning the free variables is that we add an identity matrix with size of the nullity / $I_{nullity}$ starting at the row **rank(**$A$**) + 1** of **matrixd**, we know that dimension is the number of column of matrix $A$, and the number of rank can be obtained after we can compute the reduced row echelon form of $A$, then we can get the number for the nullity and create the final **matrixd**.

  We then copy **matrixd** into $ANull$, a matrix that is defined with Armadillo' functions so we can do more modification later on from the matrix that contain the basis for null space of $A$.

```
            float matrixd[C_MatA][4] = {};

            for (int i = 0; i < rankA; i++)
            {
                  for (int j = 0; j < nullity; j++)
                  {
                        matrixd[i][j] = -1*matrixc[i][j + rankA];
```

```
                }
            }
            for (int i = rankA; i < C_MatA; i++)
            {
                for (int j = 0; j < nullity; j++)
                {
                    if (i-j == rankA)
                    {
                        matrixd[i][j] = 1;
                    }
                    else
                    {
                        matrixd[i][j] = 0;
                    }
                }
            }

            for (int i = 0; i < C_MatA; i++)
            {
                for (int j = 0; j < nullity; j++)
                cout <<setw(6) << setprecision(4) << matrixd[i][j] << "\
                    t";
                cout << endl;
            }

            arma::mat ANull(C_MatA,nullity,fill::zeros); //declare matrix
                ANull with size of C_MatA X nullity all have 0 entries with
                 Armadillo.
            // Copy from matrixd to ANull
            for (int i = 0; i < C_MatA; ++i)
            {
                for(int j = 0; j<nullity; ++j)
                {
                    ANull[i+j*C_MatA] = matrixd[i][j] ;
                }
            }
```

- The last codes below that we commented are optional, they are computation from Armadillo, if you want you can try it by yourself.

```
            cout <<"Rank of matrix A :" << arma::rank(A) << endl;
            cout << endl;

            cout <<"Nullity of matrix A :" << A.n_cols - arma::rank(A) <<
                endl;
            cout << endl;
            cout <<"Orthonormal basis of Null space of matrix A :\n" <<
                arma::null(A) << endl;
            cout << endl;
```

```
llo ]# ./main
Matrix A (Coefficient Matrix):
  -1.0000   2.0000        0   4.0000   5.0000  -3.0000
   3.0000  -7.0000   2.0000        0   1.0000   4.0000
   2.0000  -5.0000   2.0000   4.0000   6.0000   1.0000
   4.0000  -9.0000   2.0000  -4.0000  -4.0000   7.0000


G*****************Freya*****************


****************Row Echelon Form for matrix A****************

     1      -0      -4     -28     -37      13
     0       1      -2     -12     -16       5
    -0       0       0       0       0       0
     0       0       0       0       0       0

rank: 2

dimension: 6

nullity: 4

****************Basis for the Null Space of Matrix A****************

     4      28      37     -13
     2      12      16      -5
     1       0       0       0
     0       1       0       0
     0       0       1       0
     0       0       0       1
```

```
Basis for Null Space of Matrix A:
column(0) :
   4.0000
   2.0000
   1.0000
        0
        0
        0

column(1) :
   28.0000
   12.0000
         0
    1.0000
         0
         0

column(2) :
   37.0000
   16.0000
         0
         0
    1.0000
         0

column(3) :
  -13.0000
   -5.0000
         0
         0
         0
    1.0000
```

**Figure 23.58:** *The computation of a rank and nullity for matrix A and its basis for null space (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Rank and Nullity of a Matrix with Armadillo/main.cpp**).*

```
B:
[-1  2  0  4  5 -3]
[ 3 -7  2  0  1  4]
[ 2 -5  2  4  6  1]
[ 4 -9  2 -4 -4  7]


B after first operation to make zeroes under the first leading 1:
 1      -2       0      -4      -5       3
 0      -1/3     2/3     4      16/3    -5/3
 0      -1/2     1       6       8      -5/2
 0      -1/4     1/2     3       4      -5/4

B after second operation to make zeroes under the second leading 1:
 1      -2       0      -4      -5       3
 0      -1/3     2/3     4      16/3    -5/3
 0       0       0       0       0       0
 0       0       0       0       0       0
B after first operation to make zeroes above the second leading 1:
 1       0      -4      -28     -37     13
 0      -1/3     2/3     4      16/3    -5/3
 0       0       0       0       0       0
 0       0       0       0       0       0
B after operation to make leading 1's at the diagonal:
 1       0      -4      -28     -37     13
 0       1      -2      -12     -16      5
 0       0       0       0       0       0
 0       0       0       0       0       0
```

**Figure 23.59:** *The computation to obtain the reduced row echelon form of a matrix with Gauss-Jordan elimination (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Overdetermined Symbolic Matrix with SymbolicC++ Case 2/main.cpp**).*

## x. C++ Computation: Gauss-Jordan Elimination (Manual) for Overdetermined Symbolic Matrix

The linear system

$$\begin{aligned} x_1 &- 2x_2 &= b_1 \\ x_1 &- x_2 &= b_2 \\ x_1 &+ x_2 &= b_3 \\ x_1 &+ 2x_2 &= b_4 \\ x_1 &+ 3x_2 &= b_5 \end{aligned}$$

is overdetermined, so it cannot be consistent for all possible values of $b_1, b_2, b_3, b_4$, and $b_5$. Exact conditions under which the system is consistent can be obtained by solving the linear system by Gauss-Jordan elimination.

**Solution:**

The augmented matrix is row equivalent to

$$\begin{bmatrix} 1 & 0 & 2b_2 - b_1 \\ 0 & 1 & b_2 - b_1 \\ 0 & 0 & b_3 - 3b_2 + 2b_1 \\ 0 & 0 & b_4 - 4b_2 + 3b_1 \\ 0 & 0 & b_5 - 5b_2 + 4b_1 \end{bmatrix}$$

Thus, the system is consistent if and only if $b_1, b_2, b_3, b_4$, and $b_5$ satisfy the conditions

$$\begin{aligned} 2b_1 &- 3b_2 &+ b_3 & & & &= 0 \\ 3b_1 &- 4b_2 & + & &+ b_4 & &= 0 \\ 4b_1 &- 5b_2 & + & & & + b_5 &= 0 \end{aligned}$$

Solving this homogeneous linear system yields

$$
\begin{aligned}
b_1 &= 5r - 4s \\
b_2 &= 4r - 3s \\
b_3 &= 2r - s \\
b_4 &= r \\
b_5 &= s
\end{aligned}
$$

where $r$ and $s$ are arbitrary.

The C++ code in this section is still manual, in accordance to the fact that we create a lot of for loop, first is to make zeroes under the first leading 1, then another for loop to make zeroes under the second leading 1, then another for loop to make zero above the second leading 1 till we achieve the reduced row echelon form with Gauss-Jordan elimination.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 5 // number of rows
#define C 3 // number of columns

// Driver program
int main()
{
       Symbolic b1("b1");
       Symbolic b2("b2");
       Symbolic b3("b3");
       Symbolic b4("b4");
       Symbolic b5("b5");

       // Construct a symbolic matrix of size 5 X 3
       Matrix<Symbolic> B_mat(5,3);
       B_mat[0][0] = 1;  B_mat[0][1] = -2; B_mat[0][2] = b1;
       B_mat[1][0] = 1;  B_mat[1][1] = -1; B_mat[1][2] = b2;
       B_mat[2][0] = 1;  B_mat[2][1] = 1; B_mat[2][2] = b3;
       B_mat[3][0] = 1;  B_mat[3][1] = 2; B_mat[3][2] = b4;
       B_mat[4][0] = 1;  B_mat[4][1] = 3; B_mat[4][2] = b5;
       cout << "B:\n" << B_mat <<endl;
       cout << endl;

       // First operation to make the first leading 1 at the first row,
           first column
       for (int i = 0; i < R; i++)
       {
              for (int j = 0; j < C; j++)
              {
```

```
                        if (B_mat[0][0] != 1 && B_mat[0][0] !=0 )
                        {
                                Symbolic pivot = B_mat[0][0];
                                for (int k = 0; k < C; k++)
                                {
                                        B_mat[i][k] = B_mat[i][k] / pivot;
                                }
                        }
                        if (B_mat[0][0] == -1 )
                        {
                                for (int k = 0; k < C; k++)
                                {
                                        B_mat[i][k] = B_mat[i][k] * (-1) ;
                                }

                        }
                        break;
                }
        }

        // First operation of Gauss-Jordan elimination to make zeroes under
             the first leading 1
        for (int i = 1; i < R; i++)
        {
                for (int j = 0; j < C; j++)
                {
                        Symbolic pivot = B_mat[0][0]/B_mat[i][j];
                        for (int k = 0; k < C; k++)
                        {
                                B_mat[i][k] = (pivot*B_mat[i][k]) - B_mat[0][k
                                    ];
                        }
                        j = C-1;
                }
        }
        cout << "B after first operation to make zeroes under the first
            leading 1:" <<endl;
        // Show the matrix nicely
        for (int i = 0; i < R; i++)
        {
                for (int j = 0; j < C; j++)
                cout <<setw(2) << B_mat[i][j] << "\t";
                cout << endl;
        }
        cout <<endl;

        // Second operation of Gauss-Jordan elimination to make zeroes
            under the second leading 1
```

```
            for (int i = 2; i < R; i++)
            {
                   for (int j = 1; j < C; j++)
                   {
                          Symbolic pivot = B_mat[1][1]/B_mat[i][j];
                          for (int k = 0; k < C; k++)
                          {
                                 B_mat[i][k] = (pivot*B_mat[i][k]) - B_mat[1][k
                                     ];
                          }
                          j = C-1;
                   }
            }

            cout << "B after second operation to make zeroes under the second
                leading 1:" <<endl;

            for (int i = 0; i < R; i++)
            {
                   for (int j = 0; j < C; j++)
                   cout <<setw(2) << B_mat[i][j] << "\t";
                   cout << endl;
            }


            // First operation of Gauss-Jordan elimination to make zeroes above
                 the second leading 1
            for (int i = 1; i < R; i++)
            {
                   for (int j = 1; j < C; j++)
                   {
                          Symbolic pivot = B_mat[i-1][j]/B_mat[1][1];

                          for (int k = 0; k < C; k++)
                          {
                                 B_mat[i-1][k] = B_mat[i-1][k] - (pivot*B_mat[
                                     i][k]) ;
                          }
                          j = C-1;
                          i = R-1;
                   }
            }

            cout << "B after first operation to make zeroes above the second
                leading 1:" <<endl;

            for (int i = 0; i < R; i++)
            {
```

```
                for (int j = 0; j < C; j++)
                cout <<setw(2) << B_mat[i][j] << "\t";
                cout << endl;
        }

        return 0;
}
```

**C++ Code 133:** *main.cpp "Gauss-Jordan Elimination for Overdetermined Symbolic Matrix"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The first thing to do is to make the entry at the first row, first column into 1, then with forward elimination we make zeroes under the first leading 1.

```
        for (int i = 0; i < R; i++)
        {
                for (int j = 0; j < C; j++)
                {
                        if (B_mat[0][0] != 1 && B_mat[0][0] !=0 )
                        {
                                Symbolic pivot = B_mat[0][0];
                                for (int k = 0; k < C; k++)
                                {
                                        B_mat[i][k] = B_mat[i][k] / pivot;
                                }
                        }
                        if (B_mat[0][0] == -1 )
                        {
                                for (int k = 0; k < C; k++)
                                {
                                        B_mat[i][k] = B_mat[i][k] * (-1) ;
                                }

                        }
                        break;
                }
        }

        for (int i = 1; i < R; i++)
        {
                for (int j = 0; j < C; j++)
```

```
                {
                        Symbolic pivot = B_mat[0][0]/B_mat[i][j];
                        for (int k = 0; k < C; k++)
                        {
                                B_mat[i][k] = (pivot*B_mat[i][k]) - B_mat
                                    [0][k];
                        }
                        j = C-1;
                }
        }
```

- To become true reduced row echelon form the last step is to make zeroes above the second leading 1, then the third leading 1, till the last leading 1 before the last column / vector $\boldsymbol{b}$ from the linear system $A\boldsymbol{x} = \boldsymbol{b}$.

```
        for (int i = 1; i < R; i++)
        {
                for (int j = 1; j < C; j++)
                {
                        Symbolic pivot = B_mat[i-1][j]/B_mat[1][1];

                        for (int k = 0; k < C; k++)
                        {
                                B_mat[i-1][k] = B_mat[i-1][k] - (pivot*
                                    B_mat[i][k]) ;
                        }
                        j = C-1;
                        i = R-1;
                }
        }
```

```
B:
[ 1 -2 b1]
[ 1 -1 b2]
[ 1  1 b3]
[ 1  2 b4]
[ 1  3 b5]


B after first operation to make zeroes under the first leading 1:
 1      -2      b1
 0       1        b2-b1
 0       3        b3-b1
 0       4        b4-b1
 0       5        b5-b1

B after second operation to make zeroes under the second leading 1:
 1      -2      b1
 0       1        b2-b1
 0       0        1/3*b3+2/3*b1-b2
 0       0        1/4*b4+3/4*b1-b2
 0       0        1/5*b5+4/5*b1-b2
B after first operation to make zeroes above the second leading 1:
 1       0        -b1+2*b2
 0       1        b2-b1
 0       0        1/3*b3+2/3*b1-b2
 0       0        1/4*b4+3/4*b1-b2
 0       0        1/5*b5+4/5*b1-b2
```

**Figure 23.60:** *The computation to solve overdetermined system by Gauss-Jordan elimination (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Overdetermined Symbolic Matrix with SymbolicC++/main.cpp**).*

## xi. C++ Computation: Gauss-Jordan Elimination (Automatic) for Overdetermined Symbolic Matrix

The linear system

$$\begin{array}{rrrrrl} x_1 & +4x_2 & -2x_3 & +4x_4 & +5x_5 & =-3 \\ 3x_1 & -7x_2 & +2x_3 & & +x_5 & =4 \\ 2x_1 & -5x_2 & +2x_3 & +4x_4 & +6x_5 & =1 \\ 4x_1 & -9x_2 & +2x_3 & -4x_4 & -4x_5 & =7 \\ 14x_1 & +2x_2 & +2x_3 & +5x_4 & +x_5 & =12 \end{array}$$

is overdetermined, so it cannot be consistent for all possible values of $\boldsymbol{b} = (-3, 4, 1, 7, 12)$. Exact conditions under which the system is consistent can be obtained by solving the linear system by Gauss-Jordan elimination.

**Solution:**

The augmented matrix is row equivalent to

$$\begin{bmatrix} 1 & 0 & 0 & 4 & \frac{27}{5} & -\frac{7}{5} \\ 0 & 1 & 0 & 4 & \frac{26}{5} & -\frac{11}{5} \\ 0 & 0 & 1 & 8 & \frac{53}{5} & -\frac{18}{5} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Thus, the system is consistent if and only if it satisfies the conditions

$$\begin{array}{rrrrrl} x_1 & & & +4x_4 & +\frac{27}{5}x_5 & =-\frac{7}{5} \\ & x_2 & & +4x_4 & +\frac{26}{5}x_5 & =-\frac{11}{5} \\ & & x_3 & +8x_4 & +\frac{53}{5}x_5 & =-\frac{18}{5} \end{array}$$

Solving this homogeneous linear system yields

$$
\begin{aligned}
x_1 &= -\frac{7}{5} - 4r - \frac{27}{5}s \\
x_2 &= -\frac{11}{5} - 4r - \frac{26}{5}s \\
x_3 &= -\frac{18}{5} - 8r - \frac{53}{5}s \\
x_4 &= r \\
x_5 &= s
\end{aligned}
$$

where $r$ and $s$ are arbitrary.

Upgrading from the last section, this time we code less since we make the Gauss-Jordan elimination automatic. The coding still use **SymbolicC++** so it can come in handy, when we need to make symbolic computation we can just add the symbolic inside. This Symbolic declaration is also works even if the input is a float number.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 5 // number of rows
#define C 6 // number of columns

// Driver program
int main()
{
        Symbolic b1("b1"); // Don't mind this, only for example, you can
            replace numeric value below with this symbolic value.
        Symbolic b2("b2");
        Symbolic b3("b3");
        Symbolic b4("b4");
        Symbolic b5("b5");

        // Construct a symbolic matrix of size R X C
        Matrix<Symbolic> B_mat(R,C);
        B_mat[0][0] = 1;  B_mat[0][1] = 4; B_mat[0][2] = -2; B_mat[0][3] =
            4; B_mat[0][4] = 5; B_mat[0][5] = -3;
        B_mat[1][0] = 3;  B_mat[1][1] = -7; B_mat[1][2] = 2; B_mat[1][3] =
            0; B_mat[1][4] = 1; B_mat[1][5] = 4;
        B_mat[2][0] = 2;  B_mat[2][1] = -5; B_mat[2][2] = 2; B_mat[2][3] =
            4; B_mat[2][4] = 6; B_mat[2][5] = 1;
        B_mat[3][0] = 4;  B_mat[3][1] = -9; B_mat[3][2] = 2; B_mat[3][3] =
            -4; B_mat[3][4] = -4; B_mat[3][5] = 7;
        B_mat[4][0] = 14;  B_mat[4][1] = 2; B_mat[4][2] = 2; B_mat[4][3] =
            5; B_mat[4][4] = 1; B_mat[4][5] = 12;
```

```
cout << "B:\n" << B_mat <<endl;
cout << endl;

// First operation of Gauss—Jordan elimination to make zeroes under
     the first leading 1
for (int f = 1; f < R; f++)
{
       for (int i = f; i < R; i++)
       {

              for (int j = 0; j < C; j++)
              {
                     Symbolic pivot = B_mat[f—1][f—1]/B_mat[i][f
                         —1];

                     for (int k = 0; k < C; k++)
                     {
                            B_mat[i][k] = (pivot*B_mat[i][k]) —
                                B_mat[f—1][k];

                     }
                     j = C—1;
              }
       }
}

cout << "B after operation to make zeroes under all the leading 1's:
    " <<endl;

for (int i = 0; i < R; i++)
{
       for (int j = 0; j < C; j++)
       cout <<setw(2) << B_mat[i][j] << "\t";
       cout << endl;
}
cout << endl;

// Operation of Gauss—Jordan elimination to make zeroes above all
    leading 1's
for (int j = 1; j < C; j++)
{
       if (B_mat[j][j] == 0)
       {
              break;
       }
       else
       {
```

```
                        for (int i = 0; i < j; i++)
                        {
                                Symbolic pivot = B_mat[i][j]/B_mat[j][j];
                                //cout << "i : "<< i << endl;
                                //cout << "j : "<< j << endl;
                                //cout << "pivot: " << pivot << endl;
                                for (int k = 0; k < C; k++)
                                {
                                        B_mat[i][k] = B_mat[i][k] — (pivot*B_mat
                                            [j][k]) ;
                                }
                                /*
                                for (int i = 0; i < R; i++)
                                {
                                        for (int j = 0; j < C; j++)
                                        cout <<setw(2) << B_mat[i][j] << "\t";
                                        cout << endl;
                                }
                                */
                        }
                }
        }

        cout << "B after operation to make zeroes above all the leading 1's
            :" <<endl;

        for (int i = 0; i < R; i++)
        {
                for (int j = 0; j < C; j++)
                cout <<setw(2) << B_mat[i][j] << "\t";
                cout << endl;
        }
        cout << endl;

        // Operation to make leading 1's all 1 at the diagonal
        for (int i = 0; i < R; i++)
        {
                if (B_mat[i][i] !=1)
                {
                        Symbolic pivot = B_mat[i][i];
                        for (int k = 0; k < C; k++)
                        {
                                B_mat[i][k] = B_mat[i][k] / pivot ;
                        }
                }
        }

        cout << "B after operation to make leading 1's at the diagonal:" <<
```

```
                endl;

            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                cout <<setw(2) << B_mat[i][j] << "\t";
                cout << endl;
            }
            cout << endl;
            return 0;
        }
```

**C++ Code 134:** *main.cpp "Automatic Gauss-Jordan Elimination for Overdetermined Linear System"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The first step in Gauss-Jordan elimination, to make zeroes below all leading 1's, leading 1's will normally be located in the diagonal of the matrix. The code is simple enough as it is forward elimination method.

```
            for (int f = 1; f < R; f++)
            {
                for (int i = f; i < R; i++)
                {

                    for (int j = 0; j < C; j++)
                    {
                        Symbolic pivot = B_mat[f-1][f-1]/B_mat[i
                            ][f-1];

                        for (int k = 0; k < C; k++)
                        {
                            B_mat[i][k] = (pivot*B_mat[i][k]) -
                                B_mat[f-1][k];

                        }
                        j = C-1;
                    }
                }
            }
```

- For the Gauss-Jordan elimination' step when we make zeroes above the leading 1's, we are taking the pivot as

$$\text{pivot} = \frac{B[i][j]}{B[j][j]}$$

with $B[j][j]$ will be the diagonal entry, the place where the leading 1's should be located in all kind of standard matrix, then we use for loop with $j = 1, 2, \ldots, C$ and a for loop inside that with $i = 0, 1, 2, \ldots, j$, $C$ is the number of column.

So we are making zeroes from (column 2, row 1), then toward (column 3, row 1) and then (column 3, row 2), then moving again to (column 4, row 1), then (column 4, row 2), then (column 4, row 3) till the last will be (column $m$, row $m - 1$). Beware that indexing of matrix entry in C++ is 0 to indicate the first row.

With this step, Gauss-Jordan is one step beyond Gaussian elimination, since we do not need to do backward substitution to obtain the solution for $x_1, x_2, \ldots, x_n$, given we are able to make leading 1's until column $C - 1$, knowing the column number $C$ is reserved for $\boldsymbol{b}$, thus we can obtain all solutions directly.

We can uncomment the commands to see the step by step transition to proceed to the final matrix.

```
for (int j = 1; j < C; j++)
{
        if (B_mat[j][j] == 0)
        {
                break;
        }
        else
        {
                for (int i = 0; i < j; i++)
                {
                        Symbolic pivot = B_mat[i][j]/B_mat[j][j];
                        //cout << "i : "<< i << endl;
                        //cout << "j : "<< j << endl;
                        //cout << "pivot: " << pivot << endl;
                        for (int k = 0; k < C; k++)
                        {
                                B_mat[i][k] = B_mat[i][k] - (pivot*
                                    B_mat[j][k]) ;
                        }
                        /*
                        for (int i = 0; i < R; i++)
                        {
                                for (int j = 0; j < C; j++)
                                cout <<setw(2) << B_mat[i][j] << "\t
                                    ";
                                cout << endl;
                        }
```

```
                                    */
                                }
                            }
                        }
```

```
B:
[ 1  4 -2  4  5 -3]
[ 3 -7  2  0  1  4]
[ 2 -5  2  4  6  1]
[ 4 -9  2 -4 -4  7]
[14  2  2  5  1 12]


B after operation to make zeroes under all the leading 1's:
 1        4       -2        4        5       -3
 0       -19/3     8/3     -4       -14/3    13/3
 0        0       10/39    80/39    106/39   -12/13
 0        0        0        0        0        0
 0        0        0        0        0        0

B after operation to make zeroes above all the leading 1's:
 1        0        0        4       27/5     -7/5
 0       -19/3     0       -76/3    -494/15  209/15
 0        0       10/39    80/39    106/39   -12/13
 0        0        0        0        0        0
 0        0        0        0        0        0

B after operation to make leading 1's at the diagonal:
 1        0        0        4       27/5     -7/5
 0        1        0        4       26/5     -11/5
 0        0        1        8       53/5     -18/5
 0        0        0        0        0        0
 0        0        0        0        0        0
```

**Figure 23.61:** *The computation to solve overdetermined system by Gauss-Jordan elimination (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Symbolic Matrix with SymbolicC++ Automatic Version/main.cpp**).*

## xii. C++ Computation: Gauss-Jordan Elimination, Rank and Basis for Null Space

Find the rank and nullity of the matrix

$$A = \begin{bmatrix} 1 & -3 & 2 & 2 & 1 \\ 0 & 3 & 6 & 0 & -3 \\ 2 & -3 & -2 & 4 & 4 \\ 3 & -6 & 0 & 6 & 5 \\ -2 & 9 & 2 & -4 & -5 \end{bmatrix}$$

**Solution:**
With Gauss-Jordan elimination we will obtain the reduced row echelon form of $A$ as

$$R = \begin{bmatrix} 1 & 0 & 0 & 2 & \frac{4}{3} \\ 0 & 1 & 0 & 0 & -\frac{1}{6} \\ 0 & 0 & 1 & 0 & -\frac{5}{12} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since the matrix has three leading 1's, thus

$$\text{rank}(A) = 3$$

$$\text{dimension}(A) = 5$$

To find the nullity of $A$, we must find the dimension of the solution space of the linear system $A\boldsymbol{x} = \mathbf{0}$. The resulting matrix will be identical to matrix $R$ with additional last column of zeros, and hence the corresponding system of equations will be

$$\begin{aligned} x_1 + 2x_4 + \frac{4}{3}x_5 &= 0 \\ x_2 - \frac{1}{6}x_5 &= 0 \\ x_3 - \frac{5}{12}x_5 &= 0 \end{aligned}$$

Solving these equations for the leading variables yields

$$\begin{aligned} x_1 &= -2x_4 - \frac{4}{3}x_5 \\ x_2 &= \frac{1}{6}x_5 \\ x_3 &= \frac{5}{12}x_5 \end{aligned}$$

from which we obtain the general solution

$$\begin{aligned} x_1 &= -2x_4 - \frac{4}{3}x_5 \\ x_2 &= \frac{1}{6}x_5 \\ x_3 &= \frac{5}{12}x_5 \\ x_4 &= r \\ x_5 &= s \end{aligned}$$

or in column vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = r \begin{bmatrix} -2 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + s \begin{bmatrix} -\frac{4}{3} \\ \frac{1}{6} \\ \frac{5}{12} \\ 0 \\ 1 \end{bmatrix}$$

Because the two vectors form a basis for the solution space,

$$\text{nullity}(A) = \text{dimension}(A) - \text{rank}(A) = 2$$

In the C++ code, the different between previous section is that we load the matrix from textfile **matrixA.txt**, and then we are using two libraries at once

1. **SymbolicC++**
   To do the computation in Symbolic term, we declare the matrix that we create with this library as **B** in the C++ code, even if this case is only numeric we can be flexible for future project and change some entries into symbolic term. We load the textfile manually here and then saved as matrix **B**.

   We perform Gauss-Jordan elimination directly for matrix **B** and write the functions right inside the **int main()**.

2. **Armadillo**
   To do the computation in Float number term, we declare the matrix as **A**, with this declaration calling **arma::mat A(R,C,fill::zeros)**. To load the textfile is also easy as Armadillo presents a one line function to load a textfile that can be called as a matrix.

   After we load the textfile and save it as Armadillo matrix named **A**, we save it in float term for standard array in C++ with **float matrixc[R][C]**. The main point is to apply the functions of Gauss-Jordan that are written before the **int main()** into **matrixc**. Then compare it with the result from applying the same Gauss-Jordan elimination for matrix **B** from **SymbolicC++**.

The computation with **SymbolicC++** and **Armadillo** resulting in same number of rank but different basis for Null Space, the algorithm is the same, only that **SymbolicC++** will treat the entries and computation in Symbolic term while **Armadillo** will treat the entries and computation in float term. Personally, we prefer to use **SymbolicC++**' answer for this case.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include "symbolicc++.h"

using namespace std;
using namespace arma;

#define R 5 // number of rows
#define C 5 // number of columns
```

```
// Function to print the matrix
void PrintMatrix(float a[][C])
{
	int r = R;
	int c = C;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int GaussJordan1(float a[][C], int r, int n)
{
	// First operation of Gauss—Jordan elimination to make zeroes under
	     the leading 1's
	for (int f = 1; f < r; f++)
	{
		for (int i = f; i < r; i++)
		{
			if (a[i][f—1] == 0)
			{
				// Do nothing, just skip.
			}
			else
			{
				float pivot = a[f—1][f—1]/a[i][f—1];
				for (int k = 0; k < n; k++)
				{
					a[i][k] = (pivot*a[i][k]) — a[f—1][k];
				}
			}
		}
	}
	return 0;
}

int GaussJordan2(float a[][C], int r, int n)
{
	// Second operation of Gauss—Jordan elimination to make zeroes
	    above the leading 1's
	for (int j = 1; j < n; j++)
	{
		if (a[j][j] == 0)
		{
```

```
                }
                else
                {
                        for (int i = 0; i < j; i++)
                        {
                                float pivot = a[i][j]/a[j][j];
                                for (int k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] - (pivot*a[j][k]) ;
                                }
                        }
                }
        }
        return 0;
}


int GaussJordanS(float a[][C], int r, int n)
{
        // Swap row that has zeroes at j row, j column to the row below that
             has nonzero at the same j column
        for (int j = 0; j < n; j++)
        {
                for (int i = j; i < r; i++)
                {
                        if (a[i][j] !=0 && a[j][j] ==0)
                        {
                                for (int k = 0; k < n; k++)
                                {
                                        swap(a[i][k], a[j][k]);
                                }
                        }
                }
        }
        return 0;
}

int GaussJordan3(float a[][C], int r, int n)
{
        // Operation to make Leading 1's in all rows
        for (int i = 0; i < r; i++)
        {
                if (a[i][i] !=1)
                {
                        float pivot = a[i][i];
                        for (int k = 0; k < n; k++)
                        {
                                a[i][k] = a[i][k] / pivot ;
```

```
                        }
                }
        }
        return 0;
}

struct matrix
{
        int arr1[R][C];
};

struct matrix func(int r, int c)
{
        struct matrix matrix_mem;
        std::fstream in("matrixA.txt");
        float matrixtiles[R][C];
        for (int i = 0; i < r; ++i)
        {
                for (int j = 0; j < c; ++j)
                {
                        in >> matrixtiles[i][j];
                        matrix_mem.arr1[i][j] = matrixtiles[i][j] ;
                }
        }
        return matrix_mem;
}

// Driver code
int main(int argc, char** argv)
{
        arma::mat A(R,C,fill::zeros); //declare matrix A with size of R X C
            all have 0 entries with Armadillo.
        A.load("matrixA.txt"); // load matrixA.txt with Armadillo
        Matrix<Symbolic> B(R,C); // Declare matrix B with size R X C with
            SymbolicC++

        // Load matrixA.txt manually
        struct matrix a;
        a = func(R,C); // address of arr1
        for (int i = 0; i < R; ++i)
        {
                for (int j = 0; j < C; ++j)
                {
                        B[i][j] = a.arr1[i][j];
                }
        }

        cout << "G*****************Freya*****************" << endl;
```

```
cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;

// Gauss—Jordan computation for matrixc that is loaded with
    Armadillo still not perfect
float matrixc[R][C] = {};
for (int i = 0; i < R; ++i)
{
       for(int j = 0; j<C; ++j)
       {
              matrixc[i][j] = A[i+j*R] ;
       }
}
cout << endl;
cout << "A:\n" << B <<endl;
// *************************Computation**************************
// Swap row that has zeroes at j row,j column to the row below that
    has nonzero at the same j column
for (int j = 0; j < C; j++)
{
       for (int i = j; i < R; i++)
       {
              if (B[i][j] !=0 && B[j][j] ==0)
              {
                     for (int k = 0; k < C; k++)
                     {
                            swap(B[i][k], B[j][k]);
                     }
              }
       }
}
// Gauss—Jordan step 1: make zeroes below all leading 1's
for (int f = 1; f < R; f++)
{
       for (int i = f; i < R; i++)
       {
              if (B[i][f—1] == 0)
              {
                     // Do nothing, just skip.
              }
              else
              {
                     Symbolic pivot = B[f—1][f—1]/B[i][f—1];
                     for (int k = 0; k < C; k++)
                     {
                            B[i][k] = (pivot*B[i][k]) — B[f—1][k];
                     }
                     //cout << "A:\n" << B <<endl;
              }
```

```
                }
        }
        // Gauss—Jordan step 2: make zeroes above all leading 1's
        for (int j = 1; j < C; j++)
        {
                if (B[j][j] == 0)
                {

                }
                else
                {
                        for (int i = 0; i < j; i++)
                        {
                                Symbolic pivot = B[i][j]/B[j][j];
                                //cout << "i: " << i <<endl;
                                //cout << "j: " << j <<endl;
                                //cout << "pivot: " << pivot <<endl;
                                for (int k = 0; k < C; k++)
                                {
                                        B[i][k] = B[i][k] — (pivot*B[j][k]) ;
                                }
                                //cout << "A:\n" << B <<endl;
                        }
                }
        }

        // Operation to make Leading 1's in all rows
        for (int i = 0; i < R; i++)
        {
                if (B[i][i] !=1)
                {
                        Symbolic pivot = B[i][i];
                        for (int k = 0; k < C; k++)
                        {
                                B[i][k] = B[i][k] / pivot ;
                        }
                }
        }

        // ************************End of Computation
            **********************
        cout << "****************Reduced Row Echelon Form for matrix A
            ****************" << endl;
        cout << endl;

        cout << "A RREF (with SymbolicC++):\n" << B <<endl;
        cout << endl;
```

```
///*
GaussJordan1(matrixc, R, C);
GaussJordan2(matrixc, R, C);
GaussJordanS(matrixc, R, C);
GaussJordan3(matrixc, R, C);
cout << "A RREF (with Armadillo):" << endl;
PrintMatrix(matrixc);
cout << endl;
//*/

// To compute rank, dimension and nullity.
int rankA = 0;
for (int i = 0; i < R; ++i)
{
       for(int j = 0; j<C; ++j)
       {
              if (B[i][j] == 1)
              {
                     rankA = rankA + 1;
              }
              else
              {
                     rankA = rankA;
              }
       }
}
int nullity = C — rankA;

cout << "rank: " << rankA <<endl;
cout << endl;
cout << "dimension: " << C <<endl;
cout << endl;
cout << "nullity: " << nullity <<endl;
cout << endl;
cout << "****************Basis for the Null Space of Matrix A
    ****************" << endl;
cout << endl;

// To compute the Basis for the Null Space of matrix A, matrixd will
     have the size of C X nullity
// for now we can only input the nullity manually, just put as much
    numbers as you like the computation is amazing
Symbolic matrixd[C][96] = {};

for (int i = 0; i < rankA; i++)
{
       for (int j = 0; j < nullity; j++)
       {
```

```
                matrixd[i][j] = -1*B[i][j + rankA];
        }
    }
    for (int i = rankA; i < C; i++)
    {
        for (int j = 0; j < nullity; j++)
        {
            if (i-j == rankA)
            {
                matrixd[i][j] = 1;
            }
            else
            {
                matrixd[i][j] = 0;
            }
        }
    }

    for (int i = 0; i < C; i++)
    {
        for (int j = 0; j < nullity; j++)
        cout <<setw(6) << setprecision(4) << matrixd[i][j] << "\t";
        cout << endl;
    }

    arma::mat ANull(C,nullity,fill::zeros); //declare matrix ANull with
        size of C X nullity all have 0 entries with Armadillo.
    // Copy from matrixd to ANull
    for (int i = 0; i < C; ++i)
    {
        for(int j = 0; j<nullity; ++j)
        {
            ANull[i+j*C] = matrixd[i][j] ;
        }
    }
    //ANull.print("Null Basis :\n");
    cout << endl;
    cout << "Basis for Null Space in Column Form: " << endl;

    for (int j = 0; j < nullity; ++j)
    {
        cout << "column(" << j << ") : " << endl << ANull.col(j) <<
            endl;
    }

    // Optional: Armadillo' computation
    cout << endl;
    cout << "****************Armadillo Computation of Matrix A
```

```
                ****************" << endl;
            cout << endl;

            cout <<"Rank of matrix A :" << arma::rank(A) << endl;
            cout << endl;

            cout <<"Nullity of matrix A :" << A.n_cols - arma::rank(A) << endl;
            cout << endl;
            cout <<"Orthonormal basis of Null space of matrix A :\n" << arma::
                null(A) << endl;
            cout << endl;
        }
```

**C++ Code 135:** *main.cpp "Gauss-Jordan Elimination Rank and Basis for Null Space"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The functions that are defined before **int main()** are designed to perform Gauss-Jordan elimination / row operations for matrix $A$ that is loaded with **Armadillo** with **arma::mat A(R,C,fill::zeros)** then saved into 2-dimensional array **float matrixc[R][C] = {}**, hence the operations will be done in Float term.

```
            int GaussJordan1(float a[][C], int r, int n)
            {
                ...

                return 0;
            }
            int GaussJordan2(float a[][C], int r, int n)
            {
                ...

                return 0;
            }
            int GaussJordanS(float a[][C], int r, int n)
            {
                ...

                return 0;
            }
            int GaussJordan3(float a[][C], int r, int n)
            {
```

```
        ...

        return 0;
}
```

- The C++ computation code with for loop that are written inside the **int main()** is to perform Gauss-Jordan elimination / row operations for matrix $A$ that is loaded manually and saved as Symbolic matrix with **SymbolicC++** thus the operations will be done in Symbolic term. In this section the entries are all numeric, but with this code we can change the entries in matrix $B[i][j]$ into symbolic for experimentation.

```
// Swap row
for (int j = 0; j < C; j++)
{
        for (int i = j; i < R; i++)
        {
                if (B[i][j] !=0 && B[j][j] ==0)
                {
                        for (int k = 0; k < C; k++)
                        {
                                swap(B[i][k], B[j][k]);
                        }
                }
        }
}
...

...
for (int i = 0; i < R; i++)
{
        if (B[i][i] !=1)
        {
                Symbolic pivot = B[i][i];
                for (int k = 0; k < C; k++)
                {
                        B[i][k] = B[i][k] / pivot ;
                }
        }
}
```

```
G****************Freya****************
Matrix A (Coefficient Matrix):
   1.0000  -3.0000   2.0000   2.0000   1.0000
        0   3.0000   6.0000        0  -3.0000
   2.0000  -3.0000  -2.0000   4.0000   4.0000
   3.0000  -6.0000        0   6.0000   5.0000
  -2.0000   9.0000   2.0000  -4.0000  -5.0000


A:
[ 1 -3  2  2  1]
[ 0  3  6  0 -3]
[ 2 -3 -2  4  4]
[ 3 -6  0  6  5]
[-2  9  2 -4 -5]

****************Reduced Row Echelon Form for matrix A*************

A RREF (with SymbolicC++):
[  1     0     0     2    4/3 ]
[  0     1     0     0    -1/6]
[  0     0     1     0   -5/12]
[  0     0     0     0     0  ]
[  0     0     0     0     0  ]


A RREF (with Armadillo):
     1       0       0       2       1
     0       1       0       0    -0.2
    -0      -0       1      -0    -0.4
  -nan    -nan    -nan    -nan    -nan
  -nan    -nan    -nan    -nan    -nan

rank: 3

dimension: 5

nullity: 2
```

```
****************Basis for the Null Space of Matrix A****************

    -2      -4/3
     0       1/6
     0       5/12
     1       0
     0       1

Basis for Null Space in Column Form:
column(0) :
  -2.0000
        0
        0
   1.0000
        0

column(1) :
  -1.3333
   0.1667
   0.4167
        0
   1.0000


****************Armadillo Computation of Matrix A****************

Rank of matrix A :3

Nullity of matrix A :2

Orthonormal basis of Null space of matrix A :
   0.9003  -0.1874
  -0.0039   0.1335
  -0.0098   0.3338
  -0.4345  -0.4403
  -0.0235   0.8011
```

**Figure 23.62:** *The computation to solve symmetric matrix by Gauss-Jordan elimination, find the rank, and the basis for the null space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination, Rank and Nullity of a Matrix with Armadillo and SymbolicC++/main.cpp**).*

**Computational Guide 23.2: Gauss-Jordan Elimination Part 1**

The algorithm to perform Gauss-Jordan elimination on matrix $A$ with size of $m \times n$ into the reduced row echelon form that has zeroes below and above the leading 1's is very important to the linear algebra world. The entry / index of the matrix $A$ can be represented as $a_{ij}$ with $i = 1, 2, \ldots m$ and $j = 1, 2, \ldots n$, the algorithm is as follow

1. Swap row by checking the matrix' index at ($j$-th row, $j$-th column) with $j = 1, 2, \ldots, n$ toward matrix index at ($i$-th row, $j$-th column) with $i = j, j+1, \ldots, m$. This is to make sure all zeroes will be put below and the diagonal entries will be made to have nonzero entries.

2. Gauss-Jordan step 1: Make zeroes below all leading 1's.
   Make a pivot, the first pivot is connected to the the first diagonal, the diagonal of the matrix is the place of the leading 1's this is why we will use $a_{ii}$ with $i = 1, 2, \ldots, m$. Thus we will do row operations to make zeroes below all $a_{ii}$ with this formula

$$\text{pivot}_k = \frac{a_{ii}}{a_{ki}} \tag{23.97}$$

   with $k = i+1, i+2, \ldots, m$. After we obtain the pivot from the first diagonal entry then we use this formula to obtain new entries for the second row

$$\begin{aligned} a_{kj} &= \text{pivot}_k \times a_{kj} - a_{ij} \\ a_{kj} &= \left( \frac{a_{ii}}{a_{ki}} \times a_{kj} \right) - a_{ij} \end{aligned} \tag{23.98}$$

   with $j = 1, 2, \ldots, n$. After we finish the looping for $j$, then for $k$, then the looping for $i$ the outermost loop will make all zeroes below all leading 1's places / the diagonal entries of the matrix.

**Computational Guide 23.3: Gauss-Jordan Elimination Part 2**

3. Gauss-Jordan step 2: Make zeroes above all leading 1's.
   Almost the same as make zeroes below all leading 1's we will make a pivot point first, except the formula to do the row operations will be different.

   This time, we will use $a_{jj}$ with $j = 2, \ldots, n$ as a vital entry for the pivot to make zeroes above the leading 1's with this formula

   $$\text{pivot}_k = \frac{a_{kj}}{a_{jj}} \tag{23.99}$$

   with $k = 1, 2, \ldots, j-1$.

   After we obtain the pivot from the second diagonal entry then we use this formula to obtain new entries for the first row

   $$\begin{aligned} a_{ks} &= a_{ks} - \text{pivot}_k \times a_{js} \\ a_{ks} &= a_{ks} - \left( \frac{a_{kj}}{a_{jj}} \times a_{js} \right) \end{aligned} \tag{23.100}$$

   with $s = 1, 2, \ldots, n$. The first loop is to run $s$ and then the second loop is $k$, then the last / outermost loop is $j$. The first diagonal here is $a_{22}$ thus we will perform row operation on the first row to make $a_{12} = 0$. Then we move on to the next diagonal which is $a_{33}$, here we will have $k = 1, 2$ and we will perform 2 row operations to make $a_{13} = 0$ and $a_{23} = 0$, and so on till the last diagonal entry in the matrix.

4. After we make zeroes below and above the leading 1's candidate, since the diagonal entries probably still not equal to 1, we perform the last computation, to mae all diagonal entries have value of 1.

   The pivot is very simple, the diagonal itself.

   $$\text{pivot}_i = a_{ii} \tag{23.101}$$

   with $i = 1, 2, \ldots, m$.

   Then we perform the computation to all the rows with this formula

   $$a_{ij} = \frac{a_{ij}}{a_{ii}} \tag{23.102}$$

   with $j = 1, 2, \ldots, n$. The first loop is $j$ then the last loop is $i$, we divide all entries in the first row with the first diagonal entry $a_{11}$, then we move to the next row till all diagonal become 1 or become the real leading 1's.

To comprehend the algorithm first is necessary, so we can write the C++ code easily.

## xiii. C++ Plot and Computation: The Geometry of the Solution Set of $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{x} = \boldsymbol{0}$

In [11] book chapter 4.7 this section is only one line explained and a picture of the line, then we think we want to dig deeper so we are able to comprehend why the lines can represent solution set for homogeneous and nonhomogeneous linear systems.

Suppose we have matrix $A$ of size $2 \times 2$

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 6 \end{bmatrix}$$

the reduced echelon form of $A$ will be

$$R = \begin{bmatrix} 1 & 3 \\ 0 & 0 \end{bmatrix}$$

With this we know that

$$\begin{aligned} \text{rank}(A) &= 1 \\ \text{dimension}(A) &= 2 \\ \text{null}(A) &= 1 \end{aligned}$$

If $\boldsymbol{x}_0$ is any solution of a consistent linear system $A\boldsymbol{x} = \boldsymbol{b}$, and if $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ is a basis for the null space of $A$, then every solution of $A\boldsymbol{x} = \boldsymbol{b}$ can be expressed in the form

$$\boldsymbol{x} = \boldsymbol{x}_0 + c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k$$

for all choices of scalars $c_1, c_2, \ldots, c_k$. The linear system $A\boldsymbol{x} = \boldsymbol{b}$ is consistent if and only if $\boldsymbol{b}$ is expressible as a linear combination of the column vectors of $A$, in another words $A\boldsymbol{x} = \boldsymbol{b}$ is consistent if and only if $\boldsymbol{b}$ is in the column space of $A$.

Geometrically the solution set of $A\boldsymbol{x} = \boldsymbol{b}$ can be viewed as the translation by $\boldsymbol{x}_0$ of the solution space of $A\boldsymbol{x} = \boldsymbol{0}$.

We will examine it, first we construct the linear system of $A\boldsymbol{x} = \boldsymbol{0}$

$$\begin{bmatrix} 1 & 3 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Since the first and second row of $A$ is linearly dependent, we will obtain the basis for the null space of $A$ as

$$\begin{aligned} x_1 + 3x_2 &= 0 \\ x_1 &= -3x_2 \end{aligned}$$

Now we set the parameter

$$\begin{aligned} x_1 &= -3r \\ x_2 &= r \end{aligned}$$

Thus the basis is

$$\boldsymbol{x} = r \begin{bmatrix} -3 \\ 1 \end{bmatrix}$$

The vector

$$\begin{bmatrix} -3 \\ 1 \end{bmatrix}$$

is the general solution of $A\boldsymbol{x} = \mathbf{0}$, we can draw a vector that is passing through $(-3, 1)$ and $(0, 0)$ to see this vector in $\mathbb{R}^2$.

Remember that every solution of $A\boldsymbol{x} = \boldsymbol{b}$ can be expressed in the form

$$\boldsymbol{x} = \boldsymbol{x}_0 + c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k$$

In this case, we have $c_1 = r$ as the parameter and $\boldsymbol{v}_1 = (-3, 1)$.

Now, we will examine the linear system of $A\boldsymbol{x} = \boldsymbol{b}$, it has a particular solution which is vector $\boldsymbol{x}_0$, in this case it will be like this

$$A\boldsymbol{x}_0 = \boldsymbol{b}$$
$$\begin{bmatrix} 1 & 3 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

with $\boldsymbol{b} = (2, 4)$.

To be consistent the linear system will only have one solution that satisfies the equation above, we will name the vector as $\boldsymbol{x}_0$ thus we can find the solution as

$$\boldsymbol{x}_0 = \begin{bmatrix} 0.2 \\ 0.6 \end{bmatrix}$$

We choose to go with matrix $A$ with size of $2 \times 2$ that will have rank of 1 and basis for the null space of $A$ is 1 as well, so we can plot the vector and line easily to comprehend the geometry counterpart of this section.

When we have a linear system of

$$A\boldsymbol{x} = \mathbf{0}$$

when we solve for $\boldsymbol{x}$ we will obtain the general solution of $A\boldsymbol{x} = \mathbf{0}$ consisting of vectors that will become a basis for the null space of $A$. This vector that we call $\boldsymbol{x}$ will go through the origin $\mathbf{0}$. So you can get the picture how to draw this vector. Furthermore with $A$ is a square matrix it will act like a scalar, thus without changing the size of $A$, if we change the entries for $A$ we will get the scalar multiple of $\boldsymbol{x}$ that will still go through the origin $\mathbf{0}$. This explains why the solution space of $A\boldsymbol{x} = \mathbf{0}$ is a line through the origin. But what the line will looks like will need another elementary line equation knowledge. To know the line equation that passes through two points $\boldsymbol{x}$ and $\mathbf{0}$ we will have

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

On this case we have the null basis as $\boldsymbol{x} = (-3, 1)$ and $\mathbf{0} = (0, 0)$ thus

$$\begin{aligned} m &= \frac{1 - 0}{-3 - 0} \\ &= -\frac{1}{3} \end{aligned}$$

$m$ is the symbol for gradient, then for the line equation that passes through that two points

$$\begin{aligned} y &= mx + c \\ &= -\frac{1}{3}x + 0 \\ y &= -\frac{1}{3}x \end{aligned}$$

we choose the value $c = 0$ by default, because the line will passes through $(0, 0)$.

Now, take a look again at the linear system of

$$A\boldsymbol{x} = \boldsymbol{b}$$

The solution set of $A\boldsymbol{x} = \boldsymbol{b}$ is also a line, but how by logic we know that this line has some connection to the line that represent the solution space of $A\boldsymbol{x} = \mathbf{0}$?

First, we know that $\boldsymbol{x}_0$ will be on the solution set of $A\boldsymbol{x} = \boldsymbol{b}$, and that if we add two vectors $\boldsymbol{x} + \boldsymbol{x}_0$, the result will also be on the solution set of $A\boldsymbol{x} = \boldsymbol{b}$.

Hence, remember again that

$$\begin{aligned} \boldsymbol{x}_0 &= (x_1, y_1) = (0.2, 0.6) \\ \boldsymbol{x} &= (-3, 1) \\ \boldsymbol{x}_0 + \boldsymbol{x}_0 &= (x_2, y_2) = (-2.8, 1.6) \end{aligned}$$

thus for the line equation

$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{y_2 - 0.6}{x_2 - x_1} \\ \frac{y - 0.6}{x - 0.2} &= \frac{1.6 - 0.6}{-2.8 - 0.2} \\ (y - 0.6)(-3) &= (x - 0.2)(1) \\ -3y + 1.8 &= x - 0.2 \\ y &= -\frac{1}{3}x + \frac{2}{3} \end{aligned}$$

this line will represents the solution set of $A\boldsymbol{x} = \boldsymbol{b}$.

The C++ code is using **Armadillo** to load the matrix $A$ and vector $\boldsymbol{b}$, it is also useful to compute rank automatically. The plot is done by gnuplot that saves the day as always. The main idea of this section is to comprehend why the solution set of $A\boldsymbol{x} = \boldsymbol{b}$ can be viewed geometrically as the translation by $\boldsymbol{x}_0$ of the solution space of $A\boldsymbol{x} = \mathbf{0}$.

If we remember the parallelogram rule for vector addition, when we have two vectors in 2-space or 3-space that are positioned so their initial points coincide (for this case coincide in $\mathbf{0}$), then the two vectors form adjacent sides of a parallelogram, and the sum $\boldsymbol{x} + \boldsymbol{x}_0$ is the vector represented by the arrow from the same initial point $\mathbf{0}$ to the opposite vertex of the parallelogram. Thus if $\boldsymbol{x} \in A\boldsymbol{x} = \mathbf{0}$ and $\boldsymbol{x}_0 \in A\boldsymbol{x} = \boldsymbol{b}$, then $\boldsymbol{x} + \boldsymbol{x}_0 \in A\boldsymbol{x} = \boldsymbol{b}$ This explains the translation of the solution space in another way by using parallelogram rule for vector addition.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

using namespace std;
using namespace arma;

#define C_Mat 3 // Define the number of column for the linear system Ax=b
#define R_Mat 2 // Define the number of row for the linear system Ax=b
#define C_MatA 2 // Define the number of column for the matrix A
#define R_MatA 2 // Define the number of row for the matrix A

// Function to print the matrix
void PrintMatrix(float a[][C_Mat])
{
	int r = R_Mat;
	int c = C_Mat;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(4) << a[i][j] << "\t";
		cout << endl;
	}
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float a[][C_Mat], int r, int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < r; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < r && a[i + c][i] == 0)
			c++;
			if ((i + c) == r)
```

```
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(a[j][k], a[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row echelon
                                    form(diagonal matrix)
                                float pro = a[j][i] / a[i][i];
                                for (k = 0; k <= n; k++)
                                a[j][k] = a[j][k] - (a[i][k]) * pro;
                        }
                }
        }
        return flag;
}
int PerformOperation2(float a[][C_Mat], int r, int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pivot = 0;

        // Performing elementary operations again to make leading 1's
        for (i = 2; i < r; i++)
        {
                for (j = 0; j < n; j++)
                {
                        if (a[i][j] != 1 && a[i][j] !=0)
                        {
                                float pivot = a[i][j];
                                for (k = 0; k <= n; k++)
                                {
                                        a[i][k] = a[i][k] / pivot;
                                }
                                j = n-1;
                        }
                }
        }
        return flag;
}
int PerformOperation3(float a[][C_Mat], int r, int n)
{
```

```
int i, j, k = 0, c, flag = 0, m = 0;
float pivot = 0;

// Performing elementary operations again to make reduced row
    echelon form
for (i = 2; i < r; i++)
{
       for (j = 0; j < n; j++)
       {
              if (a[i][j] != 0 && a[i-1][j] !=0)
              {
                     if ( a[i-1][j] == -1 )
                     {
                            for (k = 0; k <= n; k++)
                            {
                                   a[i-1][k] = -1*(a[i-1][k]);
                            }
                     }
                     float pivot = a[i-1][j]/a[i][j];
                     for (k = 0; k <= n; k++)
                     {
                            a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                ;
                     }
              }
       }
}
for (i = 2; i < r; i++)
{
       for (j = 0; j < n; j++)
       {
              if (a[i][j] != 0 && a[i-2][j] !=0 && a[i-1][j] == 0)
              {
                     for (k = 0; k <= n; k++)
                     {
                            swap(a[i][k], a[i-1][k]); // swap row
                                that has zero entry in between two
                                nonzero entries downward
                     }
              }
       }
}
for (i = 2; i < r; i++)
{
       for (j = 0; j < n; j++)
       {
              if (a[i][j] != 0 && a[i-1][j] !=0)
              {
```

```
                                float pivot = a[i-1][j]/a[i][j];
                                for (k = 0; k < n; k++)
                                {
                                        a[i][k] = a[i][k] - (a[i-1][k] * pivot)
                                            ;
                                }
                                //break;
                                j = n-1;
                                i = r-1;
                        }
                }

        }

        return flag;
}
// Function to print the desired result if unique solutions exists,
    otherwise
// prints no solution or infinite solutions depending upon the input given.
void PrintResult(float a[][C_Mat], int r, int flag)
{
        cout << "The solution/s : ";

        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < r; i++)
                cout << "x(" << i << ") = "<< a[i][r] / a[i][i] << ", ";
        }
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float a[][C_Mat], int r, int n, int flag)
{
        int i, j;
        float sum;

        // flag == 2 for infinite solution
```

```
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < r; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}

// Driver code
int main(int argc, char** argv)
{
        arma::mat A(R_MatA,C_MatA,fill::zeros); //declare matrix A with size
             of R X C all have 0 entries with Armadillo.
        A.load("matrixA.txt");
        mat B;
        B.load("vectorB.txt");
        mat X;
        X = solve(A,B,solve_opts::force_approx);

        cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
        cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
        cout <<"Solution:" << "\n" << X <<endl;

        float matrixc[R_Mat][C_Mat] = {};
        float matrixd[R_Mat][C_Mat] = {};

        for (int i = 0; i < R_Mat; ++i)
        {
                for(int j = 0; j<C_Mat-1; ++j)
                {
                        matrixc[i][j] = A[i+j*R_Mat] ;
                        matrixd[i][j] = A[i+j*R_Mat] ;
                }
        }
        for (int i = 0; i < R_Mat; ++i)
        {
                for(int j = C_Mat-1; j<C_Mat; ++j)
                {
                        matrixc[i][j] = B[i] ;
                        matrixd[i][j] = 0 ;
                }
        }
```

```
cout <<"Matrix for the linear system Ax=b :" <<endl;
PrintMatrix(matrixc);
cout <<"Matrix for the linear system Ax=0 :" <<endl;
PrintMatrix(matrixd);
//cout <<"Rank of matrix A :" << arma::rank(A) << endl;

// Order of Matrix(n)
int n = 7, flag = 0;

// Performing Matrix transformation
flag = PerformOperation(matrixc, R_Mat, C_Mat);

if (flag == 1)
flag = CheckConsistency(matrixc, R_Mat, C_Mat, flag);

cout << endl;
cout << "G*****************************Freya*****************" <<
     endl;
cout << endl;

PerformOperation2(matrixc, R_Mat, C_Mat);
PerformOperation3(matrixc, R_Mat, C_Mat);
PerformOperation2(matrixc, R_Mat, C_Mat);

cout << "Row Echelon Form for linear system Ax=b : " << endl;
PrintMatrix(matrixc);

cout << endl;
cout << "G*****************************Freya*****************" <<
     endl;
cout << endl;

PerformOperation(matrixd, R_Mat, C_Mat);
PerformOperation2(matrixd, R_Mat, C_Mat);
PerformOperation3(matrixd, R_Mat, C_Mat);
PerformOperation2(matrixd, R_Mat, C_Mat);
cout <<"Row Echelon Form for linear system Ax=0 :" <<endl;
PrintMatrix(matrixd);

// Plotting time
Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_C_x;
```

```
std::vector<double> pts_C_y;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;

float o = 0;
float xx;
float xy;
// If conditional for determining vector x
if (arma::rank(A) == 0)
{
       xx = 1;
       xy = 0;
}
if (arma::rank(A) == 1)
{
       xx = -matrixc[0][1];
       xy = 1;
}
if (arma::rank(A) == 2)
{
       xx = 0;
       xy = 0;
       cout << "Matrix A has rank=2, thus has no basis for the null
           space" << endl;
}
// For vector x0
float x0x = X[0];
float x0y = X[1];

// Create a vector x with origin at (0,0) and terminal point at
    (-3,1)
pts_B_x .push_back(o);
pts_B_y .push_back(o);
pts_B_dx.push_back(xx);
pts_B_dy.push_back(xy);
// Create a vector x0 with origin at (0,0) and terminal point at
    (0.2,0.6)
pts_C_x .push_back(o);
pts_C_y .push_back(o);
pts_C_dx.push_back(x0x);
pts_C_dy.push_back(x0y);
// Create a vector x+x0
pts_D_x .push_back(o);
pts_D_y .push_back(o);
```

```
            pts_D_dx.push_back(xx + x0x);
            pts_D_dy.push_back(xy + x0y);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-3.3:1.3]\nset yrange [-1:2]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "f(x) = -x/3\n";
            gp << "g(x) = -x/3 + 0.66666 \n";
            gp << "plot '-' with vectors title 'x', '-' with vectors title 'x_
                {0}','-' with vectors title 'x + x_{0}' lc 'green',f(x) title '
                Solution Space of Ax=0' lt 3,g(x) with linespoints pointtype 1
                pointsize 0.1 pi -2 lc 'black' title 'Solution Set of Ax=b'\n";
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        }
```

**C++ Code 136:** *main.cpp "2D Plot of Solution Space Ax*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- With **Armadillo** we load two textfiles **matrixA.txt** and **vectorB.txt** and then declare two 2-dimensional array with name **matrixc** (that will represent $A\boldsymbol{x} = \boldsymbol{b}$) and the other is **matrixd** (that will represent $A\boldsymbol{x} = \mathbf{0}$).

  Afterwards, we will perform row operations to **matrixc** and **matrixd** to find the solution for $A\boldsymbol{x} = \boldsymbol{b}$ and to find the null basis for $A\boldsymbol{x} = \mathbf{0}$.

```
            arma::mat A(R_MatA,C_MatA,fill::zeros); //declare matrix A with
                 size of R X C all have 0 entries with Armadillo.
            A.load("matrixA.txt");
            mat B;
            B.load("vectorB.txt");
            mat X;
            X = solve(A,B,solve_opts::force_approx);

            cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
            cout <<"Vector B (Nonhomogeneous System):" << "\n" << B <<endl;
            cout <<"Solution:" << "\n" << X <<endl;
```

```
float matrixc[R_Mat][C_Mat] = {};
float matrixd[R_Mat][C_Mat] = {};

for (int i = 0; i < R_Mat; ++i)
{
        for(int j = 0; j<C_Mat-1; ++j)
        {
                matrixc[i][j] = A[i+j*R_Mat] ;
                matrixd[i][j] = A[i+j*R_Mat] ;
        }
}
for (int i = 0; i < R_Mat; ++i)
{
        for(int j = C_Mat-1; j<C_Mat; ++j)
        {
                matrixc[i][j] = B[i] ;
                matrixd[i][j] = 0 ;
        }
}
```

- We create an if conditional for determining vector $\boldsymbol{x}$, since it will depends on the rank of matrix $A$. Thus, we declare the variables **xx** and **xy** first without giving them any value, the value itself will depends on rank$(A)$.

  For reminder, when rank$(A) = 0$, there should be another basis which is $(0, 1)$, for the case where the rank is 0 for $\mathbb{R}^2$ we will have two basis vectors that are linearly independent and span $\mathbb{R}^2$, the easiest choice is the standard basis $(1, 0)$ and $(0, 1)$.

```
float o = 0;
float xx;
float xy;
// If conditional for determining vector x
if (arma::rank(A) == 0)
{
        xx = 1;
        xy = 0;
}
if (arma::rank(A) == 1)
{
        xx = -matrixc[0][1];
        xy = 1;
}
if (arma::rank(A) == 2)
{
        xx = 0;
        xy = 0;
        cout << "Matrix A has rank=2, thus has no basis for the
            null space" << endl;
}
```

```
            // For vector x0
            float x0x = X[0];
            float x0y = X[1];
```

**Figure 23.63:** *The plot for vectors $\boldsymbol{x}, \boldsymbol{x}_0, \boldsymbol{x}+\boldsymbol{x}_0$, the solution space of $A\boldsymbol{x}=\boldsymbol{0}$ and the solution set of $A\boldsymbol{x}=\boldsymbol{b}$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot for General Solution and Particular Solution for Linear System Ax=b and Ax=0 with Armadillo/main.cpp**).*

```
Matrix A (Coefficient Matrix):
   1.0000   3.0000
   2.0000   6.0000

Vector B (Nonhomogeneous System):
   2.0000
   4.0000

Solution:
   0.2000
   0.6000

Matrix for the linear system Ax=b :
     1       3       2
     2       6       4
Matrix for the linear system Ax=0 :
     1       3       0
     2       6       0

G****************************Freya*****************

Row Echelon Form for linear system Ax=b :
     1       3       2
     0       0       0

G****************************Freya*****************

Row Echelon Form for linear system Ax=0 :
     1       3       0
     0       0       0
```

**Figure 23.64:** *The computation to solve matrix $A$ into reduced row echelon form for linear system $A\boldsymbol{x} = \mathbf{0}$ and $A\boldsymbol{x} = \boldsymbol{b}$, and find the solution of $A\boldsymbol{x} = \boldsymbol{b}$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot for General Solution and Particular Solution for Linear System Ax=b and Ax=0 with Armadillo/main.cpp**).*

## xiv. C++ Plot and Computation: Reflection about the $x$-axis and $y$-axis in $\mathbb{R}^2$

We are going to observe standar matrix for reflection operator about $x$-axis, $y$-axis, and the line $y = x$ in $\mathbb{R}^2$.

**Reflection about the $y$-axis**

If we reflect standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (-1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (-x,y)$$

with the standard matrix for the reflection about $y$-axis

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

**Reflection about the $x$-axis**

If we reflect standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,-1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x,-y)$$

with the standard matrix for the reflection about $x$-axis

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

**Reflection about the line** $y = x$
If we reflect standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (0,1)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (1,0)$$

For arbitrary point $(x, y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x, y) = (y, x)$$

with the standard matrix for the reflection about the line $y = x$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The C++ code is using **Armadillo** to load vector $\boldsymbol{x} = (x, y)$ and perform the computation such as matrix times vector multiplication, and then we use gnuplot to do the plotting.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 2 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
        int r = N;
        int c = N;
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}
```

```
int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for reflection about y-axis
        float matrixAy[N][N] = {};
        // Create standard matrix for reflection about x-axis
        float matrixAx[N][N] = {};
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        if (i == j)
                        {
                                matrixAy[i][j] = 1;
                                matrixAx[i][j] = 1;
                        }
                        else
                        {
                                matrixAy[i][j] = 0;
                                matrixAx[i][j] = 0;
                        }
                }
        }
        matrixAy[0][0] = -1;
        matrixAx[1][1] = -1;

        // Create standard matrix for reflection about line y=x
        float matrixAlinex[N][N] = {};
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        if (i == j)
                        {
                                matrixAlinex[i][j] = 0;
                        }
                        else
                        {
                                matrixAlinex[i][j] = 1;
                        }
                }
        }
        arma::mat ArmaAy(N,N,fill::zeros); // Declare matrix ArmaAy of size
```

```
    N X N with Armadillo
arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
arma::mat ArmaAlinex(N,N,fill::zeros); // Declare matrix ArmaAlinex
    of size N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAy[i+j*N] = matrixAy[i][j] ;
              ArmaAx[i+j*N] = matrixAx[i][j] ;
              ArmaAlinex[i+j*N] = matrixAlinex[i][j];
       }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for reflection about the y-axis:" << "\n"
    << ArmaAy <<endl;
cout <<"Standard Matrix for reflection about the x-axis:" << "\n"
    << ArmaAx <<endl;
cout <<"Standard Matrix for reflection about the line y=x:" << "\n"
    << ArmaAlinex <<endl;

arma::mat Xreflectedy = ArmaAy*X ;
arma::mat Xreflectedx = ArmaAx*X ;
arma::mat Xreflectedlinex = ArmaAlinex*X ;
cout <<"Vector x reflected about the y-axis:" << "\n" <<
    Xreflectedy <<endl;
cout <<"Vector x reflected about the x-axis:" << "\n" <<
    Xreflectedx <<endl;
cout <<"Vector x reflected about the line y=x:" << "\n" <<
    Xreflectedlinex <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
```

```
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;

float o = 0;
float e1x = 0;
float e1y = 1;
float e2x = 1;
float e2y = 0;

float vect_e1[] = { e1x,e1y };
float vect_e2[] = { e2x,e2y };

cout << "Vector e1:" << setw(15) << "vector e2:" << endl;

for (int i = 0; i < n; i++)
{
       cout << vect_e1[i] << "\t" << "\t" << vect_e2[i] ;
       cout << endl;
}
cout << endl;

// Create a vector with origin at (0,0) and terminal point at (1,0)
pts_B_x .push_back(o);
pts_B_y .push_back(o);
pts_B_dx.push_back(e1x);
pts_B_dy.push_back(e1y);
// Create a vector with origin at (0,0) and terminal point at (0,1)
pts_C_x .push_back(o);
pts_C_y .push_back(o);
pts_C_dx.push_back(e2x);
pts_C_dy.push_back(e2y);
// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
// Reflection of vector x about the y—axis
pts_E_x .push_back(o);
pts_E_y .push_back(o);
```

```
            pts_E_dx.push_back(Xreflectedy[0]);
            pts_E_dy.push_back(Xreflectedy[1]);
            // Reflection of vector x about the x-axis
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_dx.push_back(Xreflectedx[0]);
            pts_F_dy.push_back(Xreflectedx[1]);
            // Reflection of vector x about the x-axis
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_dx.push_back(Xreflectedlinex[0]);
            pts_G_dy.push_back(Xreflectedlinex[1]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "f(x) = x\n";
            gp << "plot '-' with vectors title 'e_{1}', '-' with vectors title
                'e_{2}', '-' with vectors title 'x','-' with vectors title 'x
                reflected about y-axis' lc 'green', '-' with vectors title 'x
                reflected about x-axis', '-' with vectors title 'x reflected
                about line y=x' lc 'orange',f(x) title 'y=x' lc 'black;\n";
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_dx, pts_G_dy));
        }
```

**C++ Code 137:** *main.cpp "Reflection in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We declare 3 new matrices class **Xreflectedy, Xreflectedx, Xreflectedlinex** to store the tranformation / the image of the reflection of vector $x$ for each different reference line of reflection.

```
            arma::mat Xreflectedy = ArmaAy*X ;
            arma::mat Xreflectedx = ArmaAx*X ;
```

```
arma::mat Xreflectedlinex = ArmaAlinex*X ;
```

**Figure 23.65:** *The plot to reflect vector about the y-axis, x-axis, and line $y = x$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Reflection About Axis and Line/main.cpp**).*

## xv. C++ Plot and Computation: Reflection about the $xy$-plane, $xz$-plane and $yz$-plane in $\mathbb{R}^3$

We are going to observe standar matrix for reflection operator about the $xy$-plane, $xz$-plane and $yz$-plane in $\mathbb{R}^3$.

**Reflection about the $xy$-plane**
If we reflect standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,-1) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (x, y, -z)$$

with the standard matrix for the reflection about $xy$-plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

**Reflection about the $xz$-plane**
If we reflect standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,-1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,1) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (x, -y, z)$$

with the standard matrix for the reflection about $xz$-plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Reflection about the $yz$-plane**
If we reflect standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (-1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,1) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (-x, y, z)$$

with the standard matrix for the reflection about $yz$-plane

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For the C++ code, the procedure is similar to the previous section but we enlarge the matrix into the size of $3 \times 3$, another different is we are plotting it with gnuplot' **splot**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
        int r = N;
        int c = N;
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}

int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for reflection about xy-plane
        float matrixAxy[N][N] = {};
        // Create standard matrix for reflection about xz-plane
        float matrixAxz[N][N] = {};
        // Create standard matrix for reflection about yz-plane
```

```
float matrixAyz[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              if (i == j)
              {
                     matrixAxy[i][j] = 1;
                     matrixAxz[i][j] = 1;
                     matrixAyz[i][j] = 1;
              }
              else
              {
                     matrixAxy[i][j] = 0;
                     matrixAxz[i][j] = 0;
                     matrixAyz[i][j] = 0;
              }
       }
}
matrixAxy[2][2] = -1;
matrixAxz[1][1] = -1;
matrixAyz[0][0] = -1;

arma::mat ArmaAxy(N,N,fill::zeros); // Declare matrix ArmaAxy of
    size N X N with Armadillo
arma::mat ArmaAxz(N,N,fill::zeros); // Declare matrix ArmaAxz of
    size N X N with Armadillo
arma::mat ArmaAyz(N,N,fill::zeros); // Declare matrix ArmaAyz of
    size N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAxy[i+j*N] = matrixAxy[i][j] ;
              ArmaAxz[i+j*N] = matrixAxz[i][j] ;
              ArmaAyz[i+j*N] = matrixAyz[i][j] ;
       }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for reflection about the xy-plane:" << "\n"
     << ArmaAxy <<endl;
cout <<"Standard Matrix for reflection about the xz-plane:" << "\n"
     << ArmaAxz <<endl;
cout <<"Standard Matrix for reflection about the yz-plane:" << "\n"
     << ArmaAyz <<endl;
```

```
arma::mat Xreflectedxy = ArmaAxy*X ;
arma::mat Xreflectedxz = ArmaAxz*X ;
arma::mat Xreflectedyz = ArmaAyz*X ;
cout <<"Vector x reflected about the xy-plane:" << "\n" <<
    Xreflectedxy <<endl;
cout <<"Vector x reflected about the xz-plane:" << "\n" <<
    Xreflectedxz <<endl;
cout <<"Vector x reflected about the yz-plane:" << "\n" <<
    Xreflectedyz <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
pts_D_dz.push_back(X[2]);
// Reflection of vector x about the xy-plane
pts_E_x .push_back(o);
pts_E_y .push_back(o);
```

```
            pts_E_z .push_back(o);
            pts_E_dx.push_back(Xreflectedxy[0]);
            pts_E_dy.push_back(Xreflectedxy[1]);
            pts_E_dz.push_back(Xreflectedxy[2]);
            // Reflection of vector x about the xz-plane
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(Xreflectedxz[0]);
            pts_F_dy.push_back(Xreflectedxz[1]);
            pts_F_dz.push_back(Xreflectedxz[2]);
            // Reflection of vector x about the yz-plane
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(Xreflectedyz[0]);
            pts_G_dy.push_back(Xreflectedyz[1]);
            pts_G_dz.push_back(Xreflectedyz[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-1:1]\nset yrange [-1:1]\nset zrange [-1:1]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";
            gp << "set view 60,5,1\n"; // pitch,yaw,zoom
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "splot '-' with vectors title 'x','-' with vectors title 'x
                reflected about xy-plane' lc 'green', '-' with vectors title '
                x reflected about xz-plane', '-' with vectors title 'x
                reflected about yz-plane' lc 'orange'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
        }
```

**C++ Code 138:** *main.cpp "Reflection in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.66:** *The plot to reflect vector* $\boldsymbol{x}$ *about the xy-plane, xz-plane, and line yz-plane (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Reflection About Plane/main.cpp**).*

```
Vector x:
   0.5000
   0.8000
   0.6000

Standard Matrix for reflection about the xy-plane:
   1.0000        0        0
        0   1.0000        0
        0        0  -1.0000

Standard Matrix for reflection about the xz-plane:
   1.0000        0        0
        0  -1.0000        0
        0        0   1.0000

Standard Matrix for reflection about the yz-plane:
  -1.0000        0        0
        0   1.0000        0
        0        0   1.0000

Vector x reflected about the xy-plane:
   0.5000
   0.8000
  -0.6000

Vector x reflected about the xz-plane:
   0.5000
  -0.8000
   0.6000

Vector x reflected about the yz-plane:
  -0.5000
   0.8000
   0.6000
```

**Figure 23.67:** *The computation to reflect vector* $\boldsymbol{x}$ *about the xy-plane, xz-plane, and line yz-plane (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Reflection About Plane/main.cpp**).*

## xvi. C++ Plot and Computation: Orthogonal Projection in $\mathbb{R}^2$

We are going to observe standar matrix for orthogonal projection operator on the $x$-axis and $y$-axis in $\mathbb{R}^2$.

**Orthogonal projection on the $x$-axis**

If we perform orthogonal projection on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,0)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x,0)$$

with the standard matrix for the orthogonal projection on the $x$-axis

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

**Orthogonal projection on the $y$-axis**

If we perform orthogonal projection on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (0,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,1)$$

For arbitrary point $(x, y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x, y) = (0, y)$$

with the standard matrix for the orthogonal projection on the $y$-axis

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

The C++ code is quite simple as we only need to draw 3 vectors, we still use **Armadillo** library and gnuplot.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 2 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	X.load("vectorX.txt");
	cout <<"Vector x:" << "\n" << X <<endl;

	// Create standard matrix for orthogonal projection on y-axis
	float matrixAy[N][N] = {};
```

```
// Create standard matrix for orthogonal projection on x—axis
float matrixAx[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              if (i == j)
              {
                     matrixAy[i][j] = 1;
                     matrixAx[i][j] = 1;
              }
              else
              {
                     matrixAy[i][j] = 0;
                     matrixAx[i][j] = 0;
              }
       }
}
matrixAy[0][0] = 0;
matrixAx[1][1] = 0;

arma::mat ArmaAy(N,N,fill::zeros); // Declare matrix ArmaAy of size
    N X N with Armadillo
arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAy[i+j*N] = matrixAy[i][j] ;
              ArmaAx[i+j*N] = matrixAx[i][j] ;
       }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for orthogonal projection on the x—axis:"
    << "\n" << ArmaAy <<endl;
cout <<"Standard Matrix for orthogonal projection on the y—axis:"
    << "\n" << ArmaAx <<endl;

arma::mat Xorthoprojecty = ArmaAy*X ;
arma::mat Xorthoprojectx = ArmaAx*X ;
cout <<"The orthogonal projection of vector x on the x—axis:" << "\
    n" << Xorthoprojectx <<endl;
cout <<"The orthogonal projection of vector x on the y—axis:" << "\
    n" << Xorthoprojecty <<endl;
```

```
        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;

        float o = 0;

        // Create a vector x
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(X[0]);
        pts_D_dy.push_back(X[1]);
        // The orthogonal projection of vector x on the x-axis
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(Xorthoprojectx[0]);
        pts_E_dy.push_back(Xorthoprojectx[1]);
        // The orthogonal projection of vector x on the y-axis
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(Xorthoprojecty[0]);
        pts_F_dy.push_back(Xorthoprojecty[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = x\n";
        gp << "plot '-' with vectors title 'x','-' with vectors title '
            orthogonal projection of x on x-axis' lc 'green', '-' with
            vectors title 'orthogonal projection of x on y-axis'\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
    }
```

**C++ Code 139:** *main.cpp "Orthogonal Projection in 2 dimensional space"*

To compile it, type:

**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.68:** *The plot for orthogonal projection of vector* $\boldsymbol{x}$ *on the x-axis and y-axis (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Orthogonal Projection on Axis/main.cpp**).*

## xvii. C++ Plot and Computation: Orthogonal Projection in $\mathbb{R}^3$

We are going to observe standar matrix for orthogonal projection on the $xy$-plane, $xz$-plane and $yz$-plane in $\mathbb{R}^3$.

**Orthogonal Projection on the $xy$-plane**
If we perform orthogonal projection for standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,0) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (x, y, 0)$$

with the standard matrix for the orthogonal projection on the $xy$-plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

**Orthogonal Projection on the $xz$-plane**
If we perform orthogonal projection for standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,0,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,1) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (x, 0, z)$$

with the standard matrix for the orthogonal projection on the $xz$-plane

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Orthogonal Projection on the $yz$-plane**
If we perform orthogonal projection for standard basis $\boldsymbol{e}_1, \boldsymbol{e}_2$ and $\boldsymbol{e}_3$, the images will be

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (0,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,1) \end{aligned}$$

For arbitrary point $(x, y, z)$ in $\mathbb{R}^3$ the image / transformation will become

$$T(x, y, z) = (0, y, z)$$

with the standard matrix for the orthogonal projection on the $yz$-plane

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The C++ code is not more difficult than the previous section for reflection in 3 dimensional space, the library stays the same.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	X.load("vectorX.txt");
	cout <<"Vector x:" << "\n" << X <<endl;

	// Create standard matrix for orthogonal projection on xy-plane
	float matrixAxy[N][N] = {};
	// Create standard matrix for orthogonal projection on xz-plane
	float matrixAxz[N][N] = {};
	// Create standard matrix for orthogonal projection on yz-plane
```

```
float matrixAyz[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              if (i == j)
              {
                     matrixAxy[i][j] = 1;
                     matrixAxz[i][j] = 1;
                     matrixAyz[i][j] = 1;
              }
              else
              {
                     matrixAxy[i][j] = 0;
                     matrixAxz[i][j] = 0;
                     matrixAyz[i][j] = 0;
              }
       }
}
matrixAxy[2][2] = 0;
matrixAxz[1][1] = 0;
matrixAyz[0][0] = 0;

arma::mat ArmaAxy(N,N,fill::zeros); // Declare matrix ArmaAxy of
    size N X N with Armadillo
arma::mat ArmaAxz(N,N,fill::zeros); // Declare matrix ArmaAxz of
    size N X N with Armadillo
arma::mat ArmaAyz(N,N,fill::zeros); // Declare matrix ArmaAyz of
    size N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAxy[i+j*N] = matrixAxy[i][j] ;
              ArmaAxz[i+j*N] = matrixAxz[i][j] ;
              ArmaAyz[i+j*N] = matrixAyz[i][j] ;
       }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for orthogonal projection on the xy-plane:"
     << "\n" << ArmaAxy <<endl;
cout <<"Standard Matrix for orthogonal projection on the xz-plane:"
     << "\n" << ArmaAxz <<endl;
cout <<"Standard Matrix for orthogonal projection on the yz-plane:"
     << "\n" << ArmaAyz <<endl;
```

```
arma::mat Xorthoprojectxy = ArmaAxy*X ;
arma::mat Xorthoprojectxz = ArmaAxz*X ;
arma::mat Xorthoprojectyz = ArmaAyz*X ;
cout <<"The orthogonal projection of vector x on the xy-plane:" <<
    "\n" << Xorthoprojectxy <<endl;
cout <<"The orthogonal projection of vector x on the xz-plane:" <<
    "\n" << Xorthoprojectxz <<endl;
cout <<"The orthogonal projection of vector x on the yz-plane:" <<
    "\n" << Xorthoprojectyz <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
pts_D_dz.push_back(X[2]);
// The orthogonal projection of vector x on the xy-plane
pts_E_x .push_back(o);
pts_E_y .push_back(o);
```

```
            pts_E_z .push_back(o);
            pts_E_dx.push_back(Xorthoprojectxy[0]);
            pts_E_dy.push_back(Xorthoprojectxy[1]);
            pts_E_dz.push_back(Xorthoprojectxy[2]);
            // The orthogonal projection of vector x on the xz—plane
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(Xorthoprojectxz[0]);
            pts_F_dy.push_back(Xorthoprojectxz[1]);
            pts_F_dz.push_back(Xorthoprojectxz[2]);
            // The orthogonal projection of vector x on the yz—plane
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(Xorthoprojectyz[0]);
            pts_G_dy.push_back(Xorthoprojectyz[1]);
            pts_G_dz.push_back(Xorthoprojectyz[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [0:1]\nset yrange [0:1]\nset zrange [0:1]\n";
            gp << "set xlabel 'x—axis'\n set ylabel 'y—axis'\n set zlabel 'z—
                axis'\n";
            //gp << "set view 80,45,1\n"; // pitch,yaw,zoom for xy plane
            //gp << "set view 80,90,1\n"; // pitch,yaw,zoom for xz plane
            gp << "set view 80,0,1\n"; // pitch,yaw,zoom for yz plane
            // '—' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "splot '—' with vectors title 'x','—' with vectors title '
                orthogonal projection of x on xy—plane' lc 'green', '—' with
                vectors title 'orthogonal projection of x on xz—plane', '—'
                with vectors title 'orthogonal projection of x on yz—plane' lc
                'orange'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
    }
```

**C++ Code 140:** *main.cpp "Orthogonal projection in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:

**make**
**./main**

**Figure 23.69:** *The plot for orthogonal projection of vector* ***x*** *focusing on the xy-plane (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection on Plane/main.cpp**).*

**Figure 23.70:** *The plot for orthogonal projection of vector* $\boldsymbol{x}$ *focusing on the yz-plane (top) and focusing on the xz-plane (bottom) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection on Plane/main.cpp**).*

## xviii. C++ Plot and Computation: Rotation in $\mathbb{R}^2$

Let us consider how to find the standard matrix for the rotation operator

$$T : \mathbb{R}^2 \to \mathbb{R}^2$$

that moves points counterclockwise about the origin through an angle $\theta$.

The images of the standard basis vectors are

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0) = (\cos\ \theta, \sin\ \theta) \\ T(\boldsymbol{e}_2) &= T(0,1) = (-\sin\ \theta, \cos\ \theta) \end{aligned}$$

so the standard matrix for $T$ is

$$[T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2)] \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

We rotate counterclockwise the first basis $\boldsymbol{e}_1$ through an angle of $\theta$ about the origin, then we do

**Figure 23.71:** *The illustration of rotation in* $\mathbb{R}^2$.

the same rotation toward the $\boldsymbol{e}_2$, how we obtain the trigonometry formula is from basic calculus and pythagorean formula.

In keeping with common usage we will denote this operator by $R_\theta$ and call

$$R_\theta = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \tag{23.103}$$

the rotation matrix for $\mathbb{R}^2$.

If $\boldsymbol{x} = (x, y)$ is a vector in $\mathbb{R}^2$, and if $\boldsymbol{w} = (w_1, w_2)$ is its image under the rotation, then the relationship $\boldsymbol{w} = R_\theta \boldsymbol{x}$ can be written in component form as

$$\begin{aligned} w_1 &= x \ \cos \ \theta - y \ \sin \ \theta \\ w_1 &= x \ \sin \ \theta + y \ \cos \ \theta \end{aligned} \tag{23.104}$$

these are called the rotation equations for $\mathbb{R}^2$.

For clockwise rotation, the angle will be negative, thus after replacing $\theta$ with $-\theta$ the clockwise rotation matrix is

$$R_{-\theta} = \begin{bmatrix} \cos \ \theta & \sin \ \theta \\ -\sin \ \theta & \cos \ \theta \end{bmatrix} \tag{23.105}$$

The C++ code is quite easy, as we only need to manually input the entry for the standard counterclockwise rotation matrix in $\mathbb{R}^2$, then we can change the vector $\boldsymbol{x}$ by changing the **vectorX.txt** directly, and to change the angle it can be done from the C++ code. Always remember the trigonometry $(\sin, \cos, \tan)$ in C++ needs radian input that is why we convert from degree to radian before we can do the trigonometry calculation.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

#define theta 60 // Define the degree of rotation
#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
       int r = N;
       int c = N;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
              cout << endl;
       }
}

int main() {
```

```
Gnuplot gp;

// Armadillo
arma::mat X;
X.load("vectorX.txt");
cout <<"Vector x:" << "\n" << X <<endl;

// Create standard matrix for counterclockwise rotation through
    angle theta
float matrixA[N][N] = {};
matrixA[0][0] = cos(theta*DEGTORAD);
matrixA[0][1] = -sin(theta*DEGTORAD);
matrixA[1][0] = sin(theta*DEGTORAD);
matrixA[1][1] = cos(theta*DEGTORAD);

arma::mat ArmaA(N,N,fill::zeros); // Declare matrix ArmaA of size N
    X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaA[i+j*N] = matrixA[i][j] ;
       }
}

cout <<"Standard Matrix for rotation with angle of " << theta << " :
    " << "\n" << ArmaA <<endl;

arma::mat Xrotate = ArmaA*X ;
cout <<"The rotated vector x:" << "\n" << Xrotate <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
```

```
            // The counterclockwise rotation of vector x through angle theta
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_dx.push_back(Xrotate[0]);
            pts_E_dy.push_back(Xrotate[1]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "f(x) = x\n";
            gp << "plot '-' with vectors title 'x','-' with vectors title 'x
                after rotation' lc 'green'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        }
```

**C++ Code 141:** *main.cpp "Rotation in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.72:** *The computation and plot for counterclockwise rotation of vector* $\boldsymbol{x}$ *through angle of* $\theta$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Rotation/main.cpp**).*

## xix. C++ Plot and Computation: Rotations in $\mathbb{R}^3$

A rotation vectors in $\mathbb{R}^3$ is usually described in relation to a ray emanating from the origin, called the axis of rotation. As a vector revolves around the axis of rotation, it sweeps out some portion of a cone.

The angle of rotation, which is measured in the base of a cone, is described as "clockwise" or "counterclockwise" in relation to a viewpoint that is along the axis of rotation looking toward the origin.

As in $\mathbb{R}^2$, angles are positive if they are generated by counterclockwise rotations and negative if they are generated by clockwise rotations.

The most common way of describing a general axis of rotation is to specify a nonzero vector $\boldsymbol{u}$ that runs along the axis of rotation and has its initial point at the origin. The counterclockwise direction for a rotation about the axis can then be determined by a "right-hand rule": If the thumb of the right hand points in the direction of $\boldsymbol{u}$, then the cupped fingers point in a counterclockwise direction.

A rotation operator on $\mathbb{R}^3$ is a matrix operator that rotates each vector in $\mathbb{R}^3$ about some rotation axis through a fixed angle $\theta$. Below are the description of rotation operators on $\mathbb{R}^3$ whose axes of rotation are the positive coordinate axes. For each of these rotations one of the components is unchanged, and the relationships between the other components can be derived by the same procedure used to derive the rotation equations for $\mathbb{R}^2$.

**Rotation about the positive $x$-axis**
The images of the standard basis vectors are

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (1,0,0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0, \cos\ \theta, \sin\ \theta) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0, -\sin\ \theta, \cos\ \theta) \end{aligned}$$

In keeping with common usage the standard matrix for $T$ can be denoted by this operator $R_\theta$

$$R_\theta = [T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2) \mid T(\boldsymbol{e}_3)] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\ \theta & -\sin\ \theta \\ 0 & \sin\ \theta & \cos\ \theta \end{bmatrix} \tag{23.106}$$

this is the rotation matrix about positive $x$-axis for $\mathbb{R}^3$.

If $\boldsymbol{x} = (x, y, z)$ is a vector in $\mathbb{R}^3$, and if $\boldsymbol{w} = (w_1, w_2, w_3)$ is its image under the rotation, then the relationship $\boldsymbol{w} = R_\theta \boldsymbol{x}$ can be written in component form as

$$\begin{aligned} w_1 &= x \\ w_2 &= y\ \cos\ \theta - z\ \sin\ \theta \\ w_3 &= y\ \sin\ \theta + z\ \cos\ \theta \end{aligned} \tag{23.107}$$

these are called the rotation equations about the positive $x$-axis for $\mathbb{R}^3$.

**Rotation about the positive $y$-axis**
The images of the standard basis vectors are

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (\cos\ \theta, 0, -\sin\ \theta) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (0,1,0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (\sin\ \theta, 0, \cos\ \theta) \end{aligned}$$

The standard matrix for $T$ can be denoted by this operator $R_\theta$

$$R_\theta = [T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2) \mid T(\boldsymbol{e}_3)] = \begin{bmatrix} \cos\ \theta & 0 & \sin\ \theta \\ 0 & 1 & 0 \\ -\sin\ \theta & 0 & \cos\ \theta \end{bmatrix} \tag{23.108}$$

this is the rotation matrix about positive $y$-axis for $\mathbb{R}^3$.

If $\boldsymbol{x} = (x, y, z)$ is a vector in $\mathbb{R}^3$, and if $\boldsymbol{w} = (w_1, w_2, w_3)$ is its image under the rotation, then the relationship $\boldsymbol{w} = R_\theta \boldsymbol{x}$ can be written in component form as

$$\begin{aligned} w_1 &= x\ \cos\ \theta + z\ \sin\ \theta \\ w_2 &= y \\ w_3 &= -x\ \sin\ \theta + z\ \cos\ \theta \end{aligned} \tag{23.109}$$

these are called the rotation equations about the positive $y$-axis for $\mathbb{R}^3$.

**Rotation about the positive $z$-axis**
The images of the standard basis vectors are

$$\begin{aligned} T(\boldsymbol{e}_1) &= T(1,0,0) = (\cos\ \theta, \sin\ \theta, 0) \\ T(\boldsymbol{e}_2) &= T(0,1,0) = (-\sin\ \theta, \cos\ \theta, 0) \\ T(\boldsymbol{e}_3) &= T(0,0,1) = (0,0,1) \end{aligned}$$

The standard matrix for $T$ can be denoted by this operator $R_\theta$

$$R_\theta = [T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2) \mid T(\boldsymbol{e}_3)] = \begin{bmatrix} \cos\ \theta & -\sin\ \theta & 0 \\ \sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23.110}$$

this is the rotation matrix about positive $z$-axis for $\mathbb{R}^3$.

If $\boldsymbol{x} = (x, y, z)$ is a vector in $\mathbb{R}^3$, and if $\boldsymbol{w} = (w_1, w_2, w_3)$ is its image under the rotation, then the relationship $\boldsymbol{w} = R_\theta \boldsymbol{x}$ can be written in component form as

$$\begin{aligned} w_1 &= x\ \cos\ \theta - y\ \sin\ \theta \\ w_2 &= x\ \sin\ \theta + y\ \cos\ \theta \\ w_3 &= z \end{aligned} \tag{23.111}$$

these are called the rotation equations about the positive $z$-axis for $\mathbb{R}^3$.

**Yaw, Pitch, and Roll**
In aeronautics and astronautics, the orientation of an aircraft or space shuttle relative to an $xyz$-coordinate system is often described in terms of angles called yaw, pitch, and roll. If an aircraft is flying along the $y$-axis and the $xy$-plane defines the horizontal, then

1. The aircraft's angle of rotation about the $x$-axis is called the pitch.
2. The aircraft's angle of rotation about the $y$-axis is called the roll.
3. The aircraft's angle of rotation about the $z$-axis is called the yaw.

A combination of yaw, pitch, and roll can be achieved by a single rotation about some axis through the origin. This is how a space shuttle makes attitude adjustments, it doesn't perform each rotation separately. It calculates one axis, and rotates about that axis to get the correct orientation. Such rotation maneuvers are used to align an antenna, point the nose toward a celestial object, or position a payload bay for docking.

The C++ code is not difficult, since we stay with **Armadillo** and gnuplot for plotting and put the right entries for each standard matrix.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension
#define theta 60 // Define the degree of rotation
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
```

```
arma::mat X;
X.load("vectorX.txt");
cout <<"Vector x:" << "\n" << X <<endl;

// Create standard matrix for rotation about x-axis
float matrixAx[N][N] = {};
// Create standard matrix for rotation about y-axis
float matrixAy[N][N] = {};
// Create standard matrix for rotation about z-axis
float matrixAz[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              if (i == j)
              {
                     matrixAx[i][j] = 1;
                     matrixAy[i][j] = 1;
                     matrixAz[i][j] = 1;
              }
              else
              {
                     matrixAx[i][j] = 0;
                     matrixAy[i][j] = 0;
                     matrixAz[i][j] = 0;
              }
       }
}
matrixAx[1][1] = cos(theta*DEGTORAD);
matrixAx[1][2] = -sin(theta*DEGTORAD);
matrixAx[2][1] = sin(theta*DEGTORAD);
matrixAx[2][2] = cos(theta*DEGTORAD);

matrixAy[0][0] = cos(theta*DEGTORAD);
matrixAy[0][2] = sin(theta*DEGTORAD);
matrixAy[2][0] = -sin(theta*DEGTORAD);
matrixAy[2][2] = cos(theta*DEGTORAD);

matrixAz[0][0] = cos(theta*DEGTORAD);
matrixAz[0][1] = -sin(theta*DEGTORAD);
matrixAz[1][0] = sin(theta*DEGTORAD);
matrixAz[1][1] = cos(theta*DEGTORAD);

arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
arma::mat ArmaAy(N,N,fill::zeros); // Declare matrix ArmaAy of size
    N X N with Armadillo
arma::mat ArmaAz(N,N,fill::zeros); // Declare matrix ArmaAz of size
```

```
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAx[i+j*N] = matrixAx[i][j] ;
              ArmaAy[i+j*N] = matrixAy[i][j] ;
              ArmaAz[i+j*N] = matrixAz[i][j] ;
       }
}

cout <<"Standard Matrix for rotation about positive x-axis:" << "\n
    " << ArmaAx <<endl;
cout <<"Standard Matrix for rotation about positive y-axis:" << "\n
    " << ArmaAy <<endl;
cout <<"Standard Matrix for rotation about positive z-axis:" << "\n
    " << ArmaAz <<endl;

arma::mat Xrotatex = ArmaAx*X ;
arma::mat Xrotatey = ArmaAy*X ;
arma::mat Xrotatez = ArmaAz*X ;
cout <<"The rotated vector x about positive x-axis:" << "\n" <<
    Xrotatex <<endl;
cout <<"The rotated vector x about positive y-axis:" << "\n" <<
    Xrotatey <<endl;
cout <<"The rotated vector x about positive z-axis:" << "\n" <<
    Xrotatez <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
```

```
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
pts_D_dz.push_back(X[2]);
// Rotated vector x about positive x-axis
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_z .push_back(o);
pts_E_dx.push_back(Xrotatex[0]);
pts_E_dy.push_back(Xrotatex[1]);
pts_E_dz.push_back(Xrotatex[2]);
// Rotated vector x about positive y-axis
pts_F_x .push_back(o);
pts_F_y .push_back(o);
pts_F_z .push_back(o);
pts_F_dx.push_back(Xrotatey[0]);
pts_F_dy.push_back(Xrotatey[1]);
pts_F_dz.push_back(Xrotatey[2]);
// Rotated vector x about positive z-axis
pts_G_x .push_back(o);
pts_G_y .push_back(o);
pts_G_z .push_back(o);
pts_G_dx.push_back(Xrotatez[0]);
pts_G_dy.push_back(Xrotatez[1]);
pts_G_dz.push_back(Xrotatez[2]);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-0.5:1]\nset yrange [-0.5:1]\nset zrange
    [-1:1]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
//gp << "set view 80,45,1\n"; // pitch,yaw,zoom for diagonal x and y
     axis
gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
//gp << "set view 80,0,1\n"; // pitch,yaw,zoom for x axis on front
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
```

```
            gp << "splot '-' with vectors title 'x','-' with vectors title '
                rotated x about positive x-axis' lc 'green', '-' with vectors
                title 'rotated x about positive y-axis', '-' with vectors
                title 'rotated x about positive z-axis' lc 'orange'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
    }
```

**C++ Code 142:** *main.cpp "Rotations in 3 dimensional space about positive axis"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Inside the **int main()** after we declare 2 dimensional array / matrix for rotation of each positive axes, we create identity first for the entries then replace some entries to become the standard matrix.

```
            float matrixAx[N][N] = {};
            float matrixAy[N][N] = {};
            float matrixAz[N][N] = {};
            for (int i = 0; i < N; ++i)
            {
                    for(int j = 0; j<N; ++j)
                    {
                            if (i == j)
                            {
                                    matrixAx[i][j] = 1;
                                    matrixAy[i][j] = 1;
                                    matrixAz[i][j] = 1;
                            }
                            else
                            {
                                    matrixAx[i][j] = 0;
                                    matrixAy[i][j] = 0;
                                    matrixAz[i][j] = 0;
                            }
                    }
```

```
            }
            matrixAx[1][1] = cos(theta*DEGTORAD);
            matrixAx[1][2] = -sin(theta*DEGTORAD);
            matrixAx[2][1] = sin(theta*DEGTORAD);
            matrixAx[2][2] = cos(theta*DEGTORAD);

            matrixAy[0][0] = cos(theta*DEGTORAD);
            matrixAy[0][2] = sin(theta*DEGTORAD);
            matrixAy[2][0] = -sin(theta*DEGTORAD);
            matrixAy[2][2] = cos(theta*DEGTORAD);

            matrixAz[0][0] = cos(theta*DEGTORAD);
            matrixAz[0][1] = -sin(theta*DEGTORAD);
            matrixAz[1][0] = sin(theta*DEGTORAD);
            matrixAz[1][1] = cos(theta*DEGTORAD);
```

**Figure 23.73:** *The computation and plot for counterclockwise rotation of vector* $\boldsymbol{x}$ *through positive x-axis, y-axis, and z-axis with angle of* $\theta$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotations About Positive Axis/main.cpp**).*

## xx. C++ Plot and Computation: Rotation by an Arbitrary Unit Vector $\boldsymbol{u} = (a, b, c)$ in $\mathbb{R}^3$

Find the standard matrix for a rotation of $180^0$ about the axis determined by the vector $\boldsymbol{x} = (2, 2, 1)$. Afterwards determine when a vector $\boldsymbol{w} = (0.3, 0.4, 0.5)$ is rotated $180^0$ counterclockwise about the unit vector $\boldsymbol{x}$.

**Solution:**

The standard matrix for a counterclockwise rotation through an angle $\theta$ about an axis in $\mathbb{R}^3$, which is determined by an arbitrary unit vector $\boldsymbol{u} = (a, b, c)$ that has its initial point at the origin is

$$\begin{bmatrix} a^2(1-\cos\ \theta)+\cos\ \theta & ab(1-\cos\ \theta)-c\ \sin\ \theta & ac(1-\cos\ \theta)+b\ \sin\ \theta \\ ab(1-\cos\ \theta)+c\ \sin\ \theta & b^2(1-\cos\ \theta)+\cos\ \theta & bc(1-\cos\ \theta)-a\ \sin\ \theta \\ ac(1-\cos\ \theta)-b\ \sin\ \theta & bc(1-\cos\ \theta)+a\ \sin\ \theta & c^2(1-\cos\ \theta)+\cos\ \theta \end{bmatrix} \tag{23.112}$$

if you want to derive from this general formula you can try

1. $\boldsymbol{u} = (1, 0, 0)$ will become standard matrix for counterclockwise rotation about the positive $x$-axis.
2. $\boldsymbol{u} = (0, 1, 0)$ will become standard matrix for counterclockwise rotation about the positive $y$-axis.
3. $\boldsymbol{u} = (0, 0, 1)$ will become standard matrix for counterclockwise rotation about the positive $z$-axis.

Now for the problem at hand we will find that

$$\frac{\boldsymbol{x}}{||\boldsymbol{x}||} = \frac{(2, 2, 1)}{\sqrt{2^2+2^2+1^2}} = \left(\frac{2}{3}, \frac{2}{3}, \frac{1}{3}\right)$$

thus

$$(a, b, c) = \left(\frac{2}{3}, \frac{2}{3}, \frac{1}{3}\right)$$

with the knowledge that $\cos\ 180^0 = -1$ and $\sin\ 180^0 = 0$, the matrix is

$$R_{180^0} = \begin{bmatrix} -\frac{1}{9} & \frac{8}{9} & \frac{4}{9} \\ \frac{8}{9} & -\frac{1}{9} & \frac{4}{9} \\ \frac{4}{9} & \frac{4}{9} & -\frac{7}{9} \end{bmatrix}$$

to find the rotated vector $\boldsymbol{w}$, we just need to multiply the standard matrix above with the vector itself

$$R_{180^0}\boldsymbol{w} = \begin{bmatrix} -\frac{1}{9} & \frac{8}{9} & \frac{4}{9} \\ \frac{8}{9} & -\frac{1}{9} & \frac{4}{9} \\ \frac{4}{9} & \frac{4}{9} & -\frac{7}{9} \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.4 \\ 0.5 \end{bmatrix}$$

The C++ code is quite easy to comprehend, this section is talking about general rotation through arbitrary unit vector that can be coned down into rotation about positive $x$-axis or positive $y$-axis or any kind of axis. We know that making a unit vector will never change the direction of the original vector, in fact it makes the calculation easier. The rotation toward $\boldsymbol{x}$ and toward $\frac{\boldsymbol{x}}{||\boldsymbol{x}||}$ (the unit vector of $\boldsymbol{x}$) will produce the same result.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension
#define theta 180 // Define the degree of rotation
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	arma::mat W;
	X.load("vectorX.txt");
	W.load("vectorW.txt");
	cout <<"Vector w:" << "\n" << W <<endl;
	cout <<"Vector x:" << "\n" << X <<endl;

	float a = X[0] / (sqrt(X[0]*X[0] + X[1]*X[1] + X[2]*X[2]));
	float b = X[1] / (sqrt(X[0]*X[0] + X[1]*X[1] + X[2]*X[2]));
	float c = X[2] / (sqrt(X[0]*X[0] + X[1]*X[1] + X[2]*X[2]));

	// Create standard matrix for rotation about arbitrary unit vector x
```

```
float matrixAx[N][N] = {};

matrixAx[0][0] = a*a*(1 - cos(theta*DEGTORAD)) + cos(theta*DEGTORAD
    );
matrixAx[0][1] = a*b*(1 - cos(theta*DEGTORAD)) - c*sin(theta*
    DEGTORAD);
matrixAx[0][2] = a*c*(1 - cos(theta*DEGTORAD)) + b*sin(theta*
    DEGTORAD);
matrixAx[1][0] = a*b*(1 - cos(theta*DEGTORAD)) + c*sin(theta*
    DEGTORAD);
matrixAx[1][1] = b*b*(1 - cos(theta*DEGTORAD)) + cos(theta*DEGTORAD
    );
matrixAx[1][2] = b*c*(1 - cos(theta*DEGTORAD)) - a*sin(theta*
    DEGTORAD);
matrixAx[2][0] = a*c*(1 - cos(theta*DEGTORAD)) - b*sin(theta*
    DEGTORAD);
matrixAx[2][1] = b*c*(1 - cos(theta*DEGTORAD)) + a*sin(theta*
    DEGTORAD);
matrixAx[2][2] = c*c*(1 - cos(theta*DEGTORAD)) + cos(theta*DEGTORAD
    );

arma::mat ArmaUnitvectorx(N,1,fill::zeros); // Declare unit vector x
     of size N X 1 with Armadillo
ArmaUnitvectorx[0] = a ;
ArmaUnitvectorx[1] = b ;
ArmaUnitvectorx[2] = c ;

arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAx[i+j*N] = matrixAx[i][j] ;
       }
}
cout <<"Unit vector x:" << "\n" << ArmaUnitvectorx <<endl;

cout <<"Standard Matrix for rotation about arbitrary unit vector x:"
     << "\n" << ArmaAx <<endl;

arma::mat Wrotatex = ArmaAx*W ;
cout <<"The rotated vector w through an angle " << theta << " degree
     about the unit vector x :" << "\n" << Wrotatex <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
```

```
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;

float o = 0;

// Create a unit vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(a);
pts_D_dy.push_back(b);
pts_D_dz.push_back(c);
// Create a vector w
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_z .push_back(o);
pts_E_dx.push_back(W[0]);
pts_E_dy.push_back(W[1]);
pts_E_dz.push_back(W[2]);
// Rotated vector w about unit vector x
pts_F_x .push_back(o);
pts_F_y .push_back(o);
pts_F_z .push_back(o);
pts_F_dx.push_back(Wrotatex[0]);
pts_F_dy.push_back(Wrotatex[1]);
pts_F_dz.push_back(Wrotatex[2]);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-0.5:1]\nset yrange [-0.5:1]\nset zrange
    [-1:1]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
//gp << "set view 80,45,1\n"; // pitch,yaw,zoom for diagonal x and y
     axis
```

```
        gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
        //gp << "set view 80,0,1\n"; // pitch,yaw,zoom for x axis on front
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "splot '-' with vectors title 'unit vector x','-' with
            vectors title 'w' lc 'blue','-' with vectors title 'w rotated
            counterclockwise about unit vector x' lc 'green'\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
            pts_E_dy, pts_E_dz));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
            pts_F_dy, pts_F_dz));
    }
```

**C++ Code 143:** *main.cpp "Rotation about arbitrary unit vector in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

xterm

```
ctor ]# ./main
Vector w:
   0.3000
   0.4000
   0.5000

Vector x:
   2.0000
   2.0000
   1.0000

Unit vector x:
   0.6667
   0.6667
   0.3333

Standard Matrix for rotation about arbitrary unit vector x:
  -0.1111   0.8889   0.4444
   0.8889  -0.1111   0.4444
   0.4444   0.4444  -0.7778

The rotated vector w through an angle 180 degree about the unit vector x :
   0.5444
   0.4444
  -0.0778
```

**Figure 23.74:** *The computation and plot for counterclockwise rotation of vector* $\boldsymbol{w}$ *through the unit vector* $\boldsymbol{x}$ *with angle of* $\theta$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation About Arbitrary Unit Vector/main.cpp**).*

## xxi. C++ Plot and Computation: Dilation and Contraction on $\mathbb{R}^2$

Dilation and contraction will never change the direction of a vector, thus all the coordinates are multiplied by a factor $k$.

**Dilation on $\mathbb{R}^2$**

If we apply dilation on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (k,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,k)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (kx,ky)$$

with the standard matrix for the dilation

$$\begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix}$$

with $k > 1$.

**Contraction on $\mathbb{R}^2$**

If we apply contraction on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (k,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,k)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (kx,ky)$$

with the standard matrix for the contraction

$$\begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix}$$

with $0 \leq k < 1$.

For the C++ code, we construct two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and we will apply contraction transformation on vector $\boldsymbol{x}$ and dilation transformation on vector $\boldsymbol{y}$.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
```

```
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 2 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	arma::mat Y;
	X.load("vectorX.txt");
	Y.load("vectorY.txt");
	cout <<"Vector x:" << "\n" << X <<endl;
	cout <<"Vector y:" << "\n" << Y <<endl;

	// Create standard matrix for contraction
	float matrixAcontraction[N][N] = {};
	// Create standard matrix for dilation
	float matrixAdilation[N][N] = {};
	for (int i = 0; i < N; ++i)
	{
		for(int j = 0; j<N; ++j)
		{
			if (i == j)
			{
				matrixAcontraction[i][j] = 0.3;
				matrixAdilation[i][j] = 2;
			}
			else
			{
				matrixAcontraction[i][j] = 0;
```

```
                        matrixAdilation[i][j] = 0;
                }
        }
}

arma::mat ArmaAcontraction(N,N,fill::zeros);
arma::mat ArmaAdilation(N,N,fill::zeros);
for (int i = 0; i < N; ++i)
{
        for(int j = 0; j<N; ++j)
        {
                ArmaAcontraction[i+j*N] = matrixAcontraction[i][j] ;
                ArmaAdilation[i+j*N] = matrixAdilation[i][j] ;
        }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for contraction (0 <= k < 1):" << "\n" <<
    ArmaAcontraction <<endl;
cout <<"Standard Matrix for dilation (k>1):" << "\n" <<
    ArmaAdilation <<endl;

arma::mat Xcontraction = ArmaAcontraction*X ;
arma::mat Ydilation = ArmaAdilation*Y ;
cout <<"The contraction of vector x:" << "\n" << Xcontraction <<endl
    ;
cout <<"The dilation of vector y:" << "\n" << Ydilation <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;

float o = 0;
```

```
        // Create a vector x
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_dx.push_back(X[0]);
        pts_C_dy.push_back(X[1]);
        // Create a vector y
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(Y[0]);
        pts_D_dy.push_back(Y[1]);
        // The contraction of vector x
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(Xcontraction[0]);
        pts_E_dy.push_back(Xcontraction[1]);
        // The dilation of vector y
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(Ydilation[0]);
        pts_F_dy.push_back(Ydilation[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [0:2.5]\nset yrange [0:1.5]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "plot '-' with vectors title 'x','-' with vectors title 'y'
            lc 'blue','-' with vectors title 'contraction of x' lc 'green',
             '-' with vectors title 'dilation of y'\n";
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
    }
```

**C++ Code 144:** *main.cpp "Contraction and dilation on 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.75:** *The computation and plot for contraction on vector* $\boldsymbol{x}$ *and dilation on vector* $\boldsymbol{y}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Contraction and Dilation/main.cpp**).*

## xxii. C++ Plot and Computation: Expansion and Compression on $\mathbb{R}^2$

If dilation or contraction make all coordinates multiplied by a factor $k$, in expansion or compression we only multiply one of the coordinate with a factor $k$.

**Compression of $\mathbb{R}^2$ in the $x$-direction**

If we apply compression on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (k,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (kx,y)$$

with the standard matrix for the dilation

$$\begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

with $0 \leq k < 1$.

**Expansion of $\mathbb{R}^2$ in the $x$-direction**

If we apply expansion on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (k,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (kx,y)$$

with the standard matrix for the dilation

$$\begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$

with $k > 1$.

**Compression of $\mathbb{R}^2$ in the $y$-direction**

If we apply compression on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,k)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x,ky)$$

with the standard matrix for the dilation

$$\begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

with $0 \leq k < 1$.

**Expansion of $\mathbb{R}^2$ in the $y$-direction**

If we apply expansion on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,k)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x,ky)$$

with the standard matrix for the dilation

$$\begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}$$

with $k > 1$.

The C++ code can represent 4 transformations of compression and expansion in both the $x$ and $y$-direction. The input vector is only one **vectorX.txt**, the computation is simple just like the previous section on contraction and dilation.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 2 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
        int r = N;
        int c = N;
        for (int i = 0; i < r; i++)
        {
```

```
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}

int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for compression in the x—direction
        float matrixAcompressionx[N][N] = {};
        // Create standard matrix for expansion in the x—direction
        float matrixAexpansionx[N][N] = {};
        // Create standard matrix for compression in the y—direction
        float matrixAcompressiony[N][N] = {};
        // Create standard matrix for expansion in the y—direction
        float matrixAexpansiony[N][N] = {};
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        if (i == j)
                        {
                                matrixAcompressionx[i][j] = 0.3;
                                matrixAexpansionx[i][j] = 2;
                                matrixAcompressiony[i][j] = 0.3;
                                matrixAexpansiony[i][j] = 2;
                        }
                        else
                        {
                                matrixAcompressionx[i][j] = 0;
                                matrixAexpansionx[i][j] = 0;
                                matrixAcompressionx[i][j] = 0;
                                matrixAexpansionx[i][j] = 0;
                        }
                }
        }
        matrixAcompressionx[1][1] = 1;
        matrixAexpansionx[1][1] = 1;
        matrixAcompressiony[0][0] = 1;
        matrixAexpansiony[0][0] = 1;

        arma::mat ArmaAcompressionx(N,N,fill::zeros);
```

```
arma::mat ArmaAexpansionx(N,N,fill::zeros);
arma::mat ArmaAcompressiony(N,N,fill::zeros);
arma::mat ArmaAexpansiony(N,N,fill::zeros);
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAcompressionx[i+j*N] = matrixAcompressionx[i][j] ;
              ArmaAexpansionx[i+j*N] = matrixAexpansionx[i][j] ;
              ArmaAcompressiony[i+j*N] = matrixAcompressiony[i][j] ;
              ArmaAexpansiony[i+j*N] = matrixAexpansiony[i][j] ;
       }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for compression in the x-direction (0 <= k
    < 1):" << "\n" << ArmaAcompressionx <<endl;
cout <<"Standard Matrix for expansion in the x-direction (k>1):" <<
     "\n" << ArmaAexpansionx <<endl;
cout <<"Standard Matrix for compression in the y-direction (0 <= k
    < 1):" << "\n" << ArmaAcompressiony <<endl;
cout <<"Standard Matrix for expansion in the y-direction (k>1):" <<
     "\n" << ArmaAexpansiony <<endl;

arma::mat Xcompressionx = ArmaAcompressionx*X ;
arma::mat Xexpansionx = ArmaAexpansionx*X ;
arma::mat Xcompressiony = ArmaAcompressiony*X ;
arma::mat Xexpansiony = ArmaAexpansiony*X ;
cout <<"The compression of vector x in the x-direction:" << "\n" <<
     Xcompressionx <<endl;
cout <<"The expansion of vector x in the x-direction:" << "\n" <<
    Xexpansionx <<endl;
cout <<"The compression of vector x in the y-direction:" << "\n" <<
     Xcompressiony <<endl;
cout <<"The expansion of vector x in the y-direction:" << "\n" <<
    Xexpansiony <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
```

```
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;

float o = 0;

// Create a vector x
pts_C_x .push_back(o);
pts_C_y .push_back(o);
pts_C_dx.push_back(X[0]);
pts_C_dy.push_back(X[1]);
// The compression of vector x in the x-direction
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_dx.push_back(Xcompressionx[0]);
pts_D_dy.push_back(Xcompressionx[1]);
// The expansion of vector x in the x-direction
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_dx.push_back(Xexpansionx[0]);
pts_E_dy.push_back(Xexpansionx[1]);
// The compression of vector x in the y-direction
pts_F_x .push_back(o);
pts_F_y .push_back(o);
pts_F_dx.push_back(Xcompressiony[0]);
pts_F_dy.push_back(Xcompressiony[1]);
// The expansion of vector x in the y-direction
pts_G_x .push_back(o);
pts_G_y .push_back(o);
pts_G_dx.push_back(Xexpansiony[0]);
pts_G_dy.push_back(Xexpansiony[1]);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [0:2.5]\nset yrange [0:1.5]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
gp << "plot '-' with vectors title 'x','-' with vectors title '
    compression of x in the x-direction' lc 'green', '-' with
    vectors title 'expansion of x in the x-direction' lc 'blue',
```

```
                '—' with vectors title 'compression of x in the y—direction',
                '—' with vectors title 'expansion of x in the y—direction'\n";
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_dx, pts_G_dy));
    }
```

**C++ Code 145:** *main.cpp "Compression and expansion of 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- ———————————————————————————————————————————————————————————————————————

**Figure 23.76:** *The computation and plot for compression and expansion on vector* $\boldsymbol{x}$ *in the x-direction and y-direction (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Compression and Expansion/main.cpp**).*

## xxiii. C++ Plot and Computation: Shears

Shears is an important subject, since by learning this transformation we can predict the tectonic movement and the outcome for the shape of the earth in the future.

**Shears of $\mathbb{R}^2$ in the $x$-direction with factor $k$**

If we apply shear on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,0)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (k,1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x+ky, y)$$

with the standard matrix

$$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

there are two cases for shears in the $x$-direction one with $k > 0$ and the other is with $k < 0$.

**Shears of $\mathbb{R}^2$ in the $y$-direction with factor $k$**

If we apply shear on standard basis $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, the images will be

$$T(\boldsymbol{e}_1) = T(1,0) = (1,k)$$
$$T(\boldsymbol{e}_2) = T(0,1) = (0,1)$$

For arbitrary point $(x,y)$ in $\mathbb{R}^2$ the image / transformation will become

$$T(x,y) = (x, y+kx)$$

with the standard matrix

$$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

there are two cases for shears in the $y$-direction one with $k > 0$ and the other is with $k < 0$.

Basically the shear in the $x$-direction won't have any effect on basis that is parallel with $x$-axis. The same as the shear in the $y$-direction won't have any effect on basis that is parallel with $y$-axis.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"
```

```
#define N 2 // Define the number of dimension

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat B1x;
	arma::mat B2x;
	B1x.load("basis1x.txt");
	B2x.load("basis2x.txt");
	cout <<"Basis 1 for shear :" << "\n" << B1x <<endl;
	cout <<"Basis 2 for shear :" << "\n" << B2x <<endl;

	// Create standard matrix for Shear in the x-direction with k>0
	float matrixAshearxpositive[N][N] = {};
	// Create standard matrix for Shear in the x-direction with k<0
	float matrixAshearxnegative[N][N] = {};
	// Create standard matrix for Shear in the y-direction with k>0
	float matrixAshearypositive[N][N] = {};
	// Create standard matrix for Shear in the y-direction with k<0
	float matrixAshearynegative[N][N] = {};

	for (int i = 0; i < N; ++i)
	{
		for(int j = 0; j<N; ++j)
		{
			if (i == j)
			{
				matrixAshearxpositive[i][j] = 1;
				matrixAshearxnegative[i][j] = 1;
				matrixAshearypositive[i][j] = 1;
				matrixAshearynegative[i][j] = 1;
```

```
                }
                else
                {
                        matrixAshearxpositive[i][j] = 0;
                        matrixAshearxnegative[i][j] = 0;
                        matrixAshearypositive[i][j] = 0;
                        matrixAshearxnegative[i][j] = 0;
                }
        }
}
matrixAshearxpositive[0][1] = 0.3;
matrixAshearxnegative[0][1] = -0.5;
matrixAshearypositive[1][0] = 0.3;
matrixAshearynegative[1][0] = -0.5;

arma::mat ArmaAshearxpositive(N,N,fill::zeros);
arma::mat ArmaAshearxnegative(N,N,fill::zeros);
arma::mat ArmaAshearypositive(N,N,fill::zeros);
arma::mat ArmaAshearynegative(N,N,fill::zeros);
for (int i = 0; i < N; ++i)
{
        for(int j = 0; j<N; ++j)
        {
                ArmaAshearxpositive[i+j*N] = matrixAshearxpositive[i][
                    j] ;
                ArmaAshearxnegative[i+j*N] = matrixAshearxnegative[i][
                    j] ;
                ArmaAshearypositive[i+j*N] = matrixAshearypositive[i][
                    j] ;
                ArmaAshearynegative[i+j*N] = matrixAshearynegative[i][
                    j] ;
        }
}

//PrintMatrix(matrixAy);
//PrintMatrix(matrixAx);
cout <<"Standard Matrix for shear in the x-direction (k > 0):" << "
    \n" << ArmaAshearxpositive <<endl;
cout <<"Standard Matrix for shear in the x-direction (k < 0):" << "
    \n" << ArmaAshearxnegative <<endl;
cout <<"Standard Matrix for shear in the y-direction (k > 0):" << "
    \n" << ArmaAshearypositive <<endl;
cout <<"Standard Matrix for shear in the y-direction (k < 0):" << "
    \n" << ArmaAshearynegative <<endl;

arma::mat B1xshearxpositive = ArmaAshearxpositive*B1x ;
arma::mat B2xshearxpositive = ArmaAshearxpositive*B2x ;
arma::mat B1xshearxnegative = ArmaAshearxnegative*B1x ;
```

```
arma::mat B2xshearxnegative = ArmaAshearxnegative*B2x ;
arma::mat B1yshearypositive = ArmaAshearypositive*B1x ;
arma::mat B2yshearypositive = ArmaAshearypositive*B2x ;
arma::mat B1yshearynegative = ArmaAshearynegative*B1x ;
arma::mat B2yshearynegative = ArmaAshearynegative*B2x ;

cout <<"The shear of Basis 1 in the x-direction (k > 0):" << "\n"
    << B1xshearxpositive <<endl;
cout <<"The shear of Basis 2 in the x-direction (k > 0):" << "\n"
    << B2xshearxpositive <<endl;
cout <<"The shear of Basis 1 in the x-direction (k < 0):" << "\n"
    << B1xshearxnegative <<endl;
cout <<"The shear of Basis 2 in the x-direction (k < 0):" << "\n"
    << B2xshearxnegative <<endl;
cout <<"The shear of Basis 1 in the y-direction (k > 0):" << "\n"
    << B1yshearypositive <<endl;
cout <<"The shear of Basis 2 in the y-direction (k > 0):" << "\n"
    << B2yshearypositive <<endl;
cout <<"The shear of Basis 1 in the y-direction (k < 0):" << "\n"
    << B1yshearynegative <<endl;
cout <<"The shear of Basis 2 in the y-direction (k < 0):" << "\n"
    << B2yshearynegative <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_H_x;
std::vector<double> pts_H_y;
std::vector<double> pts_H_dx;
std::vector<double> pts_H_dy;
```

```
std::vector<double> pts_I_x;
std::vector<double> pts_I_y;
std::vector<double> pts_I_dx;
std::vector<double> pts_I_dy;
std::vector<double> pts_J_x;
std::vector<double> pts_J_y;
std::vector<double> pts_J_dx;
std::vector<double> pts_J_dy;

float o = 0;

// Create a basis 1 for shear in the x-direction
pts_C_x .push_back(o);
pts_C_y .push_back(o);
pts_C_dx.push_back(B1x[0]);
pts_C_dy.push_back(B1x[1]);
// Create a basis 2 for shear in the x-direction
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_dx.push_back(B2x[0]);
pts_D_dy.push_back(B2x[1]);
// Create a basis 1 for shear in the y-direction
pts_E_x .push_back(o);
pts_E_y .push_back(1.3);
pts_E_dx.push_back(B1x[0]);
pts_E_dy.push_back(B1x[1]);
// Create a basis 2 for shear in the y-direction
pts_F_x .push_back(o);
pts_F_y .push_back(1.3);
pts_F_dx.push_back(B2x[0]);
pts_F_dy.push_back(B2x[1]);
// The shear of vector basis 2 in the x-direction with k>0
pts_G_x .push_back(o);
pts_G_y .push_back(o);
pts_G_dx.push_back(B2xshearxpositive[0]);
pts_G_dy.push_back(B2xshearxpositive[1]);
// The shear of vector basis 2 in the x-direction with k<0
pts_H_x .push_back(o);
pts_H_y .push_back(o);
pts_H_dx.push_back(B2xshearxnegative[0]);
pts_H_dy.push_back(B2xshearxnegative[1]);
// The shear of vector basis 1 in the y-direction with k>0
pts_I_x .push_back(o);
pts_I_y .push_back(1.3);
pts_I_dx.push_back(B1yshearypositive[0]);
pts_I_dy.push_back(B1yshearypositive[1]);
// The shear of vector basis 1 in the y-direction with k<0
pts_J_x .push_back(o);
```

```
            pts_J_y .push_back(1.3);
            pts_J_dx.push_back(B1yshearynegative[0]);
            pts_J_dy.push_back(B1yshearynegative[1]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-0.55:2.5]\nset yrange [-0.3:2.5]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "plot '-' with vectors title 'e_{1} for x-direction','-'
                with vectors title 'e_{2} for x-direction' lc 'green', '-'
                with vectors title 'e_{1} for y-direction' lc 'blue', '-' with
                 vectors title 'e_{2} for y-direction', '-' with vectors title
                 'shear in the x-direction (k>0)' lc 'black' dashtype 2, '-'
                with vectors title 'shear in the x-direction (k<0)' lc 'black'
                dashtype 3,'-' with vectors title 'shear in the y-direction (k
                >0)' lc 'red' dashtype 2,'-' with vectors title 'shear in the y
                -direction (k<0)' lc 'red' dashtype 3\n";
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_dx, pts_G_dy));
            gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_dx, pts_H_dy));
            gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_dx, pts_I_dy));
            gp.send1d(boost::make_tuple(pts_J_x, pts_J_y, pts_J_dx, pts_J_dy));
        }
```

**C++ Code 146:** *main.cpp "Shears in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.77:** *The computation and plot for shears on standard basis in the x-direction (below) and in the y-direction (above) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Shears/main.cpp**).*

## xxiv. C++ Plot and Computation: Orthogonal Projection On Lines Through Origin in $\mathbb{R}^2$

These are special cases of the more general operator $T : \mathbb{R}^2 \to \mathbb{R}^2$ that maps each point into its orthogonal projection on a line $L$ through the origin that makes an angle $\theta$ with the positive $x$-axis.

We need to analyze the images of the standard basis vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$.
**Pythagoras and Trigonometry Method**

**Figure 23.78:** *The image of the first standard basis is called $T(\boldsymbol{e}_1)$.*

The first way to know $T(\boldsymbol{e}_1)$ is to use pythagoras and trigonometry from calculus, notice that $\boldsymbol{e}_1$ will have the length of 1 and it will be divided into $1-f$ and $f$ by a line that create two right angle triangles out of the $\triangle O\boldsymbol{e}_1 T(\boldsymbol{e}_1)$. From here the pythagoras is quite simple

$$T(\boldsymbol{e}_1) = (x, y) = (1-f, y)$$

observe the big right angle $\triangle O\boldsymbol{e}_1 T(\boldsymbol{e}_1)$, we will know that the hypothenuse is 1 and we will know that the adjacent side will be $\cos\ \theta$, the opposite side will be $\sin\ \theta$. After that, we can split it into two smaller right angle triangles, we can use either triangle to find the value of $y$ the result will be the same thus

$$\begin{aligned} \cos\ \theta &= \frac{y}{\sin\ \theta} \\ y &= \sin\ \theta\ \cos\ \theta \end{aligned}$$

for the $x$ part we will have

$$\begin{aligned} \sin\ \theta &= \frac{f}{\sin\ \theta} \\ f &= \sin^2\theta \end{aligned}$$

then

$$\begin{aligned} T(\boldsymbol{e}_1) &= (x, y) \\ &= (1 - f, y) \\ &= (1 - \sin^2\theta, \sin\ \theta\ \cos\ \theta) \\ &= (\cos^2\theta, \sin\ \theta\ \cos\ \theta) \end{aligned}$$

That is the end of finding the $T(\boldsymbol{e}_1)$ with Pythagoras from calculus. If we want to find it from another point of view of orthogonal projection method, you can read more on that after we determine $T(\boldsymbol{e}_2)$.

**Figure 23.79:** *The image of the second standard basis is called $T(\boldsymbol{e}_2)$.*

With the same pythagoras method from before, in order to determine $T(\boldsymbol{e}_2)$ we will analyze the right angle $\triangle O\boldsymbol{e}_2T(\boldsymbol{e}_2)$. We can then divide this big right angle triangle into two right angle triangles, thus

$$T(\boldsymbol{e}_2) = (x, y)$$

We know that the length of $\boldsymbol{e}_2$ is 1, hence for the big right angle $\triangle O\boldsymbol{e}_2T(\boldsymbol{e}_2)$ the opposite will have the length of $\sin\ \theta$ and the adjacent will have the length of $\cos\ \theta$. We will now focus on the right angle triangle below (with hypothenuse of $\sin\ \theta$, adjacent $x$, opposite $y$) to find the value of $x$ and $y$, hence

$$\begin{aligned} \cos\ \theta &= \frac{x}{\sin\ \theta} \\ x &= \sin\ \theta\ \cos\ \theta \end{aligned}$$

$$\begin{aligned} \sin\ \theta &= \frac{y}{\sin\ \theta} \\ y &= \sin^2\theta \end{aligned}$$

then

$$\begin{aligned} T(\boldsymbol{e}_2) &= (x, y) \\ &= (\sin\ \theta\ \cos\ \theta, \sin^2\theta) \end{aligned}$$

That is the end of finding the $T(\boldsymbol{e}_2)$ with Pythagoras from calculus.

**Orthogonal Projection Method**

In example 4 of chapter 3.3 section Orthogonality [11], it explains how we obtain the orthogonal projections of the standard basis vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ on the line $l$ that is through origin and make an angle $\theta$ with the positive $x$-axis in $\mathbb{R}^2$. Hence we will re-explain further.

**Figure 23.80:** *The orthogonal projections of the standard basis vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ on the line $l$.*

Our first task is to find the orthogonal projection of $\boldsymbol{e}_1$ along $\vec{a} = \boldsymbol{a}$.

$$||\boldsymbol{a}|| = \sqrt{\sin^2\theta + \cos^2\theta} = 1$$

Now we can use the orthogonal projection formula

$$\begin{aligned} T(\boldsymbol{e}_1) &= \text{proj}_{\boldsymbol{a}}\boldsymbol{e}_1 \\ \text{proj}_{\boldsymbol{a}}\boldsymbol{e}_1 &= \frac{\boldsymbol{e}_1 \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}\boldsymbol{a} \\ &= \frac{(1,0)(\cos\ \theta, \sin\ \theta)}{1}(\cos\ \theta, \sin\ \theta) \\ &= (\cos\ \theta)(\cos\ \theta, \sin\ \theta) \\ &= (\cos^2\theta, \sin\ \theta\ \cos\ \theta) \end{aligned}$$

$$
\begin{aligned}
T(\boldsymbol{e}_2) &= \text{proj}_{\boldsymbol{a}}\boldsymbol{e}_2 \\
\text{proj}_{\boldsymbol{a}}\boldsymbol{e}_2 &= \frac{\boldsymbol{e}_1 \cdot \boldsymbol{a}}{||\boldsymbol{a}||^2}\boldsymbol{a} \\
&= \frac{(0,1)(\cos\ \theta, \sin\ \theta)}{1}(\cos\ \theta, \sin\ \theta) \\
&= (\sin\ \theta)(\cos\ \theta, \sin\ \theta) \\
&= (\sin\ \theta\ \cos\ \theta, \sin^2\theta)
\end{aligned}
$$

That is the end of finding the $T(\boldsymbol{e}_1)$ and $T(\boldsymbol{e}_2)$ with orthogonal projection method.

No matter which method we use, we will obtain the same result so the standard matrix for $T$ is

$$
T = [T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2)] \begin{bmatrix} \cos^2\theta & \sin\ \theta\ \cos\ \theta \\ \sin\ \theta\ \cos\ \theta & \sin^2\theta \end{bmatrix} \tag{23.113}
$$

we often denote this operator by

$$
P_\theta = \begin{bmatrix} \cos^2\theta & \sin\ \theta\ \cos\ \theta \\ \sin\ \theta\ \cos\ \theta & \sin^2\theta \end{bmatrix} \tag{23.114}
$$

The C++ code is very simple, if you already read the previous section on rotation and orthogonal projection on axis, then this one will not need to think too much for the code. The difficulty is more on the above, where you have to find the standard matrix / operator for this transformation.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define theta 30 // Define the degree between the line l and positive x-axis
#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
        int r = N;
        int c = N;
        for (int i = 0; i < r; i++)
        {
```

```
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}

int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for orthogonal projection on line l
        float matrixA[N][N] = {};
        matrixA[0][0] = cos(theta*DEGTORAD)*cos(theta*DEGTORAD);
        matrixA[0][1] = sin(theta*DEGTORAD)*cos(theta*DEGTORAD);
        matrixA[1][0] = sin(theta*DEGTORAD)*cos(theta*DEGTORAD);
        matrixA[1][1] = sin(theta*DEGTORAD)*sin(theta*DEGTORAD);

        arma::mat ArmaA(N,N,fill::zeros); // Declare matrix ArmaA of size N
            X N with Armadillo
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        ArmaA[i+j*N] = matrixA[i][j] ;
                }
        }

        cout <<"Standard Matrix for orthogonal projection on line through
            origin with angle theta with the positive x-axis of " << theta
            << " :" << "\n" << ArmaA <<endl;

        arma::mat Xortho = ArmaA*X ;
        cout <<"The orthogonal projection of vector x on the line l:" << "\n
            " << Xortho <<endl;

        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
```

```
        float o = 0;

        // Create a vector x
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(X[0]);
        pts_D_dy.push_back(X[1]);
        // The orthogonal projection of vector x on line l
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(Xortho[0]);
        pts_E_dy.push_back(Xortho[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        // Put the degree in the equation for f(x) = (theta*DEGTORAD )* x
        gp << "f(x) = tan(30*0.0174532925199432957)*x\n";
        gp << "plot '-' with vectors title 'x','-' with vectors title '
            orthogonal projection of x on l' lc 'green', f(x) title 'l'
            dashtype 3\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
    }
```

**C++ Code 147:** *main.cpp "Orthogonal Projection on lines through origin"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.81:** *The computation and plot for orthogonal projection of vector* $\boldsymbol{x}$ *on line* $l$ *that makes an angle* $\theta$ *with positive x-axis (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Orthogonal Projection on Lines Through the Origin/main.cpp**).*

## xxv. C++ Plot and Computation: Reflection About Lines Through the Origin in $\mathbb{R}^2$

These are special cases of the more general operator

$$H_\theta : \mathbb{R}^2 \to \mathbb{R}^2$$

that maps each point into its reflection about a line $L$ through the origin that makes an angle $\theta$ with the positive $x$-axis

We could find the standard matrix for $H_\theta$ by finding the images of the standard basis vectors, but instead we will take advantage of our work previously on orthogonal projection.

**Orthogonal Projection Method**

**Figure 23.82:** *The the reflection of vector $\boldsymbol{x}$ about the line L that went through the origin and makes an angle $\theta$ with positive x-axis.*

You should be able to see that for every vector $\boldsymbol{x}$ in $\mathbb{R}^n$

$$P_\theta \boldsymbol{x} = \boldsymbol{x} + (P_\theta \boldsymbol{x} - \boldsymbol{x})$$

hence

$$\begin{aligned} H_\theta \boldsymbol{x} &= P_\theta \boldsymbol{x} + (P_\theta \boldsymbol{x} - \boldsymbol{x}) \\ &= 2P_\theta \boldsymbol{x} - \boldsymbol{x} \\ H_\theta \boldsymbol{x} &= (2P_\theta - I)\boldsymbol{x} \end{aligned}$$

thus

$$H_\theta = (2P_\theta - I)$$

$$H_\theta = 2\begin{bmatrix} \cos^2\theta & \sin\ \theta\ \cos\ \theta \\ \sin\ \theta\ \cos\ \theta & \sin^2\theta \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2\cos^2\theta - 1 & 2\sin\ \theta\ \cos\ \theta \\ 2\sin\ \theta\ \cos\ \theta & 2\sin^2\theta - 1 \end{bmatrix}$$

with the help of double angle formula and trigonometry sum

$$\sin(\theta+\theta) = 2\sin\ \theta\ \cos\ \theta$$
$$\sin(2\theta) = 2\sin\ \theta\ \cos\ \theta$$

$$\cos(\theta+\theta) = \cos\ \theta\ \cos\ \theta - \sin\ \theta\ \sin\ \theta$$
$$\cos(2\theta) = \cos^2\theta - \sin^2\theta$$
$$= 2\cos^2\theta - 1$$

and

$$\cos(2\theta) = 1 - 2\sin^2\theta$$
$$-\cos(2\theta) = 2\sin^2\theta - 1$$

and hence we will obtain the standard matrix as

$$H_\theta = \begin{bmatrix} \cos\ 2\theta & \sin\ 2\theta \\ \sin\ 2\theta & -\cos\ 2\theta \end{bmatrix}$$

The C++ code is very easy to comprehend after you can grasp the derivation of the standard matrix for reflection about line that went through the origin.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define theta 30 // Define the degree between the line l and positive x-axis
#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
       int r = N;
       int c = N;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
              cout << endl;
```

```
        }
}

int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for reflection about line through the
            origin
        float matrixA[N][N] = {};
        matrixA[0][0] = cos(2*theta*DEGTORAD);
        matrixA[0][1] = sin(2*theta*DEGTORAD);
        matrixA[1][0] = sin(2*theta*DEGTORAD);
        matrixA[1][1] = -cos(2*theta*DEGTORAD);

        arma::mat ArmaA(N,N,fill::zeros); // Declare matrix ArmaA of size N
            X N with Armadillo
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        ArmaA[i+j*N] = matrixA[i][j] ;
                }
        }

        cout <<"Standard Matrix for reflection about line through origin
            with angle theta with the positive x-axis of " << theta << " :"
             << "\n" << ArmaA <<endl;

        arma::mat Xreflection = ArmaA*X ;
        cout <<"The reflection of vector x about the line l:" << "\n" <<
            Xreflection <<endl;

        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;

        float o = 0;
```

```
        // Create a vector x
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(X[0]);
        pts_D_dy.push_back(X[1]);
        // The reflection of vector x about the line l
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(Xreflection[0]);
        pts_E_dy.push_back(Xreflection[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        // Put the degree in the equation for f(x) = (theta*DEGTORAD )* x
        gp << "f(x) = tan(30*0.0174532925199432957)*x\n";
        gp << "plot '-' with vectors title 'x','-' with vectors title '
            reflection of x about l' lc 'green', f(x) title 'l' dashtype 3\n
            ";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
    }
```

**C++ Code 148:** *main.cpp "Reflection about lines through the origin in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.83:** *The computation and plot for reflection of vector* $\boldsymbol{x}$ *on line* $l$ *that makes an angle* $\theta$ *with positive x-axis (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Reflection About Line Through the Origin/main.cpp**).*

## xxvi. C++ Plot and Computation: Find Angle of Rotation in $\mathbb{R}^2$

It can be proved that if $A$ is a $2 \times 2$ matrix with orthonormal column vectors and for which $\det(A) = 1$, then multiplication by $A$ is a rotation through some angle $\theta$. Verify that

$$A = \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

satisfies the stated conditions and find the angle of rotation.

**Solution:**

The standard matrix of counterclockwise rotation through an angle $\theta$ is

$$R_\theta = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

The column vectors for $R_\theta$ are orthonormal

$$\begin{bmatrix} \cos\ \theta \\ \sin\ \theta \end{bmatrix}, \quad \begin{bmatrix} -\sin\ \theta \\ \cos\ \theta \end{bmatrix}$$

The pairings of $(\pm\cos\ \theta, \pm\sin\ \theta)$ will always have the norm of 1, hence it will be orthonormal. For example if we have vector $\boldsymbol{v} = (\sin\ \frac{\pi}{6}, \cos\ \frac{\pi}{6})$ then we will have

$$\begin{aligned} ||\boldsymbol{v}|| &= \sqrt{\sin^2\left(\frac{\pi}{6}\right) + \cos^2\left(\frac{\pi}{6}\right)} \\ &= \sqrt{\frac{1}{4} + \frac{3}{4}} \\ ||\boldsymbol{v}|| &= 1 \end{aligned}$$

it is proven then that the length of all the pairings of $(\pm\cos\ \theta, \pm\sin\ \theta)$ will always be 1, remember that any two nonzero vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ in $\mathbb{R}^n$ will be orthogonal if

$$\boldsymbol{v} \cdot \boldsymbol{w} = 0$$

we can see that the column vectors in $R_\theta$, which are $(\cos\ \theta, \sin\ \theta)$ and $(-\sin\ \theta, \cos\ \theta)$ are orthogonal as well, thus any vectors in $\mathbb{R}^2$ with this pairings will be orthonormal.

Now we will observe the general determinant for standard matrix of $R_\theta$, by using simple formula of determinant for $2 \times 2$ matrix

$$\det(R_\theta) = \cos^2\theta + \sin^2\theta = 1$$

The standard matrix of $R_\theta$ will have

- Orthonormal column vectors
- $\det(R_\theta) = 1$

hence if matrix $A$ fulfilled the 2 conditions above, then matrix $A$ can be said as a matrix transformation that rotate vector / points through angle of $\theta$ in a counterclockwise direction.

We will first analyze through trigonometry, we know that

| $\theta$ | $\frac{\pi}{4}$ | $\frac{3\pi}{4}$ | $\frac{5\pi}{4}$ | $\frac{7\pi}{4}$ |
|---|---|---|---|---|
| cos | $\frac{1}{\sqrt{2}}$ | $-\frac{1}{\sqrt{2}}$ | $-\frac{1}{\sqrt{2}}$ | $\frac{1}{\sqrt{2}}$ |
| sin | $\frac{1}{\sqrt{2}}$ | $\frac{1}{\sqrt{2}}$ | $-\frac{1}{\sqrt{2}}$ | $-\frac{1}{\sqrt{2}}$ |

**Table 23.3:** *The* sin *and* cos *value for certain radian / degree.*

hence you will only need to use the first column vector to find the angle $\theta$

$$\cos^{-1}\left(-\frac{1}{\sqrt{2}}\right) = \theta$$
$$\sin^{-1}\left(\frac{1}{\sqrt{2}}\right) = \theta$$

in order to match the first column vector of matrix $A$, the only degree that is suitable is $135^0$ or in radian $\frac{3\pi}{4}$. The angle of rotation is $135^0$.

If you want to try the computation for second column vector it will also has the same result.

$$\cos^{-1}\left(-\frac{1}{\sqrt{2}}\right) = \theta$$
$$\sin^{-1} -\left(-\frac{1}{\sqrt{2}}\right) = \sin^{-1}\left(\frac{1}{\sqrt{2}}\right) = \theta$$

This section makes heavy note on the direction that matters the most on rotation case, the length of the vector / axis of reference for the rotation does not matter too much, as we can use length of 1 or length of 2 and still get the correct transformation for the rotation.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#define pi 3.1415926535897
```

```
using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
        int r = N;
        int c = N;
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}

int main() {
        Gnuplot gp;
        float theta;
        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for counterclockwise rotation through
            angle theta
        float matrixA[N][N] = {};
        float a = -1/(sqrt(2));
        float b = -1/(sqrt(2));
        float c = 1/(sqrt(2));
        float d = -1/(sqrt(2));

        matrixA[0][0] = a;
        matrixA[0][1] = b;
        matrixA[1][0] = c;
        matrixA[1][1] = d;

        arma::mat ArmaA(N,N,fill::zeros); // Declare matrix ArmaA of size N
            X N with Armadillo
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        ArmaA[i+j*N] = matrixA[i][j] ;
                }
        }

        cout <<"Matrix A :" << "\n" << ArmaA <<endl;
```

```
arma::mat Xrotate = ArmaA*X ;
cout <<"The rotated vector x:" << "\n" << Xrotate <<endl;

// Formula to find theta
if (a < 0 && c > 0 ) // Second quadrant
{
       cout <<"The angle of rotation is (in degree) :" << " " << (
           asin(c)+ pi/2) * RADTODEG <<endl;
}
else if (a < 0 && c < 0) // third quadrant
{
       cout <<"The angle of rotation is (in degree) :" << " " << (
           acos(-a)+ pi) * RADTODEG <<endl;
}
else if (a > 0 && c < 0) // fourth quadrant
{
       cout <<"The angle of rotation is (in degree) : " << " " << (
           acos(a)+ ((3*pi)/2) ) * RADTODEG <<endl;
}
else if (a > 0 && c > 0) // FIrst quadrant
{
       cout <<"The angle of rotation is (in degree) :" << " " <<
           acos(a) * RADTODEG <<endl;
}

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
// The counterclockwise rotation of vector x through angle theta
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_dx.push_back(Xrotate[0]);
pts_E_dy.push_back(Xrotate[1]);
```

```
            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "f(x) = x\n";
            gp << "plot '-' with vectors title 'x','-' with vectors title 'x
                after counterclockwise rotation' lc 'green'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
    }
```

**C++ Code 149:** *main.cpp "Find the angle of rotation in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In this case we start with the matrix transformation $A$, we do not know the angle of rotation and we will compute it later on.

```
            // Create standard matrix for counterclockwise rotation through
                angle theta
            float matrixA[N][N] = {};
            float a = -1/(sqrt(2));
            float b = -1/(sqrt(2));
            float c = 1/(sqrt(2));
            float d = -1/(sqrt(2));

            matrixA[0][0] = a;
            matrixA[0][1] = b;
            matrixA[1][0] = c;
            matrixA[1][1] = d;
```

- The formula to find $\theta$ in C++ is using no more than **if** and **if else** conditional, since trigonometry is periodic and the sign of each sin and cos are different in each quadrant we can easily determine the angle of rotation here.

```
            if (a < 0 && c > 0 ) // Second quadrant
            {
                cout <<"The angle of rotation is (in degree) :" << " "
                    << (asin(c)+ pi/2) * RADTODEG <<endl;
            }
            else if (a < 0 && c < 0) // third quadrant
            {
```

```
                cout <<"The angle of rotation is (in degree) :" << " "
                    << (acos(-a)+ pi) * RADTODEG <<endl;
        }
        else if (a > 0 && c < 0) // fourth quadrant
        {
                cout <<"The angle of rotation is (in degree) : " << " "
                    << ( acos(a)+ ((3*pi)/2) ) * RADTODEG <<endl;
        }
        else if (a > 0 && c > 0) // FIrst quadrant
        {
                cout <<"The angle of rotation is (in degree) :" << " "
                    << acos(a) * RADTODEG <<endl;
        }
```

```
Vector x:
   0.5000
   0.8000

Matrix A  :
  -0.7071  -0.7071
   0.7071  -0.7071

The rotated vector x:
  -0.9192
  -0.2121

The angle of rotation is (in degree): 135
```

**Figure 23.84:** *The computation and plot for rotation of vector* ***x*** *with the matrix transformation is A, at the end we will be able to determine the angle of rotation (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Rotation Find the Angle/main.cpp**).*

## xxvii. C++ Computation: Angle of Rotation and Trace of Matrix in $\mathbb{R}^3$

If $A$ is a $3 \times 3$ matrix with orthonormal column vectors and for which $\det(A) = 1$, then multiplication by $A$ is a rotation about some axis through some angle $\theta$ that has the standard matrix as follow

$$\begin{bmatrix} a^2(1-\cos\ \theta)+\cos\ \theta & ab(1-\cos\ \theta)-c\ \sin\ \theta & ac(1-\cos\ \theta)+b\ \sin\ \theta \\ ab(1-\cos\ \theta)+c\ \sin\ \theta & b^2(1-\cos\ \theta)+\cos\ \theta & bc(1-\cos\ \theta)-a\ \sin\ \theta \\ ac(1-\cos\ \theta)-b\ \sin\ \theta & bc(1-\cos\ \theta)+a\ \sin\ \theta & c^2(1-\cos\ \theta)+\cos\ \theta \end{bmatrix}$$

Show that the angle of rotation satisfies the equation

$$\cos\ \theta = \frac{\mathrm{tr}(A)-1}{2}$$

**Solution:**
Remember that $\mathrm{tr}(A)$ is the sum of diagonal from matrix $A$ and $\boldsymbol{u} = (a, b, c)$ is an arbitrary unit vector, we can assign a random $(a, b, c)$ that fits the criteria for example $(a, b, c) = (0, 0, 1)$ will resulting in the standard matrix

$$\begin{bmatrix} \cos\ \theta & -\sin\ \theta & 0 \\ \sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23.115}$$

Thus the computation will satisfies the equation

$$\begin{aligned} \cos\ \theta &= \frac{\mathrm{tr}(A)-1}{2} \\ \cos\ \theta &= \frac{\cos\ \theta + \cos\ \theta + 1 - 1}{2} \\ \cos\ \theta &= \cos\ \theta \end{aligned}$$

In the C++ code we can use the computing power of a computer to do symbolic computation, even if in the end we still will need to substitute the value for $a, b, c$ to be able to prove that the equation is valid.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 3 // number of rows
#define C 3 // number of columns

// Driver program
int main()
{
        Symbolic a("a");
        Symbolic b("b");
        Symbolic c("c");
        Symbolic t("t");
```

```
// Construct a symbolic matrix of size 3 X 3
Matrix<Symbolic> B_mat(3,3);
B_mat[0][0] = a*a*(1-cos(t)) + cos(t); B_mat[0][1] = a*b*(1-cos(t)
    ) - c*sin(t); B_mat[0][2] = a*c*(1-cos(t)) + b*sin(t);
B_mat[1][0] = a*b*(1-cos(t)) + c*sin(t); B_mat[1][1] = b*b*(1-cos(
    t)) + cos(t); B_mat[1][2] = b*c*(1-cos(t)) - a*sin(t);
B_mat[2][0] = a*c*(1-cos(t)) - b*sin(t); B_mat[2][1] = b*c*(1-cos(
    t)) + a*sin(t); B_mat[2][2] = c*c*(1-cos(t)) + cos(t);

cout << endl;
cout << "Standard matrix for counterclockwise rotation" <<endl;
cout << "through an angle theta in 3 dimensional space:" <<endl;
cout << "B:\n" << B_mat <<endl;
cout << endl;

Symbolic traceB = 0;
// Operation to compute the trace of a matrix
for (int i = 0; i < R; i++)
{
       traceB += B_mat[i][i];;
}
cout << "tr(B): " << traceB << endl;
cout << endl;
Symbolic x = (traceB-1)/2;
cout << "(tr(B) - 1 ) / 2: \n" << x << endl;
cout << endl;
cout << "(tr(B) - 1 ) / 2 with a=0, b=0, c=1: " << x[a ==0, b==0, c
    ==1] << endl;

// Compute determinant with manual Cofactor expansion along the
    first row
Symbolic b11 = B_mat[0][0];
Symbolic b12 = B_mat[0][1];
Symbolic b13 = B_mat[0][2];
Symbolic C11 = B_mat[1][1]*B_mat[2][2] - B_mat[1][2]*B_mat[2][1];
Symbolic C12 = B_mat[1][0]*B_mat[2][2] - B_mat[1][2]*B_mat[2][0];
Symbolic C13 = B_mat[1][0]*B_mat[2][1] - B_mat[1][1]*B_mat[2][0];
cout << endl;

Symbolic detB = b11*C11 - b12*C12 + b13*C13;
cout << "det(B) in symbolic term:\n" << detB <<endl;
cout << endl;
cout << "det(B) with a=0, b=0, c=1:\n" << detB[ a == 0, b == 0, c
    ==1 ] << " = 1"<< endl;
cout << endl;

return 0;
```

```
    }
```

**C++ Code 150:** *main.cpp "Angle of rotation and trace of matrix in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The operation to compute trace of matrix $A$ symbolically, then we can get the symbolic result and substitute $(a, b, c)$ that has to be unit vector.

```
Symbolic traceB = 0;

for (int i = 0; i < R; i++)
{
       traceB += B_mat[i][i];;
}
cout << "tr(B): " << traceB << endl;
cout << endl;
Symbolic x = (traceB—1)/2;
cout << "(tr(B) — 1 ) / 2: \n" << x << endl;
cout << endl;
cout << "(tr(B) — 1 ) / 2 with a=0, b=0, c=1: " << x[a ==0, b
    ==0, c==1] << endl;
```

```
rminant of Square Symbolic Matrix with SymbolicC++ ]# ./main

Standard matrix for counterclockwise rotation
through an angle θ in 3 dimensional space:
B:
[-a^(2)*cos(θ)+a^(2)+cos(θ)  -a*b*cos(θ)+a*b-c*sin(θ)   -a*c*cos(θ)+a*c+b*sin(θ) ]
[ -a*b*cos(θ)+a*b+c*sin(θ)  -b^(2)*cos(θ)+b^(2)+cos(θ)  -b*c*cos(θ)+b*c-a*sin(θ) ]
[ -a*c*cos(θ)+a*c-b*sin(θ)   -b*c*cos(θ)+b*c+a*sin(θ)  -c^(2)*cos(θ)+c^(2)+cos(θ)]


tr(B): -a^(2)*cos(θ)+a^(2)+3*cos(θ)-b^(2)*cos(θ)+b^(2)-c^(2)*cos(θ)+c^(2)

(tr(B) - 1 ) / 2:
-1/2*a^(2)*cos(θ)+1/2*a^(2)+3/2*cos(θ)-1/2*b^(2)*cos(θ)+1/2*b^(2)-1/2*c^(2)*cos(θ)+1/2*c^(2)-1/2

(tr(B) - 1 ) / 2 with a=0, b=0, c=1: cos(θ)

det(B) in symbolic term:
-a^(2)*cos(θ)^(3)-a^(4)*cos(θ)*sin(θ)^(2)+a^(2)*cos(θ)^(2)+a^(4)*sin(θ)^(2)-cos(θ)^(3)*b^(2)+cos(θ)^(2)*b^(2)-cos(θ)^(3)*c^(2)+cos(θ)^(2)*c
^(2)+cos(θ)^(3)+cos(θ)*a^(2)*sin(θ)^(2)-2*a^(2)*b^(2)*cos(θ)*sin(θ)^(2)+2*a^(2)*b^(2)*sin(θ)^(2)-c^(4)*sin(θ)^(2)*cos(θ)+c^(4)*sin(θ)^(2)+c
^(2)*sin(θ)^(2)*cos(θ)-2*c^(2)*sin(θ)^(2)*b^(2)*cos(θ)+2*c^(2)*sin(θ)^(2)*b^(2)-2*c^(2)*sin(θ)^(2)*a^(2)*cos(θ)+2*c^(2)*sin(θ)^(2)*a^(2)-b^
(4)*sin(θ)^(2)*cos(θ)+b^(4)*sin(θ)^(2)+b^(2)*sin(θ)^(2)*cos(θ)

det(B) with a=0, b=0, c=1:
cos(θ)^(2)+sin(θ)^(2) = 1
```

**Figure 23.85:** *The computation to determine the angle of rotation relation toward the trace of matrix A, the standard matrix for counterclockwise rotation through line that went through origin with an angle of θ in 3 dimensional space (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Find Trace and Determinant of Square Symbolic Matrix with SymbolicC++/main.cpp**).*

## xxviii. C++ Plot and Computation: Determine Axis of Rotation and Angle of Rotation in $\mathbb{R}^3$

If $A$ is a $3 \times 3$ matrix (other than the identity matrix) satisfying the conditions:

1. $\det(A) = 1$.

2. The angle of rotation can be determined with

$$\cos\ \theta = \frac{\text{tr}(A) - 1}{2}$$

3. The matrix consists of orthonormal column vectors.

then multiplication by matrix $A$ is a rotation about an axis of rotation. Suppose that $\boldsymbol{x}$ is a vector we want to rotate we can determine $\boldsymbol{u}$, the axis of rotation with

$$\boldsymbol{u} = A\boldsymbol{x} + A^T\boldsymbol{x} + [1 - \text{tr}(A)]\boldsymbol{x} \tag{23.116}$$

For example, suppose that the matrix $A$ is

$$\begin{bmatrix} \frac{1}{9} & -\frac{4}{9} & \frac{8}{9} \\ \frac{8}{9} & \frac{4}{9} & \frac{1}{9} \\ -\frac{4}{9} & \frac{7}{9} & \frac{4}{9} \end{bmatrix}$$

and vector $\boldsymbol{x}$ is

$$\begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

to determine the axis of rotation we just need to use the formula

$$\boldsymbol{u} = A\boldsymbol{x} + A^T\boldsymbol{x} + [1 - \text{tr}(A)]\boldsymbol{x}$$

we will obtain

$$\boldsymbol{u} = \begin{bmatrix} 1.7778 \\ 3.5556 \\ 3.5556 \end{bmatrix}$$

$\boldsymbol{u}$, the axis of rotation is still having the length the same as vector $\boldsymbol{x}$, thus we will normalize the axis of rotation to obtain the axis of rotation with length / norm of 1, with the formula

$$\frac{\boldsymbol{u}}{||\boldsymbol{u}||}$$

we will obtain the normalized axis of rotation as follow

$$\frac{\boldsymbol{u}}{||\boldsymbol{u}||} = \begin{bmatrix} 0.3333 \\ 0.6667 \\ 0.6667 \end{bmatrix}$$

for the angle of rotation, it can be determined with

$$\begin{aligned}\cos\ \theta &= \frac{\mathrm{tr}(A)-1}{2}\\ &= \frac{1-1}{2}\\ &= 0\\ \theta &= \cos^{-1}(0)\\ \theta &= 90^0\end{aligned}$$

Besides multiplying with matrix $A$ directly, with the knowledge of the axis of rotation and the angle of rotation we can determine the rotated vector $\boldsymbol{x}$ as well.

The C++ code is using **Armadillo** library again for easeness when we need to multiply matrix and vector to determine the axis of rotation, we draw all the vectors that play part in this section from the axis of rotation, the normalized axis of rotation, the vector that needs to be rotated, and the rotated vector.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

#define N 3 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
       int r = N;
       int c = N;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
              cout << endl;
       }
}

int main() {
```

```
Gnuplot gp;

// Armadillo
arma::mat X;
X.load("vectorX.txt");
cout <<"Vector x = " << "\n" << X <<endl;

// Create standard matrix for rotation about arbitrary unit vector x
float matrixAx[N][N] = {};

matrixAx[0][0] = float(1.0/9.0);
matrixAx[0][1] = float(-4.0/9.0);
matrixAx[0][2] = float(8.0/9.0);
matrixAx[1][0] = float(8.0/9.0);
matrixAx[1][1] = float(4.0/9.0);
matrixAx[1][2] = float(1.0/9.0);
matrixAx[2][0] = float(-4.0/9.0);
matrixAx[2][1] = float(7.0/9.0);
matrixAx[2][2] = float(4.0/9.0);

float traceA = 0;
// Operation to compute the trace of a matrix
for (int i = 0; i < N; i++)
{
       traceA += matrixAx[i][i];;
}

arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAx[i+j*N] = matrixAx[i][j] ;
       }
}
cout <<"tr(A) = " << traceA <<endl;
// cout <<"tr(A):" << "\n" << trace(ArmaAx) <<endl;
float theta = acos((traceA - 1)/2);
cout <<"Angle of rotation = acos ( (tr(A) - 1) / 2 ) = " << float(
    theta*RADTODEG) <<endl;

cout <<"Standard Matrix for rotation about an axis of rotation:\n"
    << "A = \n " << ArmaAx <<endl;
cout <<"A^T = " << "\n" << ArmaAx.t() <<endl;

arma::mat axisofrotation(N,1,fill::zeros); // Declare matrix
    axisofrotation of size N X 1 with Armadillo
```

```
axisofrotation = ArmaAx*X + (ArmaAx.t())*X + (1-traceA)*X;
float normaor = sqrt(axisofrotation[0]*axisofrotation[0] +
    axisofrotation[1]*axisofrotation[1] + axisofrotation[2]*
    axisofrotation[2]) ;

cout <<"Axis of rotation = \n" << axisofrotation <<endl;
cout << "Norm for axis of rotation = " << normaor << endl;
cout <<"Normalized axis of rotation (vector with length / norm of 1)
     = \n" << axisofrotation/normaor <<endl;

arma::mat Xrotatex = ArmaAx*X ;
cout <<"The rotated vector x through an angle " << float(theta*
    RADTODEG) << " degree about an axis of rotation : " << "\n" <<
    Xrotatex <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;

float o = 0;

// Create an axis of rotation
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(axisofrotation[0]);
```

```
            pts_D_dy.push_back(axisofrotation[1]);
            pts_D_dz.push_back(axisofrotation[2]);
            // Create a vector x
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(X[0]);
            pts_E_dy.push_back(X[1]);
            pts_E_dz.push_back(X[2]);
            // Rotated vector x about an axis of rotation
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(Xrotatex[0]);
            pts_F_dy.push_back(Xrotatex[1]);
            pts_F_dz.push_back(Xrotatex[2]);
            // Create an axis of rotation
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(axisofrotation[0]/normaor);
            pts_G_dy.push_back(axisofrotation[1]/normaor);
            pts_G_dz.push_back(axisofrotation[2]/normaor);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [-0.5:1]\nset yrange [-0.5:1]\nset zrange
                [-1:1]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";
            //gp << "set view 80,45,1\n"; // pitch,yaw,zoom for diagonal x and y
                 axis
            gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
            //gp << "set view 80,0,1\n"; // pitch,yaw,zoom for x axis on front
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "splot '-' with vectors title 'axis of rotation' dashtype
                2,'-' with vectors title 'x' lc 'blue','-' with vectors title
                'x rotated counterclockwise about an axis of rotation' lc 'green
                ','-' with vectors title 'normalized axis of rotation' lc '
                black'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
```

```
    }
```

**C++ Code 151:** *main.cpp "Determine axis of rotation and angle of rotation in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Inside the main driver **int main()** we are using the syntax of **float(1.0/9.0)** to do the computation and input the entries of matrix $A$ with float number, we cannot just use **1/9** because it will result in 0, that is a big rounding that makes a very big error.

```
        matrixAx[0][0] = float(1.0/9.0);
        matrixAx[0][1] = float(-4.0/9.0);
        matrixAx[0][2] = float(8.0/9.0);
        matrixAx[1][0] = float(8.0/9.0);
        matrixAx[1][1] = float(4.0/9.0);
        matrixAx[1][2] = float(1.0/9.0);
        matrixAx[2][0] = float(-4.0/9.0);
        matrixAx[2][1] = float(7.0/9.0);
        matrixAx[2][2] = float(4.0/9.0);
```

```
Axis of rotation =
   1.7778
   3.5556
   3.5556

Norm for axis of rotation = 5.33333
Normalized axis of rotation (vector with length / norm of 1) =
   0.3333
   0.6667
   0.6667

The rotated vector x through an angle 90 degree about an axis of rotation :
   0.2222
   2.7778
   1.1111
```

**Figure 23.86:** *The plot and computation to determine the angle of rotation and the axis of rotation given we know the matrix A in 3 dimensional space (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation Determine Axis of Rotation/-main.cpp**).*

## xxix. C++ Plot and Computation: Composition of Two Transformations in $\mathbb{R}^2$

We are going to plot composition of several matrix transformations, for this example we will only do two matrix transformations, the first is a counterclockwise rotation of angle $60^0$ then followed by orthogonal projection on $y$-axis.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define theta 60 // Define the degree of rotation
#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
       int r = N;
       int c = N;
       for (int i = 0; i < r; i++)
       {
              for (int j = 0; j < c; j++)
              cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
              cout << endl;
       }
}

int main() {
       Gnuplot gp;

       // Armadillo
       arma::mat X;
       X.load("vectorX.txt");
       cout <<"Vector x:" << "\n" << X <<endl;

       // Create standard matrix for counterclockwise rotation through
           angle theta
       float matrixA[N][N] = {};
       matrixA[0][0] = cos(theta*DEGTORAD);
```

```
matrixA[0][1] = -sin(theta*DEGTORAD);
matrixA[1][0] = sin(theta*DEGTORAD);
matrixA[1][1] = cos(theta*DEGTORAD);

// Create standard matrix for orthogonal projection on y-axis
float matrixAorthoproj[N][N] = {};
// Create standard matrix for orthogonal projection on x-axis
float matrixAorthoprojx[N][N] = {};

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              if (i == j)
              {
                     matrixAorthoproj[i][j] = 1;
                     matrixAorthoprojx[i][j] = 1;
              }
              else
              {
                     matrixAorthoproj[i][j] = 0;
                     matrixAorthoprojx[i][j] = 0;
              }
       }
}
matrixAorthoproj[0][0] = 0;
matrixAorthoprojx[1][1] = 0;

arma::mat ArmaA(N,N,fill::zeros);
arma::mat ArmaAorthoproj(N,N,fill::zeros);
arma::mat ArmaAorthoprojx(N,N,fill::zeros);

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaA[i+j*N] = matrixA[i][j] ;
              ArmaAorthoproj[i+j*N] = matrixAorthoproj[i][j] ;
              ArmaAorthoprojx[i+j*N] = matrixAorthoprojx[i][j] ;
       }
}

// Do the composition transformation
arma::mat XT1 = ArmaA*X ; // rotate first
arma::mat XT2 = ArmaAorthoproj*XT1 ; // orthogonal projection on y-
    axis

cout <<"The first transformation on vector x / T1(x) :" << "\n" <<
```

```
            XT1 <<endl;
        cout <<"The second transformation on vector x / T2(x) :" << "\n" <<
            XT2 <<endl;

        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;

        float o = 0;

        // Create a vector x
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(X[0]);
        pts_D_dy.push_back(X[1]);
        // The first transformation
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(XT1[0]);
        pts_E_dy.push_back(XT1[1]);
        // The second transformation
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(XT2[0]);
        pts_F_dy.push_back(XT2[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = x\n";
        gp << "plot '-' with vectors title 'x','-' with vectors title 'T1(x
            )' lc 'green', '-' with vectors title 'T2(T1(x))' lc 'blue'\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
    }
```

**C++ Code 152:** *main.cpp "Composition of two transformations in two dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

-

**Figure 23.87:** *The plot of composition, the first transformation is a counterclockwise rotation through an angle of* $60^0$*, then the resulting vector will be transformed with an orthogonal projection on y-axis in 2 dimensional space (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Composition/main.cpp**).*

## xxx. C++ Plot and Computation: Composition of Three Transformations in $\mathbb{R}^3$

Find the standard matrix for the operator $T : \mathbb{R}^3 \to \mathbb{R}^3$ that first rotates a vector counterclockwise about the $z$-axis through an angle $\theta$, then reflects the resulting vector about the $yz$-plane, and then projects that vector orthogonally onto the $xy$-plane.

**Solution:**

The operator $T$ can be expressed as the composition

$$T = T_3 \circ T_2 \circ T_1$$

where $T_1$ is the rotation about the $z$-axis, $T_2$ is the reflection about the $yz$-plane, and $T_3$ is the orthogonal projection on the $xy$-plane.

$$[T_1] = \begin{bmatrix} \cos\,\theta & -\sin\,\theta & 0 \\ \sin\,\theta & \cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad [T_2] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad [T_3] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

the standard matrix for $T$ is

$$[T] = [T_3 \circ T_2 \circ T_1] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\,\theta & -\sin\,\theta & 0 \\ \sin\,\theta & \cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos\,\theta & \sin\,\theta & 0 \\ \sin\,\theta & \cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The code is pretty simple, we only need to modify from previous section where we have do transformation in 3 dimensional space for rotation case about positive axis, reflection and orthogonal projection on a plane.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

#define N 3 // Define the number of dimension
#define theta 60 // Define the degree of rotation
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
       int r = N;
       int c = N;
```

```
        for (int i = 0; i < r; i++)
        {
                for (int j = 0; j < c; j++)
                cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
                cout << endl;
        }
}

int main() {
        Gnuplot gp;

        // Armadillo
        arma::mat X;
        X.load("vectorX.txt");
        cout <<"Vector x:" << "\n" << X <<endl;

        // Create standard matrix for reflection about xy-plane
        float matrixAxy[N][N] = {};
        // Create standard matrix for reflection about xz-plane
        float matrixAxz[N][N] = {};
        // Create standard matrix for reflection about yz-plane
        float matrixAyz[N][N] = {};
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        if (i == j)
                        {
                                matrixAxy[i][j] = 1;
                                matrixAxz[i][j] = 1;
                                matrixAyz[i][j] = 1;
                        }
                        else
                        {
                                matrixAxy[i][j] = 0;
                                matrixAxz[i][j] = 0;
                                matrixAyz[i][j] = 0;
                        }
                }
        }
        matrixAxy[2][2] = -1;
        matrixAxz[1][1] = -1;
        matrixAyz[0][0] = -1;

        // Create standard matrix for orthogonal projection on xy-plane
        float matrixAorthogonalprojxy[N][N] = {};
        // Create standard matrix for orthogonal projection on xz-plane
        float matrixAorthogonalprojxz[N][N] = {};
```

```
// Create standard matrix for orthogonal projection on yz—plane
float matrixAorthogonalprojyz[N][N] = {};
for (int i = 0; i < N; ++i)
{
	for(int j = 0; j<N; ++j)
	{
		if (i == j)
		{
			matrixAorthogonalprojxy[i][j] = 1;
			matrixAorthogonalprojxz[i][j] = 1;
			matrixAorthogonalprojyz[i][j] = 1;
		}
		else
		{
			matrixAorthogonalprojxy[i][j] = 0;
			matrixAorthogonalprojxz[i][j] = 0;
			matrixAorthogonalprojyz[i][j] = 0;
		}
	}
}
matrixAorthogonalprojxy[2][2] = 0;
matrixAorthogonalprojxz[1][1] = 0;
matrixAorthogonalprojyz[0][0] = 0;

// Create standard matrix for rotation about positive x—axis
float matrixArotatex[N][N] = {};
// Create standard matrix for rotation about positive y—axis
float matrixArotatey[N][N] = {};
// Create standard matrix for rotation about positive z—axis
float matrixArotatez[N][N] = {};
for (int i = 0; i < N; ++i)
{
	for(int j = 0; j<N; ++j)
	{
		if (i == j)
		{
			matrixArotatex[i][j] = 1;
			matrixArotatey[i][j] = 1;
			matrixArotatez[i][j] = 1;
		}
		else
		{
			matrixArotatex[i][j] = 0;
			matrixArotatey[i][j] = 0;
			matrixArotatez[i][j] = 0;
		}
	}
}
```

```
matrixArotatex[1][1] = cos(theta*DEGTORAD);
matrixArotatex[1][2] = -sin(theta*DEGTORAD);
matrixArotatex[2][1] = sin(theta*DEGTORAD);
matrixArotatex[2][2] = cos(theta*DEGTORAD);

matrixArotatey[0][0] = cos(theta*DEGTORAD);
matrixArotatey[0][2] = -sin(theta*DEGTORAD);
matrixArotatey[0][2] = sin(theta*DEGTORAD);
matrixArotatey[2][2] = cos(theta*DEGTORAD);

matrixArotatez[0][0] = cos(theta*DEGTORAD);
matrixArotatez[0][1] = -sin(theta*DEGTORAD);
matrixArotatez[1][0] = sin(theta*DEGTORAD);
matrixArotatez[1][1] = cos(theta*DEGTORAD);

arma::mat ArmaAxy(N,N,fill::zeros); // Declare matrix ArmaAxy of
    size N X N with Armadillo
arma::mat ArmaAxz(N,N,fill::zeros); // Declare matrix ArmaAxz of
    size N X N with Armadillo
arma::mat ArmaAyz(N,N,fill::zeros); // Declare matrix ArmaAyz of
    size N X N with Armadillo
arma::mat ArmaAorthogonalprojxy(N,N,fill::zeros);
arma::mat ArmaAorthogonalprojxz(N,N,fill::zeros);
arma::mat ArmaAorthogonalprojyz(N,N,fill::zeros);
arma::mat ArmaArotatex(N,N,fill::zeros);
arma::mat ArmaArotatey(N,N,fill::zeros);
arma::mat ArmaArotatez(N,N,fill::zeros);

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAxy[i+j*N] = matrixAxy[i][j] ;
              ArmaAxz[i+j*N] = matrixAxz[i][j] ;
              ArmaAyz[i+j*N] = matrixAyz[i][j] ;
              ArmaAorthogonalprojxy[i+j*N] = matrixAorthogonalprojxy
                  [i][j] ;
              ArmaAorthogonalprojxz[i+j*N] = matrixAorthogonalprojxz
                  [i][j] ;
              ArmaAorthogonalprojyz[i+j*N] = matrixAorthogonalprojyz
                  [i][j] ;
              ArmaArotatex[i+j*N] = matrixArotatex[i][j];
              ArmaArotatey[i+j*N] = matrixArotatey[i][j];
              ArmaArotatez[i+j*N] = matrixArotatez[i][j];

       }
}
```

```
arma::mat Xreflectedxy = ArmaAxy*X ;
arma::mat Xreflectedxz = ArmaAxz*X ;
arma::mat Xreflectedyz = ArmaAyz*X ;

arma::mat Xorthoprojectxy = ArmaAorthogonalprojxy*X ;
arma::mat Xorthoprojectxz = ArmaAorthogonalprojxz*X ;
arma::mat Xorthoprojectyz = ArmaAorthogonalprojyz*X ;

arma::mat Xrotatex = ArmaArotatex*X ;
arma::mat Xrotatey = ArmaArotatey*X ;
arma::mat Xrotatez = ArmaArotatez*X ;

arma::mat T1 = Xrotatez;
arma::mat T2 = ArmaAyz*T1;
arma::mat T3 = ArmaAorthogonalprojxy*T2;

cout <<"T1(x) = Vector x is rotated counterclockwise about the
    positive z-axis with angle of " << theta << " :" << "\n" << T1
    <<endl;
cout <<"T2(T1(x)) = T1(x) is reflected about the yz-plane:" << "\n"
     << T2 <<endl;
cout <<"T3(T2(T1(x))) = The orthogonal projection of T2(T1(x)) on
    the xy-plane:" << "\n" << T3 <<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
```

```
std::vector<double> pts_G_dz;

float o = 0;

// Create a vector x
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
pts_D_dz.push_back(X[2]);
// First transformation
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_z .push_back(o);
pts_E_dx.push_back(T1[0]);
pts_E_dy.push_back(T1[1]);
pts_E_dz.push_back(T1[2]);
// Second transformation
pts_F_x .push_back(o);
pts_F_y .push_back(o);
pts_F_z .push_back(o);
pts_F_dx.push_back(T2[0]);
pts_F_dy.push_back(T2[1]);
pts_F_dz.push_back(T2[2]);
// Third transformation
pts_G_x .push_back(o);
pts_G_y .push_back(o);
pts_G_z .push_back(o);
pts_G_dx.push_back(T3[0]);
pts_G_dy.push_back(T3[1]);
pts_G_dz.push_back(T3[2]);

// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-1:1]\nset yrange [-1:1]\nset zrange [-1:1]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
gp << "set view 60,5,1\n"; // pitch,yaw,zoom
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
gp << "splot '-' with vectors title 'x','-' with vectors title 'T1(
    x)' lc 'green', '-' with vectors title 'T2(T1(x))', '-' with
    vectors title 'T3(T2(T1(x)))' lc 'orange'\n";
gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
    pts_D_dy, pts_D_dz));
gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
    pts_E_dy, pts_E_dz));
gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
```

```
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
    }
```

**C++ Code 153:** *main.cpp "Composition of three transformations in 3 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Inside the **int main()** this is the part where we compute the $T_3 \circ T_2 \circ T_1$.

```
            arma::mat T1 = Xrotatez;
            arma::mat T2 = ArmaAyz*T1;
            arma::mat T3 = ArmaAorthogonalprojxy*T2;
```

```
T1(x) = Vector x is rotated counterclockwise about the positive z-axis with angle of 60 :
  -0.4428
   0.8330
   0.6000

T2(T1(x)) = T1(x) is reflected about the yz-plane:
   0.4428
   0.8330
   0.6000

T3(T2(T1(x))) = The orthogonal projection of T2(T1(x)) on the xy-plane:
   0.4428
   0.8330
        0
```

**Figure 23.88:** *The plot of composition, the first transformation is a counterclockwise rotation of vector* $\boldsymbol{x}$ *through an angle of* $\theta = 60^0$ *about positive z-axis, then the resulting vector will be reflected about the yz-plane, and then projects the resulting vector orthogonally onto the xy-plane in 3 dimensional space (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Composition of 3 Transformations/main.cpp**).*

## xxxi. C++ Computation: Finding $T^{-1}$ In Symbolic Terms

Show that the operator $T : \mathbb{R}^2 \to \mathbb{R}^2$ defined by the equations

$$\begin{aligned} w_1 &= 2x_1 + x_2 \\ w_2 &= 3x_1 + 4x_2 \end{aligned}$$

is one-to-one, and find $T^{-1}(w_1, w_2)$.

**Solution:**

The matrix from these equation is

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

so the standard matrix for $T$ is

$$[T] = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$$

This matrix is invertible (so $T$ is one-to-one) and the standard matrix for $T^{-1}$ is

$$[T^{-1}] = [T]^{-1} = \begin{bmatrix} \frac{4}{5} & -\frac{1}{5} \\ -\frac{3}{5} & \frac{2}{5} \end{bmatrix}$$

Thus

$$[T^{-1}] \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} \frac{4}{5} & -\frac{1}{5} \\ -\frac{3}{5} & \frac{2}{5} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} \frac{4}{5}w_1 - \frac{1}{5}w_2 \\ -\frac{3}{5}w_1 + \frac{2}{5}w_2 \end{bmatrix}$$

from which we conclude that

$$T^{-1}(w_1, w_2) = (\frac{4}{5}w_1 - \frac{1}{5}w_2, -\frac{3}{5}w_1 + \frac{2}{5}w_2)$$

The C++ code is only using **SymbolicC++** to do the symbolical computation.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 2 // number of rows
#define C 2 // number of columns

// Driver program
int main()
{
       Symbolic w1("w1");
       Symbolic w2("w2");
       Matrix<Symbolic> W(R,1);
       W[0][0] = w1;
```

```
        W[1][0] = w2;

        // Construct a symbolic matrix of size R X C
        Matrix<Symbolic> T_mat(R,C);
        T_mat[0][0] = 2;  T_mat[0][1] = 1;
        T_mat[1][0] = 3;  T_mat[1][1] = 4;

        cout << "T:\n" << T_mat <<endl;
        cout << endl;
        Matrix<Symbolic> TI(R,C);
        TI = T_mat.inverse();

        cout << "T^{-1}:\n" << TI <<endl;
        cout << endl;

        cout << "T^{-1} * w :\n" << TI*W <<endl;

        return 0;
    }
```

**C++ Code 154:** *main.cpp "Find inverse of standard matrix in symbolic terms"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

```
 SymbolicC++ ]# ./main
T:
[2 1]
[3 4]


T^{-1}:
[ 4/5 -1/5]
[-3/5  2/5]


T^{-1} * w :
[ 4/5*w1-1/5*w2]
[-3/5*w1+2/5*w2]
```

**Figure 23.89:** *The computation to find the inverse of a square matrix in symbolic terms, we first find the inverse numerically then multiply it with vector* ***w*** *that are filled with symbolic terms (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Finding Inverse for Symbolic Matrix with SymbolicC++/main.cpp**).*

## xxxii. C++ Plot and Computation: Rotation Transformation for 2-Dimensional Picture

In this section we are going to transform an uploaded picture and transform it with matrix transformations that we have learn before, such as reflection, rotation, expansion, shear, expansion in positive $x$-axis direction, etc.

The methods are as follow:

1. Upload image with **SOIL** library
2. We are uploading a 2-dimensional picture, there are 2 pictures that we blend into one, the first is Uncle Scrooge painting dollar, the second one is the most beautiful Goddess Frea the Goddess from Valkyrie Profile, I am her biggest fans. Thus combined it become slike Uncle Scrooge is painting a new Dollar with the face of Freya the Goddess.
3. We are going to use OpenGL, VAO (Vertex Array Object) and VBO (Vertex Buffer Object), then create vertices of a rectangle as the template for the image we are going to put there.
4. We depend on **SFML** library to handle the key press events.
5. We always clear the screen first with **glClearColor**, good code of condunct for OpenGL graphics programming, then we do the transformation for the picture by using **GLM**, to be precise **glm::rotate** function to rotate the picture. The dimension we ar eworking on is 3-dimensional but we make it looks like 2 dimensional, because it is rotating toward positive $z$-axis, some might be fooled and think it looks like we are facing a screen of 2-dimensional that only has $x$ and $y$-axis and the picture is rotating toward $y$ axis, but from the code if you read more, it is rotating towards positive$z$-axis.

Unlike previous codes that only depend on **gnuplot** to plot, with **OpenGL** we are able to make the plot more interactive, hence in the future complex simulation can be done as well, but the learning curve is steeper more codes needed to be written when we use OpenGL,, as one must be fluent with VBO, VAO, vertices, GLEW, GLFW, SFML, GLM terms in order to master this graphics library. If you ever wonder, there is a real life example, Adobe Photoshop is a software that make people to be able to do editing easily, including Movie Maker too, but the engine behind all of them are none other than C++, C, Fortran. Like in this example we learn to rotate a picture 90 degree counterclockwise, then 50 degree clockwise, continous rotation. We wrap the code and the GUI showing made us able to perform the rotation with key press events. Adobe Photoshop is priced at thousand of USD per software because they are able to not only rotate an image, but also to blur the image, to make the image brightened, to crop something we do not want, to make highlight of a hair in the picture of a model, and all that without the user need to dwell and code them in C++, just click, drag and press keyboards and mouse.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
```

```
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
           Texcoord), 0.5);
}
)glsl";

int main()
{
       auto t_start = std::chrono::high_resolution_clock::now();

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL", sf::Style::
           Titlebar | sf::Style::Close, settings);
```

```
// Initialize GLEW
glewExperimental = GL_TRUE;
glewInit();

// Create Vertex Array Object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Create a Vertex Buffer Object and copy the vertex data to it
GLuint vbo;
glGenBuffers(1, &vbo);

GLfloat vertices[] = {
       // Position Color Texcoords
       -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
       0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
       0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
       -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
};

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// Create an element array
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
       0, 1, 2,
       2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);
```

```
        // Link the vertex and fragment shader into a shader program
        GLuint shaderProgram = glCreateProgram();
        glAttachShader(shaderProgram, vertexShader);
        glAttachShader(shaderProgram, fragmentShader);
        glBindFragDataLocation(shaderProgram, 0, "outColor");
        glLinkProgram(shaderProgram);
        glUseProgram(shaderProgram);

        // Specify the layout of the vertex data
        GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
        glEnableVertexAttribArray(posAttrib);
        glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), 0);

        GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
        glEnableVertexAttribArray(colAttrib);
        glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), (void*)(2 * sizeof(GLfloat)));

        GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
        glEnableVertexAttribArray(texAttrib);
        glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), (void*)(5 * sizeof(GLfloat)));

        // Load textures
        GLuint textures[2];
        glGenTextures(2, textures);

        int width, height;
        unsigned char* image;

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textures[0]);
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
            width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, textures[1]);
```

```
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
            &width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
        cout << "Press A to rotate the image counterclockwise" << endl;
        cout << "Press D to rotate the image clockwise" << endl;
        cout << "Press Spacebar to return the image at the original position
            " << endl;

        bool running = true;
        GLfloat rotatedirection = 0.0f;

        while (running)
        {
               sf::Event windowEvent;
               while (window.pollEvent(windowEvent))
               {
                      switch (windowEvent.type)
                      {
                             case sf::Event::Closed:
                             running = false;
                             break;

                             case sf::Event::KeyPressed:
                             if (windowEvent.key.code == sf::Keyboard::A) {
                                    rotatedirection += 90.0f;
                             }
                             if (windowEvent.key.code == sf::Keyboard::D) {
                                    rotatedirection += -50.0f;
                             }
                             if (windowEvent.key.code == sf::Keyboard::Space
                                 ) {
                                    rotatedirection = 0.0f;
                             }
                             cout << "rotation : " << rotatedirection <<
                                 endl;
                             break;
                      }
               }
```

```
            // Clear the screen to black
            glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT);

            // Calculate transformation
            auto t_now = std::chrono::high_resolution_clock::now();
            float time = std::chrono::duration_cast<std::chrono::duration
                <float>>(t_now - t_start).count();

            glm::mat4 transrotate = glm::mat4(1.0f);
            transrotate = glm::rotate(transrotate,
            glm::radians(rotatedirection), // to rotate continuously ->
                time*glm::radians(rotatedirection)
            glm::vec3(0.0f, 0.0f, 1.0f)); // rotate 90 degrees toward
                positive z axis


            glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
                transrotate));


            // Draw a rectangle from the 2 triangles using 6 indices
            glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

            // Swap buffers
            window.display();
        }

        glDeleteTextures(2, textures);

        glDeleteProgram(shaderProgram);
        glDeleteShader(fragmentShader);
        glDeleteShader(vertexShader);

        glDeleteBuffers(1, &ebo);
        glDeleteBuffers(1, &vbo);

        glDeleteVertexArrays(1, &vao);

        window.close();

        return 0;
    }
```

**C++ Code 155:** *main.cpp "Rotation transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**

**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Under the **while (running)** we put the keyboard press events with the help of **SFML** library, it is really handy, we choose to use this over using **GLFW** for this section to handle this functionality.

  We also assign the rotation degree with summing formula of **rotatedirection += 90.0f** instead of **rotatedirection = 90.0f**, the prior will make our rotation to be continuous, thus when you click **A** again it will rotate again, while the later if you click **A** again it won't rotate again.

```
while (running)
{
        sf::Event windowEvent;
        while (window.pollEvent(windowEvent))
        {
                switch (windowEvent.type)
                {
                        case sf::Event::Closed:
                        running = false;
                        break;

                        case sf::Event::KeyPressed:
                        if (windowEvent.key.code == sf::Keyboard::A) {
                                rotatedirection += 90.0f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::D) {
                                rotatedirection += -50.0f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::Space)
                            {
                                rotatedirection = 0.0f;
                        }
                        cout << "rotation : " << rotatedirection << endl;
                        break;
                }
        }
        ...
        ...
}
```

- The last entry in the **transrotate** of **glm::vec3(0.0f, 0.0f, 1.0f)** is for the purpose of making the picture to rotate certain degrees toward positive $z$-axis. That is why without seeing the code, only see the picture and rotation, one might mistaken by thinking the picture is rotating in

2-dimensional plane, and it is rotating toward positive $y$-axis, but in fact it is rotating toward positive $z$-axis.

```
glm::mat4 transrotate = glm::mat4(1.0f);
transrotate = glm::rotate(transrotate,
glm::radians(rotatedirection),
glm::vec3(0.0f, 0.0f, 1.0f));
```

**Figure 23.90:** *The uploaded image can be rotated counterclockwise and clockwise by pressing either A or D with keyboards with expression of the image is in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Rotate Transformation with GLM OpenGL/main.cpp**).*

## xxxiii. C++ Plot and Computation: Translation Transformation for 2-Dimensional Picture

In this section we are going to make uploaded image translated toward $x$ or $y$ axis. In game industry that has been booming since the birth of computer graphics that are getting better. This basic translation is how we move character in game like Grand Theft Auto series, Assassin's Creed, Persona, Final Fantasy series, Suikoden series, Tony Hawk Pro Skater series, and so on.

The same as in the previous section we are still using the same library, if you have read the previous section and try on your own, this one won't be too hard to understand. The key is to comprehend **GLM** functionality for matrix transformation, which feature / command in **GLM** can transform the image into rotation? which feature that can transform the image into compression?

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
	Color = color;
	Texcoord = texcoord;
	gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
```

```
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
           Texcoord), 0.5);
}
)glsl";


int main()
{
       auto t_start = std::chrono::high_resolution_clock::now();

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL", sf::Style::
           Titlebar | sf::Style::Close, settings);

       // Initialize GLEW
       glewExperimental = GL_TRUE;
       glewInit();

       // Create Vertex Array Object
       GLuint vao;
       glGenVertexArrays(1, &vao);
       glBindVertexArray(vao);

       // Create a Vertex Buffer Object and copy the vertex data to it
       GLuint vbo;
       glGenBuffers(1, &vbo);

       GLfloat vertices[] = {
              // Position Color Texcoords
              -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
              0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
              0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
              -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
       };

       glBindBuffer(GL_ARRAY_BUFFER, vbo);
       glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
           GL_STATIC_DRAW);
```

```
// Create an element array
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
       0, 1, 2,
       2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
```

```
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
    width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
    &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
cout << "Press W to translate the image to positive y-axis" << endl
    ;
cout << "Press A to translate the image to negative x-axis" << endl
    ;
cout << "Press S to translate the image to negative y-axis" << endl
    ;
cout << "Press D to translate the image to positive x-axis" << endl
    ;
cout << "Press Spacebar to return the image at the original position
    " << endl;

bool running = true;
GLfloat translatedirectionx = 0.0f;
```

```
        GLfloat translatedirectiony = 0.0f;

        while (running)
        {
                sf::Event windowEvent;
                while (window.pollEvent(windowEvent))
                {
                        switch (windowEvent.type)
                        {
                                case sf::Event::Closed:
                                running = false;
                                break;

                                case sf::Event::KeyPressed:
                                if (windowEvent.key.code == sf::Keyboard::A) {
                                        translatedirectionx += -0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::D) {
                                        translatedirectionx += 0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::S) {
                                        translatedirectiony += -0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::W) {
                                        translatedirectiony += 0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::Space
                                    ) {
                                        translatedirectionx = 0.0f;
                                        translatedirectiony = 0.0f;
                                }
                                cout << "translation toward x-axis : " <<
                                    translatedirectionx << endl;
                                cout << "translation toward y-axis : " <<
                                    translatedirectiony << endl;
                                break;
                        }
                }

                // Clear the screen to black
                glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
                glClear(GL_COLOR_BUFFER_BIT);

                // Calculate transformation
                auto t_now = std::chrono::high_resolution_clock::now();
                float time = std::chrono::duration_cast<std::chrono::duration
                    <float>>(t_now - t_start).count();
```

```
                glm::mat4 translationMat = glm::mat4(1.0f);
                translationMat = glm::translate(translationMat, glm::vec3(
                    translatedirectionx, translatedirectiony, 0.0f));

                glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
                    translationMat));

                // Draw a rectangle from the 2 triangles using 6 indices
                glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

                // Swap buffers
                window.display();
            }

            glDeleteTextures(2, textures);

            glDeleteProgram(shaderProgram);
            glDeleteShader(fragmentShader);
            glDeleteShader(vertexShader);

            glDeleteBuffers(1, &ebo);
            glDeleteBuffers(1, &vbo);

            glDeleteVertexArrays(1, &vao);

            window.close();

            return 0;
        }
```

**C++ Code 156:** *main.cpp "Translation transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We declare variable **GLfloat** for **translatedirectionx** and **translatedirectiony** with initial value of 0. Then under the while loop **while (running) {}** we specify each key press events to translate the picture toward positive or negative $x$ or $y$-axis.

```
            GLfloat translatedirectionx = 0.0f;
            GLfloat translatedirectiony = 0.0f;

            while (running)
```

```
        {
                sf::Event windowEvent;
                while (window.pollEvent(windowEvent))
                {
                        switch (windowEvent.type)
                        {
                                case sf::Event::Closed:
                                running = false;
                                break;

                                case sf::Event::KeyPressed:
                                if (windowEvent.key.code == sf::Keyboard::A
                                    ) {
                                        translatedirectionx += -0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::D
                                    ) {
                                        translatedirectionx += 0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::S
                                    ) {
                                        translatedirectiony += -0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::W
                                    ) {
                                        translatedirectiony += 0.1f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::
                                    Space) {
                                        translatedirectionx = 0.0f;
                                        translatedirectiony = 0.0f;
                                }
                                cout << "translation toward x-axis : " <<
                                    translatedirectionx << endl;
                                cout << "translation toward y-axis : " <<
                                    translatedirectiony << endl;
                                break;
                        }
                }
        }
```

**Figure 23.91:** *The uploaded image can be translated toward x or y axis by pressing either W,A, S or D with keyboards in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Translation Transformation with GLM OpenGL/-main.cpp**).*

## xxxiv. C++ Plot and Computation: Compression and Expansion Transformation for 2-Dimensional Picture

In this section we are going to perform compression and expansion toward the image in 2-dimensional space. In **GLM**, this transformations can be covered with **glm::scale**.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
	Color = color;
	Texcoord = texcoord;
	gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
	outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
	    Texcoord), 0.5);
}
```

```
)glsl";


int main()
{
	auto t_start = std::chrono::high_resolution_clock::now();

	sf::ContextSettings settings;
	settings.depthBits = 24;
	settings.stencilBits = 8;
	settings.majorVersion = 3;
	settings.minorVersion = 3;

	sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL", sf::Style::
	    Titlebar | sf::Style::Close, settings);

	// Initialize GLEW
	glewExperimental = GL_TRUE;
	glewInit();

	// Create Vertex Array Object
	GLuint vao;
	glGenVertexArrays(1, &vao);
	glBindVertexArray(vao);

	// Create a Vertex Buffer Object and copy the vertex data to it
	GLuint vbo;
	glGenBuffers(1, &vbo);

	GLfloat vertices[] = {
		// Position Color Texcoords
		-0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
		0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
		0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
		-0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
	};

	glBindBuffer(GL_ARRAY_BUFFER, vbo);
	glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
	    GL_STATIC_DRAW);

	// Create an element array
	GLuint ebo;
	glGenBuffers(1, &ebo);

	GLuint elements[] = {
		0, 1, 2,
		2, 3, 0
```

```
        };

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
            GL_STATIC_DRAW);

        // Create and compile the vertex shader
        GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertexShader, 1, &vertexSource, NULL);
        glCompileShader(vertexShader);

        // Create and compile the fragment shader
        GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
        glCompileShader(fragmentShader);

        // Link the vertex and fragment shader into a shader program
        GLuint shaderProgram = glCreateProgram();
        glAttachShader(shaderProgram, vertexShader);
        glAttachShader(shaderProgram, fragmentShader);
        glBindFragDataLocation(shaderProgram, 0, "outColor");
        glLinkProgram(shaderProgram);
        glUseProgram(shaderProgram);

        // Specify the layout of the vertex data
        GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
        glEnableVertexAttribArray(posAttrib);
        glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), 0);

        GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
        glEnableVertexAttribArray(colAttrib);
        glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), (void*)(2 * sizeof(GLfloat)));

        GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
        glEnableVertexAttribArray(texAttrib);
        glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
            GLfloat), (void*)(5 * sizeof(GLfloat)));

        // Load textures
        GLuint textures[2];
        glGenTextures(2, textures);

        int width, height;
        unsigned char* image;

        glActiveTexture(GL_TEXTURE0);
```

```
        glBindTexture(GL_TEXTURE_2D, textures[0]);
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
            width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, textures[1]);
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
            &width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
        cout << "Press W to perform expansion in the y-axis direction" <<
            endl;
        cout << "Press A to perform compression in the x-axis direction" <<
             endl;
        cout << "Press S to perform compression in the y-axis direction" <<
             endl;
        cout << "Press D to perform expansion in the x-axis direction" <<
            endl;
        cout << "Press Spacebar to return the image at the original position
            " << endl;

        bool running = true;
        GLfloat scaledirectionx = 1.0f;
        GLfloat scaledirectiony = 1.0f;

        while (running)
        {
               sf::Event windowEvent;
               while (window.pollEvent(windowEvent))
               {
```

```
                switch (windowEvent.type)
                {
                        case sf::Event::Closed:
                        running = false;
                        break;

                        case sf::Event::KeyPressed:
                        if (windowEvent.key.code == sf::Keyboard::A) {
                                scaledirectionx += -0.1f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::D) {
                                scaledirectionx += 0.1f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::S) {
                                scaledirectiony += -0.1f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::W) {
                                scaledirectiony += 0.1f;
                        }
                        if (windowEvent.key.code == sf::Keyboard::Space
                            ) {
                                scaledirectionx = 1.0f;
                                scaledirectiony = 1.0f;
                        }
                        cout << "scale toward x-axis : " <<
                            scaledirectionx << endl;
                        cout << "scale toward y-axis : " <<
                            scaledirectiony << endl;
                        break;
                }
        }

        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Calculate transformation
        auto t_now = std::chrono::high_resolution_clock::now();
        float time = std::chrono::duration_cast<std::chrono::duration
            <float>>(t_now - t_start).count();

        glm::mat4 scaleMat = glm::mat4(1.0f);
        scaleMat = glm::scale(scaleMat, glm::vec3(scaledirectionx,
            scaledirectiony, 0.0f));

        glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
            scaleMat));
```

```
                // Draw a rectangle from the 2 triangles using 6 indices
                glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

                // Swap buffers
                window.display();
            }

            glDeleteTextures(2, textures);

            glDeleteProgram(shaderProgram);
            glDeleteShader(fragmentShader);
            glDeleteShader(vertexShader);

            glDeleteBuffers(1, &ebo);
            glDeleteBuffers(1, &vbo);

            glDeleteVertexArrays(1, &vao);

            window.close();

            return 0;
        }
```

**C++ Code 157:** *main.cpp "Compression and expansion transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- With **glm::scale** we are waiting for input from keyboard press to fill the values of **scaledirectionx** and **scaledirectiony** that will determine whether the picture will be compressed in the $x$-direction or $y$-direction or be expanded in the $x$-direction or $y$-direction.

```
            glm::mat4 scaleMat = glm::mat4(1.0f);
            scaleMat = glm::scale(scaleMat, glm::vec3(scaledirectionx,
                scaledirectiony, 0.0f));

            glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
                scaleMat));
```

**Figure 23.92:** *The uploaded image can be compressed in the x-direction or y-direction, or be expanded in the x-direction or y-direction by pressing either W,A,S or D with keyboards in 2-dimensional space. (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Compression and Expansion Transformation with GLM OpenGL/main.cpp**).*

## xxxv. C++ Plot and Computation: Reflection Transformation for 2-Dimensional Picture

We are going to reflect an image toward $x$-axis, $y$-axis and toward the line $y = x$.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
           Texcoord), 0.5);
}
)glsl";
```

```
int main()
{
        auto t_start = std::chrono::high_resolution_clock::now();

        sf::ContextSettings settings;
        settings.depthBits = 24;
        settings.stencilBits = 8;
        settings.majorVersion = 3;
        settings.minorVersion = 3;

        sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL", sf::Style::
            Titlebar | sf::Style::Close, settings);

        // Initialize GLEW
        glewExperimental = GL_TRUE;
        glewInit();

        // Create Vertex Array Object
        GLuint vao;
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);

        // Create a Vertex Buffer Object and copy the vertex data to it
        GLuint vbo;
        glGenBuffers(1, &vbo);

        GLfloat vertices[] = {
                // Position Color Texcoords
                -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
                0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
                0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
                -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
        };

        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
            GL_STATIC_DRAW);

        // Create an element array
        GLuint ebo;
        glGenBuffers(1, &ebo);

        GLuint elements[] = {
                0, 1, 2,
                2, 3, 0
        };

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
    width, &height, 0, SOIL_LOAD_RGB);
```

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
    &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
cout << "Press A to reflect the image about the x-axis" << endl;
cout << "Press D to reflect the image about the y-axis" << endl;
cout << "Press F to reflect the image about the line y=x" << endl;
cout << "Press Spacebar to return the image at the original position
    " << endl;

bool running = true;
GLfloat reflectiondirection = 0.0f;
GLfloat reflectionxaxis = 1.0f;
GLfloat reflectionyaxis = 0.0f;

while (running)
{
       sf::Event windowEvent;
       while (window.pollEvent(windowEvent))
       {
              switch (windowEvent.type)
              {
                     case sf::Event::Closed:
                     running = false;
                     break;

                     case sf::Event::KeyPressed:
```

```
                    if (windowEvent.key.code == sf::Keyboard::A) {
                        reflectiondirection += 180.0f;
                        reflectionxaxis = 0.0f;
                        reflectionyaxis = 1.0f;
                        cout << "Reflection toward y-axis " <<
                            endl;
                    }
                    if (windowEvent.key.code == sf::Keyboard::D) {
                        reflectiondirection += 180.0f;
                        reflectionxaxis = 1.0f;
                        reflectionyaxis = 0.0f;
                        cout << "Reflection toward x-axis " <<
                            endl;
                    }
                    if (windowEvent.key.code == sf::Keyboard::F) {
                        reflectiondirection += 180.0f;
                        reflectionxaxis = 1.0f;
                        reflectionyaxis = 1.0f;
                        cout << "Reflection toward the line y=x
                            " << endl;
                    }
                    if (windowEvent.key.code == sf::Keyboard::Space
                        ) {
                        reflectionxaxis = 1.0f;
                        reflectionyaxis = 0.0f;
                        reflectiondirection = 0.0f;
                        cout << "Normal position" << endl;
                    }
                    break;
                }
            }

            // Clear the screen to black
            glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT);

            // Calculate transformation
            auto t_now = std::chrono::high_resolution_clock::now();
            float time = std::chrono::duration_cast<std::chrono::duration
                <float>>(t_now - t_start).count();

            glm::mat4 reflectionMat = glm::mat4(1.0f);
            reflectionMat = glm::rotate(reflectionMat,
            glm::radians(reflectiondirection), // to see animation time*
                glm::radians(reflectiondirection)
            glm::vec3(reflectionxaxis, reflectionyaxis, 0.0f));

            glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
```

```
                    reflectionMat));


                // Draw a rectangle from the 2 triangles using 6 indices
                glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

                // Swap buffers
                window.display();
            }

            glDeleteTextures(2, textures);

            glDeleteProgram(shaderProgram);
            glDeleteShader(fragmentShader);
            glDeleteShader(vertexShader);

            glDeleteBuffers(1, &ebo);
            glDeleteBuffers(1, &vbo);

            glDeleteVertexArrays(1, &vao);

            window.close();

            return 0;
        }
```

**C++ Code 158:** *main.cpp "Reflection transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We declare 3 variables / parameters here as **reflectiondirection**, **reflectionxaxis**, and **reflectionyaxis** to determine whether the picture will be reflected toward $x$ axis or $y$ axis or the line $y = x$.

  In **GLM**, the function that can cover reflection transformation is rotation, but the degree should be $180^0$ thus that is why we put **reflectiondirection += 180.0f** and it is summing, so if we press the same keyboard button again, it will goes back to its' original position.

```
            GLfloat reflectiondirection = 0.0f;
            GLfloat reflectionxaxis = 1.0f;
            GLfloat reflectionyaxis = 0.0f;
```

```
        while (running)
        {
                sf::Event windowEvent;
                while (window.pollEvent(windowEvent))
                {
                        switch (windowEvent.type)
                        {
                                case sf::Event::Closed:
                                running = false;
                                break;

                                case sf::Event::KeyPressed:
                                if (windowEvent.key.code == sf::Keyboard::A
                                    ) {
                                        reflectiondirection += 180.0f;
                                        reflectionxaxis = 0.0f;
                                        reflectionyaxis = 1.0f;
                                        cout << "Reflection toward y-axis "
                                             << endl;
                                }
                                if (windowEvent.key.code == sf::Keyboard::D
                                    ) {
                                        reflectiondirection += 180.0f;
                                        reflectionxaxis = 1.0f;
                                        reflectionyaxis = 0.0f;
                                        cout << "Reflection toward x-axis "
                                             << endl;
                                }
                                if (windowEvent.key.code == sf::Keyboard::F
                                    ) {
                                        reflectiondirection += 180.0f;
                                        reflectionxaxis = 1.0f;
                                        reflectionyaxis = 1.0f;
                                        cout << "Reflection toward the line
                                            y=x " << endl;
                                }
                                if (windowEvent.key.code == sf::Keyboard::
                                    Space) {
                                        reflectionxaxis = 1.0f;
                                        reflectionyaxis = 0.0f;
                                        reflectiondirection = 0.0f;
                                        cout << "Normal position" << endl;
                                }
                                break;
                        }
                }

                // Clear the screen to black
```

```
                glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
                glClear(GL_COLOR_BUFFER_BIT);

                // Calculate transformation
                auto t_now = std::chrono::high_resolution_clock::now();
                float time = std::chrono::duration_cast<std::chrono::
                    duration<float>>(t_now - t_start).count();

                glm::mat4 reflectionMat = glm::mat4(1.0f);
                reflectionMat = glm::rotate(reflectionMat,
                glm::radians(reflectiondirection),
                glm::vec3(reflectionxaxis, reflectionyaxis, 0.0f));

                glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr
                    (reflectionMat));

                ...
                ...
        }
```

**Figure 23.93:** *The uploaded image can be reflected toward x-axis or y-axis or the line $y = x$ by pressing either A, D or F with keyboards in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Reflection Transformation with GLM OpenGL /main.cpp**).*

**Figure 23.94:** *The uploaded image can be reflected toward x-axis or y-axis or the line $y = x$ by pressing either A, D or F with keyboards in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Reflection Transformation with GLM OpenGL /main.cpp**).*

## xxxvi. C++ Plot and Computation: Shear Transformation for 2-Dimensional Picture

In 2-dimensional space there are two kinds of shear: shear in the $x$-direction and shear in the $y$-direction.

The transform matrix for shear in the $x$-direction:

$$T = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

The transform matrix for shear in the $x$-direction:

$$T = \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

the factor $k$ could be negative or positive.

The C++ code for this section is still depending on **SFML, GLM, GLEW**, and **OpenGL**, and since the computation to do shear transformation is not in the default / generic template **<glm/gtc/matrix_transform.hpp>** we need to include **<glm/gtx/transform2.hpp>** at the beginning of the C++ code.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform2.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
	Color = color;
	Texcoord = texcoord;
```

```
        gl_Position = trans * vec4(position, 0.0, 1.0);
    }
    )glsl";
    const GLchar* fragmentSource = R"glsl(
    #version 330 core
    in vec3 Color;
    in vec2 Texcoord;
    out vec4 outColor;
    uniform sampler2D texFreya;
    uniform sampler2D texScrooge;
    void main()
    {
        outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
            Texcoord), 0.5);
    }
    )glsl";

    int main()
    {
        auto t_start = std::chrono::high_resolution_clock::now();

        sf::ContextSettings settings;
        settings.depthBits = 24;
        settings.stencilBits = 8;
        settings.majorVersion = 3;
        settings.minorVersion = 3;

        sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL", sf::Style::
            Titlebar | sf::Style::Close, settings);

        // Initialize GLEW
        glewExperimental = GL_TRUE;
        glewInit();

        // Create Vertex Array Object
        GLuint vao;
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);

        // Create a Vertex Buffer Object and copy the vertex data to it
        GLuint vbo;
        glGenBuffers(1, &vbo);

        GLfloat vertices[] = {
            // Position Color Texcoords
            -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
            0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
            0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
```

```
        -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
};

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// Create an element array
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
        0, 1, 2,
        2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(2 * sizeof(GLfloat)));
```

```
GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
    width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
    &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
cout << "Press A to subtract 1 to factor k" << endl;
cout << "Press D to add 1 to factor k" << endl;
cout << "Press Spacebar to return the image at the original position
    " << endl;

bool running = true;
```

```
        GLfloat k = 0.0f;

        while (running)
        {
                sf::Event windowEvent;
                while (window.pollEvent(windowEvent))
                {
                        switch (windowEvent.type)
                        {
                                case sf::Event::Closed:
                                running = false;
                                break;

                                case sf::Event::KeyPressed:
                                if (windowEvent.key.code == sf::Keyboard::D) {
                                        k += 1.0f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::A) {
                                        k += -1.0f;
                                }
                                if (windowEvent.key.code == sf::Keyboard::Space
                                    ) {
                                        k = 0.0f;
                                }
                                cout << "k : " << k << endl;
                                break;
                        }
                }

                // Clear the screen to black
                glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
                glClear(GL_COLOR_BUFFER_BIT);

                // Calculate transformation
                auto t_now = std::chrono::high_resolution_clock::now();
                float time = std::chrono::duration_cast<std::chrono::duration
                    <float>>(t_now - t_start).count();

                glm::mat4 shearMat = glm::mat4(1.0f);

                // change to shearX3D -> Shear in the y-direction
                // change to shearY3D -> Shear in the x-direction
                shearMat = glm::shearY3D(shearMat,
                k, // the factor k
                0.0f); // z = 0 so it will looks like we are in 2-
                    dimensional space

                glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
```

```
                    shearMat));


                // Draw a rectangle from the 2 triangles using 6 indices
                glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

                // Swap buffers
                window.display();
            }

            glDeleteTextures(2, textures);

            glDeleteProgram(shaderProgram);
            glDeleteShader(fragmentShader);
            glDeleteShader(vertexShader);

            glDeleteBuffers(1, &ebo);
            glDeleteBuffers(1, &vbo);

            glDeleteVertexArrays(1, &vao);

            window.close();

            return 0;
        }
```

**C++ Code 159:** *main.cpp "Shear transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- If you want to compute the shear transformation with **GLM**, then you have to include <glm/gtx/transform2.hpp> to use the features of the extension at **/usr/include/glm/gtx/transform2.hpp** (the location for the include header for **GLM** library that contains the feature to do shear transformation).

```
//! Transforms a matrix with a shearing on X axis
//! From GLM_GTX_transform2 extension.
template<typename T, qualifier Q>
GLM_FUNC_DECL mat<4, 4, T, Q> shearX3D(mat<4, 4, T, Q> const& m
    , T y, T z);

//! Transforms a matrix with a shearing on Y axis.
```

```
//! From GLM_GTX_transform2 extension.
template<typename T, qualifier Q>
GLM_FUNC_DECL mat<4, 4, T, Q> shearY3D(mat<4, 4, T, Q> const& m
    , T x, T z);
```

You see from the **/usr/include/glm/gtx/transform2.hpp** we have options to use **shearX3D** or **shearY3D** to make shear in the $x$-direction and the $y$-direction. The image we uploaded is actually in 3-dimensional space, but we want to make it looks like 2-dimensional space, we still use matrix $4 \times 4$ and disregard the factor $k$ for the third axis, so it will looks like the shear is in 2-dimensional space.

- To make the shear transformation in the $x$-direction we use the command under the **while (running) { }**

```
glm::mat4 shearMat = glm::mat4(1.0f);

shearMat = glm::shearY3D(shearMat,
k,
0.0f);
```

To make the shear transformation in the $y$-direction we use this command under the **while (running) { }**

```
glm::mat4 shearMat = glm::mat4(1.0f);

shearMat = glm::shearX3D(shearMat,
k,
0.0f);
```

**Figure 23.95:** *The uploaded image can be transformed with shear in the x-direction or y-direction, and change the value k by pressing either A or D with keyboards in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Shear Transformation with GLM OpenGL/main.cpp**).*

## xxxvii. C++ Plot and Computation: Composition Transformation for 2-Dimensional Picture

We will now compare whether composition of two transformations commute or not, the comparison are:

1. Shear by a factor of 2 in the $x$-direction.
2. Reflection about the line $y = x$

We will find the standard matri and plot the image of the composition of each of the transformation. The standar matrix for the shear is

$$A_1 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

and the standard matrix for reflection about the line $y = x$ is

$$A_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Thus, for the first composition, the standard matrix for the shear followed by the reflection is

$$A_2 A_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

and the standard matrix for the reflection followed by the shear is

$$A_1 A_2 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

the computations show that

$$A_1 A_2 \neq A_2 A_1$$

so the standard matrices, and hence the operators, do not commute. The two operators will produce different result of images.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform2.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
```

```
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
           Texcoord), 0.5);
}
)glsl";

int main()
{
       auto t_start = std::chrono::high_resolution_clock::now();

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(1200, 800, 32), "OpenGL", sf::Style
           ::Titlebar | sf::Style::Close, settings);

       // Initialize GLEW
       glewExperimental = GL_TRUE;
       glewInit();

       // Create Vertex Array Object
       GLuint vao;
```

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Create a Vertex Buffer Object and copy the vertex data to it
GLuint vbo;
glGenBuffers(1, &vbo);

GLfloat vertices[] = {
       // Position Color Texcoords
       -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
       0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
       0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
       -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
};

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// Create an element array
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
       0, 1, 2,
       2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
```

```
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
    width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
    &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);
```

```
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
        cout << "Press A to shear the image in the x-direction, then
            reflect it about the line y=x" << endl;
        // cout << "Press A to reflect the image about the line y=x, then
            shear it in the x-direction" << endl;
        cout << "Press Spacebar to return the image at the original position
            " << endl;

        bool running = true;
        GLfloat reflectiondirection = 0.0f;
        GLfloat reflectionxaxis = 1.0f;
        GLfloat reflectionyaxis = 0.0f;
        GLfloat k = 0.0f;

        while (running)
        {
               sf::Event windowEvent;
               while (window.pollEvent(windowEvent))
               {
                      switch (windowEvent.type)
                      {
                             case sf::Event::Closed:
                             running = false;
                             break;

                             case sf::Event::KeyPressed:
                             if (windowEvent.key.code == sf::Keyboard::A) {
                                    reflectiondirection += 180.0f;
                                    reflectionxaxis = 1.0f;
                                    reflectionyaxis = 1.0f;
                                    k = 2.0f;
                                    cout << "Shear the image in the x-
                                        direction, then reflect it about the
                                         line y=x " << endl;
                                    // cout << "Reflect the image about the
                                        line y=x, then shear it in the x-
                                        direction " << endl;
                             }
                             if (windowEvent.key.code == sf::Keyboard::Space
                                 ) {
                                    reflectionxaxis = 1.0f;
                                    reflectionyaxis = 0.0f;
                                    reflectiondirection = 0.0f;
```

```
                                k = 0.0f;
                                cout << "Normal position" << endl;
                        }
                        break;
                }
        }

        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Calculate transformation
        auto t_now = std::chrono::high_resolution_clock::now();
        float time = std::chrono::duration_cast<std::chrono::duration
            <float>>(t_now - t_start).count();

        glm::mat4 reflectionMat = glm::mat4(1.0f);
        reflectionMat = glm::rotate(reflectionMat,
        glm::radians(reflectiondirection), // to see animation time*
            glm::radians(reflectiondirection)
        glm::vec3(reflectionxaxis, reflectionyaxis, 0.0f));

        glm::mat4 shearMat = glm::mat4(1.0f);
        // change to shearY3D -> Shear in the x-direction
        shearMat = glm::shearY3D(shearMat,
        k, // the factor k
        0.0f); // z = 0 so it will looks like we are in 2-
            dimensional space

        glm::mat4 resultMat = glm::mat4(1.0f);

        // First composition
        resultMat = reflectionMat*shearMat;
        // Second composition
        // resultMat = shearMat*reflectionMat;
        glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
            resultMat));

        // Draw a rectangle from the 2 triangles using 6 indices
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

        // Swap buffers
        window.display();
    }

    glDeleteTextures(2, textures);

    glDeleteProgram(shaderProgram);
```

```
        glDeleteShader(fragmentShader);
        glDeleteShader(vertexShader);

        glDeleteBuffers(1, &ebo);
        glDeleteBuffers(1, &vbo);

        glDeleteVertexArrays(1, &vao);

        window.close();

        return 0;
    }
```

**C++ Code 160:** *main.cpp "Composition transformation for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- When we want to plot the first composition transformation $A_2A_1$ which is shear followed by reflection we uncomment the related lines that can be found below (e.g. **resultMat = reflectionMat*shearMat**) and comment the line that we do not want to compute and plot, if we want to plot the second composition transformation $A_1A_2$ which is reflection followed by shear we uncomment the related lines (e.g. **resultMat = shearMat*reflectionMat**).

```
        cout << "Press A to shear the image in the x-direction, then
            reflect it about the line y=x" << endl;
        // cout << "Press A to reflect the image about the line y=x,
            then shear it in the x-direction" << endl;

        ...
        ...

        case sf::Event::KeyPressed:
        if (windowEvent.key.code == sf::Keyboard::A) {
               reflectiondirection += 180.0f;
               reflectionxaxis = 1.0f;
               reflectionyaxis = 1.0f;
               k = 2.0f;
               cout << "Shear the image in the x-direction, then
                   reflect it about the line y=x " << endl;
               // cout << "Reflect the image about the line y=x, then
                   shear it in the x-direction " << endl;
        }
```

```
...
...

// First composition
resultMat = reflectionMat*shearMat;
// Second composition
// resultMat = shearMat*reflectionMat;
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
    resultMat));
```

**Figure 23.96:** *The uploaded image that is transformed with reflection about the line $y = x$ followed by shear in the x-direction with $k = 2$ by pressing $A$ in 2-dimensional space, $A_1A_2$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Reflection and Shear Composition Transformation with GLM OpenGL/main.cpp**).*

**Figure 23.97:** *The uploaded image that is transformed with shear in the x-direction with $k = 2$ followed by reflection about the line $y = x$ by pressing A in 2-dimensional space, $A_2A_1$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Reflection and Shear Composition Transformation with GLM OpenGL/main.cpp**).*

## xxxviii. C++ Plot and Computation: Custom Transformation with Matrix Multiplication for 2-Dimensional Picture

The title is all about custom transformation. In this case we have a $2 \times 2$ matrix $A$

$$A = \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

If we choose the vertices for the image we uploaded to be $(0,0), (1,0), (0,1)$, and $(1,1)$ then

$$\begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

the new vertices will be $(0,0), (-1,2), (2,-1)$, and $(1,1)$.

$$\begin{aligned} (0,0) &\to (0,0) \\ (1,0) &\to (-1,2) \\ (0,1) &\to (2,-1) \\ (1,1) &\to (1,1) \end{aligned}$$

The resulting image is a parallelogram image that is reflected about the line $y = x$.

We already learn from several previous sections how to upload an image with **SOIL** and **OpenGL** and do the transformation with **GLM**' functions such as Shear, Rotation, Scaling, and

Translation. Now we want to see the transformation of that image we uploaded under multiplication by matrix $A$. What will happens? As it is not a predefined function in **GLM**, but we still going to use **GLM** either way to define the matrix $A$.

The methods are simple as follows:

1. Upload / sketch the image with defined vertices, e.g. $(0,0), (1,0), (0,1)$, and $(1,1)$.
2. Then we can multiply the vertices with matrix $A$. It is a vector-matrix multiplication.
3. The resulting vertices times matrix $A$ will be the new vertices / the transformed image vertices and we obtain the image of the transformation.

```
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include<bits/stdc++.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
#include <iostream>

using namespace std;

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
       Color = color;
       Texcoord = texcoord;
       gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
```

```
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
       outColor = mix(texture(texFreya, Texcoord), texture(texScrooge,
           Texcoord), 0.5);
}
)glsl";

int main()
{
       auto t_start = std::chrono::high_resolution_clock::now();

       sf::ContextSettings settings;
       settings.depthBits = 24;
       settings.stencilBits = 8;
       settings.majorVersion = 3;
       settings.minorVersion = 3;

       sf::Window window(sf::VideoMode(1024, 800, 32), "OpenGL", sf::Style
           ::Titlebar | sf::Style::Close, settings);

       // Initialize GLEW
       glewExperimental = GL_TRUE;
       glewInit();

       // Create Vertex Array Object
       GLuint vao;
       glGenVertexArrays(1, &vao);
       glBindVertexArray(vao);

       // Create a Vertex Buffer Object and copy the vertex data to it
       GLuint vbo;
       glGenBuffers(1, &vbo);

       GLfloat vertices[] = {
              // Position Color Texcoords
              -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left
              0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right
              0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right
              -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left
       };

       glBindBuffer(GL_ARRAY_BUFFER, vbo);
       glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
           GL_STATIC_DRAW);

       // Create an element array
```

```
GLuint ebo;
glGenBuffers(1, &ebo);

GLuint elements[] = {
       0, 1, 2,
       2, 3, 0
};

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
    GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(
    GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
```

```
        glGenTextures(2, textures);

        int width, height;
        unsigned char* image;

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textures[0]);
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample.png", &
            width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, textures[1]);
        image = SOIL_load_image("/root/SourceCodes/CPP/images/sample2.png",
            &width, &height, 0, SOIL_LOAD_RGB);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image);
        SOIL_free_image_data(image);
        glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");
        cout << "Press A to transform the image under multiplication by
            matrix A" << endl;
        cout << "Press Spacebar to return the image at the original position
            " << endl;

        bool running = true;
        GLfloat a = 1.0f;
        GLfloat b = 0.0f;
        GLfloat c = 0.0f;
        GLfloat d = 1.0f;

        GLfloat a11 = -1.0f;
        GLfloat a12 = 2.0f;
        GLfloat a21 = 2.0f;
```

```
        GLfloat a22 = -1.0f;

        int n = 2;
        float mat[n][n] = {{a11,a12}, {a21,a22}};

        cout << endl;
        cout << "Matrix A: " << endl;
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                cout << setw(6) << mat[i][j] << "\t";
                cout << endl;
        }
        cout << endl;

        while (running)
        {
                sf::Event windowEvent;
                while (window.pollEvent(windowEvent))
                {
                        switch (windowEvent.type)
                        {
                                case sf::Event::Closed:
                                running = false;
                                break;

                                case sf::Event::KeyPressed:
                                if (windowEvent.key.code == sf::Keyboard::A) {
                                        a = -1.0f;
                                        b = 2.0f;
                                        c = 2.0f;
                                        d = -1.0f;
                                        cout << "Transform the image under
                                            multiplication of A" << endl;
                                }
                                if (windowEvent.key.code == sf::Keyboard::Space
                                    ) {
                                        a = 1.0f;
                                        b = 0.0f;
                                        c = 0.0f;
                                        d = 1.0f;
                                        cout << "Normal position" << endl;
                                }
                                break;
                        }
                }

                // Clear the screen to black
```

```
                glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
                glClear(GL_COLOR_BUFFER_BIT);

                // Calculate transformation
                auto t_now = std::chrono::high_resolution_clock::now();
                float time = std::chrono::duration_cast<std::chrono::duration
                    <float>>(t_now - t_start).count();

                glm::mat4 matrixA= {{a, b, 0.0f, 0.0f},
                       {c, d, 0.0f, 0.0f},
                       {0.0f, 0.0f, 0.0f, 0.0f},
                       {0.0f, 0.0f, 0.0f, 1.0f} };

                glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(
                    matrixA));

                // Draw a rectangle from the 2 triangles using 6 indices
                glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

                // Swap buffers
                window.display();
            }

            glDeleteTextures(2, textures);

            glDeleteProgram(shaderProgram);
            glDeleteShader(fragmentShader);
            glDeleteShader(vertexShader);

            glDeleteBuffers(1, &ebo);
            glDeleteBuffers(1, &vbo);

            glDeleteVertexArrays(1, &vao);

            window.close();

            return 0;
        }
```

**C++ Code 161:** *main.cpp "Custom transformation with matrix multiplication for 2 dimensional picture"*

To compile it, type:
**g++ main.cpp -o main -lSOIL -lGLEW -lsfml-graphics -lsfml-window -lsfml-system -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We define the starting normal position as variables **GLfloat a,b,c,d**, after that we define another variables to be shown as matrix $A$ at the beginning of the program **GLfloat a11,a12,a21,a22**.

```
bool running = true;
GLfloat a = 1.0f;
GLfloat b = 0.0f;
GLfloat c = 0.0f;
GLfloat d = 1.0f;

GLfloat a11 = -1.0f;
GLfloat a12 = 2.0f;
GLfloat a21 = 2.0f;
GLfloat a22 = -1.0f;

int n = 2;
float mat[n][n] = {{a11,a12}, {a21,a22}};

cout << endl;
cout << "Matrix A: " << endl;
for (int i = 0; i < n; i++)
{
	for (int j = 0; j < n; j++)
	cout << setw(6) << mat[i][j] << "\t";
	cout << endl;
}
cout << endl;
```

- Under the **while (running)** loop we redefine the variables **GLfloat a,b,c,d** under the keypress events **A** and **Spacebar**, when we press **A** it will show the transformed image under multiplication of matrix $A$. Pressing **Spacebar** will make the image to return to normal / original position.

  Take a look at **glm::mat4 matrixA** declaration, it is a $4 \times 4$ matrix, even if this section concerns about 2-dimensional image / picture, we still use $4 \times 4$ transformation matrix to transform the image. This is to represent 3-dimensional transformation with the last column represent translation for each axis in 3-dimension.

```
while (running)
{
	sf::Event windowEvent;
	while (window.pollEvent(windowEvent))
	{
		switch (windowEvent.type)
		{
			case sf::Event::Closed:
			running = false;
			break;
```

```
                        case sf::Event::KeyPressed:
                        if (windowEvent.key.code == sf::Keyboard::A
                            ) {
                               a = -1.0f;
                               b = 2.0f;
                               c = 2.0f;
                               d = -1.0f;
                               cout << "Transform the image under
                                   multiplication of A" << endl;
                        }
                        if (windowEvent.key.code == sf::Keyboard::
                            Space) {
                               a = 1.0f;
                               b = 0.0f;
                               c = 0.0f;
                               d = 1.0f;
                               cout << "Normal position" << endl;
                        }
                        break;
                }
        }

        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        ...

        glm::mat4 matrixA= {{a, b, 0.0f, 0.0f},
                {c, d, 0.0f, 0.0f},
                {0.0f, 0.0f, 0.0f, 0.0f},
                {0.0f, 0.0f, 0.0f, 1.0f} };

        glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr
            (matrixA));
```

We are going to refresh our memories, it can be reread on Chapter 3, Linear Algebra part. This is a scaling matrix on any vector $(x, y, z)$

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{bmatrix}$$

This is a translation matrix on any vector $(x, y, z)$ with translation vector as $(T_x, T_y, T_z)$

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} T_x + x \\ T_y + y \\ T_z + z \\ 1 \end{bmatrix}$$

so in this case of 2-dimensional picture if we have matrix $A$ like this

$$A = \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

then we will write the matrix $4 \times 4$ in the C++ code under **GLM'** definition of **mat4** as

$$\begin{bmatrix} -1 & 2 & 0 & 0 \\ 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

of course we do not explicitly wrote the number in floating terms, but we write variables instead, so the image will be transformed under keypress event. Notice we let the third row all zeroes, because we want to do 2-dimensional picture image transformation, so we are not doing any computation in the third axis, that will be concern for the next section where we will be working on 3-dimensional object. That is when we are able to comprehend more to this real-life world' science and engineering, help our problem solving, thus able to analyze and create more innovations.

**Figure 23.98:** *The uploaded image that is transformed with custom transformation when multiplied by matrix A by pressing A in 2-dimensional space. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Picture Custom Transformation by Matrix Multiplication with GLM OpenGL/main.cpp**).*

## xxxix. C++ Computation: Image of a Line

The invertible matrix

$$\begin{bmatrix} 3 & 1 \\ 2 & 1 \end{bmatrix}$$

maps the line $y = 2x + 1$ into another line. Find its equation.

**Solution:**

Let $(x, y)$ be a point on the line $y = 2x + 1$, and let $(x_m, y_m)$ be its image under multiplication by $A$. Then

$$\begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

thus

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -2 & 3 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= x_m - y_m \\ y &= -2x_m + 3y_m \end{aligned}$$

Substituting in $y = 2x + 1$ yields

$$\begin{aligned} -2x_m + 3y_m &= 2(x_m - y_m) + 1 \\ y_m &= \frac{4}{5}x_m + \frac{1}{5} \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = \frac{4}{5}x + \frac{1}{5}$$

for the equation of the line.

The C++ code is depending on **SymbolicC++** to compute the inverse of matrix $A$ and then do the symbolic computation. While the plotting is done with **gnuplot**. From computation to plotting the lines, we still do manual way here.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
#include "gnuplot−iostream.h"

using namespace std;

#define R 2 // number of rows
#define C 2 // number of columns

// Driver program
int main()
```

```
{
        Symbolic w1("w1");
        Symbolic w2("w2");
        Symbolic x("x");
        Symbolic y("y");
        Symbolic xm("xm");
        Symbolic ym("ym");

        Matrix<Symbolic> X(R,1);
        Matrix<Symbolic> W(R,1);
        W[0][0] = xm;
        W[1][0] = ym;

        // Construct a symbolic matrix of size R X C
        Matrix<Symbolic> A_mat(R,C);
        A_mat[0][0] = 3;  A_mat[0][1] = 1;
        A_mat[1][0] = 2;  A_mat[1][1] = 1;

        cout << "The line equation:\n" << "y = 2x+1" <<endl;
        cout << endl;
        cout << "A:\n" << A_mat <<endl;
        cout << endl;
        Matrix<Symbolic> AI(R,C);
        AI = A_mat.inverse();

        cout << "A^{-1}:\n" << AI <<endl;
        cout << endl;

        cout << "A^{-1} * [xm ym] :\n" << AI*W <<endl;

        X = (AI*W);
        // X[0] = x, X[1] = y
        cout << "y = 2x + 1 : \n y - 2x - 1 = 0 \n 0 =" << X[1] - Symbolic
            (2)*X[0] - Symbolic(1) <<endl;

        Gnuplot gp;
        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        // Put the degree in the equation for f(x) = (theta*DEGTORAD )* x
        gp << "f(x) =2*x + 1 \n";
        gp << "g(x) =(0.8)*x + (0.2) \n";
        gp << "plot f(x) title 'f(x) = original line' lc 'green', g(x) title
             'g(x) = image of line f(x)' dashtype 3\n";

        return 0;
```

```
}
```

**C++ Code 162:** *main.cpp "Image of a Line"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.99:** *The original line $f(x)$ is the line $y = 2x + 1$, after we maps the line by multiplying it with matrix $A$, we will obtain the image as line $g(x)$ ($y_m = \frac{4}{5}x_m + \frac{1}{5}$). (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Image of a Line Finding Inverse with SymbolicC++/main.cpp**).*

## xl. C++ Plot and Computation: Equation of the image of a line under Transformation in $\mathbb{R}^2$

Find an equation of the image of the line $y = 2x$ in $\mathbb{R}^2$ under

(a) A shear of factor 3 in the $x$-direction.

(b) A compression of factor $\frac{1}{2}$ in the $y$-direction.

(c) A reflection about $y = x$.

(d) A reflection about the $y$-axis.

(e) A rotation fo $60^0$ about the origin.

**Solution:**

(a) Matrix

$$A = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$$

maps the line $y = 2x$ into another line with transformation of a shear of factor 3 in the $x$-direction.

Let $(x, y)$ be a point on the line $y = 2x$, and let $(x_m, y_m)$ be its image under multiplication by $A$. Then

$$\begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

thus

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= x_m - 3y_m \\ y &= y_m \end{aligned}$$

Substituting in $y = 2x$ yields

$$\begin{aligned} y - 2x &= 0 \\ y_m - 2(x_m - 3y_m) &= 0 \\ y_m &= \frac{2}{7}x_m \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = \frac{2}{7}x$$

for the equation of the line.

(b) Matrix

$$B = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

maps the line $y = 2x$ into another line with transformation of a compression of factor $\frac{1}{2}$ in the $y$-direction.

Let $(x, y)$ be a point on the line $y = 2x$, and let $(x_m, y_m)$ be its image under multiplication by $B$. Then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= x_m \\ y &= 2y_m \end{aligned}$$

Substituting in $y = 2x$ yields

$$\begin{aligned} y - 2x &= 0 \\ 2y_m - 2(x_m) &= 0 \\ y_m &= x_m \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = x$$

(c) Matrix

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

maps the line $y = 2x$ into another line with transformation of a reflection about $y = x$.

Let $(x, y)$ be a point on the line $y = 2x$, and let $(x_m, y_m)$ be its image under multiplication by $C$. Then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= y_m \\ y &= x_m \end{aligned}$$

Substituting in $y = 2x$ yields

$$\begin{aligned} y - 2x &= 0 \\ x_m - 2(y_m) &= 0 \\ y_m &= \frac{1}{2}x_m \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = \frac{1}{2}x$$

(d) Matrix

$$D = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

maps the line $y = 2x$ into another line with transformation of a reflection about the $y$-axis.

Let $(x, y)$ be a point on the line $y = 2x$, and let $(x_m, y_m)$ be its image under multiplication by $D$. Then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= -x_m \\ y &= y_m \end{aligned}$$

Substituting in $y = 2x$ yields

$$\begin{aligned} y - 2x &= 0 \\ y_m - 2(-x_m) &= 0 \\ y_m &= -2x_m \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = -2x$$

(e) Matrix

$$E = \begin{bmatrix} \cos(60) & -\sin(60) \\ \sin(60) & \cos(60) \end{bmatrix}$$

maps the line $y = 2x$ into another line with transformation of a rotation fo $60^0$ about the origin.

Let $(x, y)$ be a point on the line $y = 2x$, and let $(x_m, y_m)$ be its image under multiplication by $E$. Then

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(60) & -\sin(60) \\ \sin(60) & \cos(60) \end{bmatrix}^{-1} \begin{bmatrix} x_m \\ y_m \end{bmatrix} = \begin{bmatrix} \cos(60) & \sin(60) \\ -\sin(60) & \cos(60) \end{bmatrix} \begin{bmatrix} x_m \\ y_m \end{bmatrix}$$

so

$$\begin{aligned} x &= 0.5x_m + 0.866025y_m \\ y &= -0.866025x_m + 0.5y_m \end{aligned}$$

Substituting in $y = 2x$ yields

$$\begin{aligned} y - 2x &= 0 \\ -0.866025x_m + 0.5y_m - 2(0.5x_m + 0.866025y_m) &= 0 \\ y_m &= -\frac{1.86603}{1.23205}x_m \end{aligned}$$

Thus $(x_m, y_m)$ satisfies

$$y = -\frac{1.86603}{1.23205}x$$

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
#include "gnuplot−iostream.h"

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

using namespace std;

#define R 2 // number of rows
#define C 2 // number of columns
#define theta 60

// Driver program
int main()
{
       Symbolic w1("w1");
       Symbolic w2("w2");
       Symbolic x("x");
       Symbolic y("y");
       Symbolic xm("xm");
       Symbolic ym("ym");

       Matrix<Symbolic> X(R,1);
       Matrix<Symbolic> XB(R,1);
       Matrix<Symbolic> XC(R,1);
       Matrix<Symbolic> XD(R,1);
       Matrix<Symbolic> XE(R,1);
       Matrix<Symbolic> W(R,1);
       W[0][0] = xm;
       W[1][0] = ym;

       // Construct a symbolic matrix of size R X C
       Matrix<Symbolic> A_mat(R,C);
       A_mat[0][0] = 1;  A_mat[0][1] = 3;
       A_mat[1][0] = 0;  A_mat[1][1] = 1;

       Matrix<Symbolic> B_mat(R,C);
       B_mat[0][0] = 1;  B_mat[0][1] = 0;
       B_mat[1][0] = 0;  B_mat[1][1] = 0.5;

       Matrix<Symbolic> C_mat(R,C);
       C_mat[0][0] = 0;  C_mat[0][1] = 1;
       C_mat[1][0] = 1;  C_mat[1][1] = 0;
```

```
Matrix<Symbolic> D_mat(R,C);
D_mat[0][0] = -1;  D_mat[0][1] = 0;
D_mat[1][0] = 0;  D_mat[1][1] = 1;

Matrix<Symbolic> E_mat(R,C);
E_mat[0][0] = cos(theta*DEGTORAD);  E_mat[0][1] = - sin(theta*
    DEGTORAD);
E_mat[1][0] = sin(theta*DEGTORAD);  E_mat[1][1] = cos(theta*DEGTORAD
    );

cout << "The line equation:\n" << "y = 2x" <<endl;
cout << endl;
cout << "
    ******************************************************************
    " <<endl;
cout << "Shear with factor 3 in the x-direction" <<endl;
cout << "
    ******************************************************************
    " <<endl;
cout << "A:\n" << A_mat <<endl;
cout << endl;
Matrix<Symbolic> AI(R,C);
AI = A_mat.inverse();

cout << "A^{-1}:\n" << AI <<endl;
cout << endl;

cout << "A^{-1} * [xm ym] :\n" << AI*W <<endl;

X = (AI*W);
// X[0] = x, X[1] = y
cout << "y = 2x : \n y - 2x = 0 \n 0 =" << X[1] - Symbolic(2)*X[0]
    <<endl;

cout << "
    ******************************************************************
    " <<endl;
cout << "Compression with factor 0.5 in the y-direction" <<endl;
cout << "
    ******************************************************************
    " <<endl;

cout << "B:\n" << B_mat <<endl;
cout << endl;
Matrix<Symbolic> BI(R,C);
BI = B_mat.inverse();

cout << "B^{-1}:\n" << BI <<endl;
```

```
cout << endl;

cout << "B^{-1} * [xm ym] :\n" << BI*W <<endl;

XB = (BI*W);
// X[0] = x, X[1] = y
cout << "y = 2x : \n y - 2x = 0 \n 0 =" << XB[1] - Symbolic(2)*XB
    [0] <<endl;

cout << "
    ********************************************************************
    " <<endl;
cout << "Reflection about the line y=x" <<endl;
cout << "
    ********************************************************************
    " <<endl;

cout << "C:\n" << C_mat <<endl;
cout << endl;
Matrix<Symbolic> CI(R,C);
CI = C_mat.inverse();

cout << "C^{-1}:\n" << CI <<endl;
cout << endl;

cout << "C^{-1} * [xm ym] :\n" << CI*W <<endl;

XC = (CI*W);
// X[0] = x, X[1] = y
cout << "y = 2x : \n y - 2x = 0 \n 0 =" << XC[1] - Symbolic(2)*XC
    [0] <<endl;

cout << "
    ********************************************************************
    " <<endl;
cout << "Reflection about the y-axis" <<endl;
cout << "
    ********************************************************************
    " <<endl;

cout << "D:\n" << D_mat <<endl;
cout << endl;
Matrix<Symbolic> DI(R,C);
DI = D_mat.inverse();

cout << "D^{-1}:\n" << DI <<endl;
cout << endl;
```

```
cout << "D^{-1} * [xm ym] :\n" << DI*W <<endl;

XD = (DI*W);
// X[0] = x, X[1] = y
cout << "y = 2x : \n y - 2x = 0 \n 0 =" << XD[1] - Symbolic(2)*XD
    [0] <<endl;

cout << "
    ******************************************************************
    " <<endl;
cout << "Counterclocwise rotation through an angle 60 degree" <<endl
    ;
cout << "
    ******************************************************************
    " <<endl;

cout << "E:\n" << E_mat <<endl;
cout << endl;
Matrix<Symbolic> EI(R,C);
EI = E_mat.inverse();

cout << "E^{-1}:\n" << EI <<endl;
cout << endl;

cout << "E^{-1} * [xm ym] :\n" << EI*W <<endl;

XE = (EI*W);
// X[0] = x, X[1] = y
cout << "y = 2x : \n y - 2x = 0 \n 0 =" << XE[1] - Symbolic(2)*XE
    [0] <<endl;

Gnuplot gp;
// Don't forget to put "\n" at the end of each line!
gp << "set xrange [-1.5:2]\nset yrange [-1.5:2]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
// Put the degree in the equation for f(x) = (theta*DEGTORAD )* x
gp << "f(x) = 2*x \n";
gp << "g(x) = (0.2857)*x \n";
gp << "h(x) = x \n";
gp << "j(x) = 0.5*x \n";
gp << "k(x) = -2*x \n";
gp << "l(x) = -1.514573*x \n";

gp << "plot f(x) title 'original line' lc 'green', g(x) title 'with
    shear of factor 3 in the x-direction' dashtype 3, h(x) title '
    with compression of factor 1/2 in the y-direction' dashtype 6,
```

```
                j(x) title 'with reflection about the line y=x' lc 'red', k(x)
                title 'with reflection about the y-axis' lc 'blue', l(x) title
                'with counterclockwise rotation of 60 degree' lc 'orange'\n";

            return 0;
    }
```

**C++ Code 163:** *main.cpp "The images of a line under various transformations"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Here, we still put the equation of line for the images of line $y = 2x$ under various transformations with manual input. The first section is concerning on computing the equation of the image of line $y = 2x$ with **SymbolicC++**

```
            gp << "f(x) = 2*x \n";
            gp << "g(x) = (0.2857)*x \n";
            gp << "h(x) = x \n";
            gp << "j(x) = 0.5*x \n";
            gp << "k(x) = -2*x \n";
            gp << "l(x) = -1.514573*x \n";
```

```
Counterclockwise rotation through an angle 60 degree
*********************************************************************
E:
[   0.5    -0.866025]
[ 0.866025    0.5   ]


E^{-1}:
[   0.5     0.866025]
[-0.866025    0.5   ]


E^{-1} * [xm ym] :
[ 0.5*xm+0.866025*ym]
[-0.866025*xm+0.5*ym]

y = 2x :
 y - 2x = 0
 0 =[-1.86603*xm-1.23205*ym]
```

**Figure 23.100:** *The original line $y = 2x$ is transformed under various transformations.* (***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Image of a Line Under Transformations with SymbolicC++/main.cpp***).

## xli. C++ Plot and Computation: Shear Transformation for 3-Dimensional Cube

In $\mathbb{R}^3$ the shear in the $xy$-direction with factor $k$ is the matrix transformation that moves each point $(x, y, z)$ parallel to the $xy$-plane to the new position $(x + kz, y + kz, z)$.

(a) Find the standard matrix for the shear in the $xy$-direction with factor $k$.

(b) How would you define the shear in the $xz$-direction with factor $k$ and the shear in the $yz$-direction with factor $k$? Find the standard matrices for this matrix transformations.

**Solution:**

(a) A matrix operator of the form $T(x, y, z) = (x + kz, y + kz, z)$ translates a point $(x, y, z)$ in the 3-dimensional space parallel to the $xy$-plane by an amount $kz$ that is proportional to the $z$ coordinate of the point. The value of $z$ coordinate never change in this shear transformation.

The standard matrix for the shear in the $xy$-direction with factor $k$ is

$$T = \begin{bmatrix} 1 & 0 & k \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix}$$

for the computer graphics when using **GLM** the standard matrix will be 4-dimensional like this

$$T_{GLM} = \begin{bmatrix} 1 & 0 & k & 0 \\ 0 & 1 & k & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b) A matrix operator of the form $T(x, y, z) = (x + ky, y, z + ky)$ translates a point $(x, y, z)$ in the 3-dimensional space parallel to the $xz$-plane by an amount $ky$ that is proportional to the $y$ coordinate of the point. The value of $y$ coordinate never change in this shear transformation.

The standard matrix for the shear in the $xz$-direction with factor $k$ is

$$T = \begin{bmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & k & 1 \end{bmatrix}$$

for the computer graphics when using **GLM** the standard matrix will be 4-dimensional like this

$$T_{GLM} = \begin{bmatrix} 1 & k & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & k & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A matrix operator of the form $T(x, y, z) = (x, y + kx, z + kx)$ translates a point $(x, y, z)$ in the 3-dimensional space parallel to the $yz$-plane by an amount $ky$ that is proportional to the $x$ coordinate of the point. The value of $x$ coordinate never change in this shear transformation.

The standard matrix for the shear in the $yz$-direction with factor $k$ is

$$T = \begin{bmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ k & 0 & 1 \end{bmatrix}$$

for the computer graphics when using **GLM** the standard matrix will be 4-dimensional like this

$$T_{GLM} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ k & 1 & 0 & 0 \\ k & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We create the C++ code with **GLAD, GLFW, GLM,** and **SOIL** to upload the texture on the cube. The nice thing is we can move the camera and see the transformed cube in another point of view / perspective. This C++ code is a simple modification from reading the book of Learn OpenGL [16].

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <SOIL/SOIL.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtx/transform2.hpp> // For ShearY3D and ShearX3D

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// camera
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

bool firstMouse = true;
float yaw = -90.0f; // yaw is initialized to -90.0 degrees since a yaw of
     0.0 results in a direction vector pointing to the right so we
    initially rotate a bit to the left.
```

```
float pitch = 0.0f;
float lastX = 800.0f / 2.0;
float lastY = 600.0 / 2.0;
float fov = 45.0f;

// timing
float deltaTime = 0.0f; // time between current frame and last frame
float lastFrame = 0.0f;

int main()
{
	// glfw: initialize and configure
	// ------------------
	glfwInit();
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	#ifdef __APPLE__
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	#endif

	// glfw window creation
	// --------------------
	GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "
		LearnOpenGL", NULL, NULL);
	if (window == NULL)
	{
		std::cout << "Failed to create GLFW window" << std::endl;
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);
	glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
	glfwSetCursorPosCallback(window, mouse_callback);
	glfwSetScrollCallback(window, scroll_callback);

	// tell GLFW to capture our mouse
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

	// glad: load all OpenGL function pointers
	// ---------------------------------------
	if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
	{
		std::cout << "Failed to initialize GLAD" << std::endl;
		return -1;
	}
```

```
        // configure global opengl state
        // ————————————————
        glEnable(GL_DEPTH_TEST);

        // build and compile our shader zprogram
        // ————————————————
        Shader ourShader("camera.vs", "camera.fs");

        // set up vertex data (and buffer(s)) and configure vertex
            attributes
        // ————————————————
        float vertices[] = {
               -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
               0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
               0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
               0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
               -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
               -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

               -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
               0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
               0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
               0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
               -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,
               -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,

               -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
               -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
               -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
               -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
               -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
               -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

               0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
               0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
               0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
               0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
               0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
               0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

               -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
               0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
               0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
               0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
               -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
               -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,

               -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
```

```
        0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
        0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
        0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
        -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,
        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f
};
// world space positions of our cubes
glm::vec3 cubePositions[] = {
        glm::vec3( 0.0f, 0.0f, 0.0f),
        glm::vec3( 2.0f, 5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3( 2.4f, -0.4f, -3.5f),
        glm::vec3(-1.7f, 3.0f, -7.5f),
        glm::vec3( 1.3f, -2.0f, -2.5f),
        glm::vec3( 1.5f, 2.0f, -2.5f),
        glm::vec3( 1.5f, 0.2f, -1.5f),
        glm::vec3(-1.3f, 1.0f, -1.5f)
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (
    void*)0);
glEnableVertexAttribArray(0);
// texture coord attribute
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (
    void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);


// load and create a texture
unsigned int texture1, texture2;
// texture 1
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D, texture1);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// set texture filtering parameters
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load image, create texture and generate mipmaps
int width, height, nrChannels;
unsigned char* image = SOIL_load_image("/root/SourceCodes/CPP/images
    /sample.png", &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);

if (image)
{
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
           GL_RGB, GL_UNSIGNED_BYTE, image);
       glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
       std::cout << "Failed to load texture" << std::endl;
}
SOIL_free_image_data(image);
// texture 2
glGenTextures(1, &texture2);
glBindTexture(GL_TEXTURE_2D, texture2);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load image, create texture and generate mipmaps
unsigned char* image2 = SOIL_load_image("/root/SourceCodes/CPP/
    images/sample.png", &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
    GL_UNSIGNED_BYTE, image);

if (image2)
{
       // note that the awesomeface.png has transparency and thus an
            alpha channel, so make sure to tell OpenGL the data type
            is of GL_RGBA
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
           GL_RGBA, GL_UNSIGNED_BYTE, image2);
       glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
       std::cout << "Failed to load texture" << std::endl;
}
```

```
        SOIL_free_image_data(image2);

        // tell opengl for each sampler to which texture unit it belongs to
            (only has to be done once)
        // ----------------
        ourShader.use();
        ourShader.setInt("texture1", 0);
        ourShader.setInt("texture2", 1);


        // render loop
        // -------------
        while (!glfwWindowShouldClose(window))
        {
            // per-frame time logic
            // -----------------
            float currentFrame = static_cast<float>(glfwGetTime());
            deltaTime = currentFrame - lastFrame;
            lastFrame = currentFrame;

            // input
            // -----
            processInput(window);

            // render
            // ------
            glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            // bind textures on corresponding texture units
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, texture1);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, texture2);

            // activate shader
            ourShader.use();

            // pass projection matrix to shader (note that in this case
                it could change every frame)
            glm::mat4 projection = glm::perspective(glm::radians(fov), (
                float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
            ourShader.setMat4("projection", projection);

            // camera/view transformation
            glm::mat4 view = glm::lookAt(cameraPos, cameraPos +
                cameraFront, cameraUp);
            ourShader.setMat4("view", view);
```

```
                // render a cube / box
                glBindVertexArray(VAO);
                int i = 0;
                // calculate the model matrix for each object and pass it to
                    shader before drawing
                glm::mat4 model = glm::mat4(1.0f); // make sure to initialize
                     matrix to identity matrix first
                model = glm::translate(model, cubePositions[i]);
                float angle = 20.0f * i;
                GLfloat k = 1.0f;
                // change to shearX3D -> Shear in the y-direction
                // change to shearY3D -> Shear in the x-direction
                model = glm::shearY3D(model,
                k, // the factor k
                0.0f); // z = 0 so it will looks like we are in 2-
                    dimensional space
                ourShader.setMat4("model", model);

                glDrawArrays(GL_TRIANGLES, 0, 36);

                // glfw: swap buffers and poll IO events (keys pressed/
                    released, mouse moved etc.)
                // -------------------------------------------------------
                glfwSwapBuffers(window);
                glfwPollEvents();
        }

        // optional: de-allocate all resources once they've outlived their
            purpose:
        // -------------------------------------------------------
        glDeleteVertexArrays(1, &VAO);
        glDeleteBuffers(1, &VBO);

        // glfw: terminate, clearing all previously allocated GLFW resources
            .
        // -------------------------------------------------------
        glfwTerminate();
        return 0;
}

// process all input: query GLFW whether relevant keys are pressed/released
     this frame and react accordingly
// -------------------------------------------------------
void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
```

```
        float cameraSpeed = static_cast<float>(2.5 * deltaTime);
        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) *
            cameraSpeed;
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) *
            cameraSpeed;
}

// glfw: whenever the window size changed (by OS or user resize) this
    callback function executes
// ---------------
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
        // make sure the viewport matches the new window dimensions; note
            that width and
        // height will be significantly larger than specified on retina
            displays.
        glViewport(0, 0, width, height);
}

// glfw: whenever the mouse moves, this callback is called
// ---------------
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);

        if (firstMouse)
        {
                lastX = xpos;
                lastY = ypos;
                firstMouse = false;
        }

        float xoffset = xpos - lastX;
        float yoffset = lastY - ypos; // reversed since y-coordinates go
            from bottom to top
        lastX = xpos;
        lastY = ypos;

        float sensitivity = 0.1f; // change this value to your liking
        xoffset *= sensitivity;
```

```
        yoffset *= sensitivity;

        yaw += xoffset;
        pitch += yoffset;

        // make sure that when pitch is out of bounds, screen doesn't get
            flipped
        if (pitch > 89.0f)
        pitch = 89.0f;
        if (pitch < -89.0f)
        pitch = -89.0f;

        glm::vec3 front;
        front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
        front.y = sin(glm::radians(pitch));
        front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
        cameraFront = glm::normalize(front);
    }

    // glfw: whenever the mouse scroll wheel scrolls, this callback is called
    // ----------------
    void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
    {
        fov -= (float)yoffset;
        if (fov < 1.0f)
        fov = 1.0f;
        if (fov > 45.0f)
        fov = 45.0f;
    }
```

**C++ Code 164:** *main.cpp "Shear transformation for 3 dimensional cube with camera moving"*

To compile it, type:
**// g++ main.cpp -o main /root/SourceCodes/CPP/src/glad.c -lSOIL -lglfw -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Inside the **int main()** we create / render a cube that is already transformed with shear in the $xy$-direction, with $k = 1$.

```
        // render a cube / box
        glBindVertexArray(VAO);
        int i = 0;
        // calculate the model matrix for each object and pass it to
            shader before drawing
```

```
            glm::mat4 model = glm::mat4(1.0f); // make sure to initialize
                matrix to identity matrix first
            model = glm::translate(model, cubePositions[i]);
            float angle = 20.0f * i;
            GLfloat k = 1.0f;
            // change to shearX3D —> Shear in the y—direction
            // change to shearY3D —> Shear in the x—direction
            model = glm::shearY3D(model,
            k, // the factor k
            0.0f); // z = 0 so it will looks like we are in 2—dimensional
                space
            ourShader.setMat4("model", model);

            glDrawArrays(GL_TRIANGLES, 0, 36);
```

**Figure 23.101:** *The 3-dimensional cube is transformed with shear transformations in the xy-direction with factor k. You can move the camera by pressing W,A,S,D and mouse' control. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Cube Matrix Transformation with Moving Camera/main.cpp**).*

**Figure 23.102:** *Another C++ code to show the 3-dimensional cube that is transformed with shear transformations and rotating automatically toward positive y-axis. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Cube with Shear Transformation Rotating/-main.cpp**).*

## xlii. C++ Plot: Load 3-Dimensional Object with Assimp and OpenGL

This section shows how to load a 3-dimensional (3D) object that can be made with computer graphics program like Blender, it is very useful, since we can apply what we have learned from Matrix Transformation to this 3-dimensional object. For example if we load an object as an airplane, we can see how it turn around in three Euler angles: pitch, yaw, and roll. We can apply the texture and we can scale the 3D object as well.

The C++ code depends on the great library **Assimp** to import the 3-dimensional object, that is a file with extension **.obj**, you can get tons of file with extension like this online or create one yourself.

We upload the texture to wrap the 3D object with **STB Image** library, it is a very popular image loading library, it is onle one header file that can be downloaded from:
**github.com/nothings/stb/blob/master/stb_image.h**

The texture is using file of extension **.jpg**, and we also rely with OpenGL and GLFW for the camera movement and user input handling. The basic knowledge is from chapter 19-21 in [16], it is a very great book and source of learning to anyone interested in Computer Graphics which rely heavily on Linear Algebra and Geometry, especially for the future development of Machine Learning and Artificial Intelligence.

What you need to prepare in the working directory:

1. main.cpp
2. Makefile (optional)
3. model_loading.fs (GLSL file to define fragment shader)
4. model_loading.vs (GLSL file to define vertex shader)
5. The 3-dimensional object file with extension **.obj**
6. The texture image file with extension **.jpg**

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#define STB_IMAGE_IMPLEMENTATION
//#include "stb_image.h" // we do not use this since ' #include "stb_image.h" '
    already included in /usr/include/learnopengl/model.h

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>
#include <learnopengl/camera.h>
#include <learnopengl/model.h>
```

```
#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 1200;
const unsigned int SCR_HEIGHT = 800;

// camera
Camera camera(glm::vec3(0.0f, 0.0f, 30.0f));
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;

// timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;

int main()
{
	// glfw: initialize and configure
	// ------------------------------
	glfwInit();
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	#ifdef __APPLE__
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	#endif

	// glfw window creation
	// --------------------
	GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "
	    LearnOpenGL", NULL, NULL);
	if (window == NULL)
	{
		std::cout << "Failed to create GLFW window" << std::endl;
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);
	glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
	glfwSetCursorPosCallback(window, mouse_callback);
	glfwSetScrollCallback(window, scroll_callback);
```

```
        // tell GLFW to capture our mouse
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

        // glad: load all OpenGL function pointers
        // ————————————
        if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
        {
               std::cout << "Failed to initialize GLAD" << std::endl;
               return -1;
        }

        // tell stb_image.h to flip loaded texture's on the y-axis (before
            loading model).
        stbi_set_flip_vertically_on_load(true);

        // configure global opengl state
        // ————————————
        glEnable(GL_DEPTH_TEST);

        // build and compile shaders
        // ————————————
        Shader ourShader("model_loading.vs", "model_loading.fs");

        // load models
        // ————————————
        // The default filesystem is the first place we build LearnOpenGL at
             /sources/Hamzstlab/LearnOpenGL
        //Model ourModel(FileSystem::getPath("backpack.obj"));
        Model ourModel("spider.obj");

        // load and create a texture
        // ————————————
        unsigned int texture;
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture); // all upcoming GL_TEXTURE_2D
             operations now have effect on this texture object
        // set the texture wrapping parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // set
             texture wrapping to GL_REPEAT (default wrapping method)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        // set texture filtering parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // load image, create texture and generate mipmaps
        int width, height, nrChannels;
        unsigned char *data = stbi_load("/root/SourceCodes/CPP/Assets/
```

```
            Textures/matrix.jpg", &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
                GL_RGB, GL_UNSIGNED_BYTE, data);
            glGenerateMipmap(GL_TEXTURE_2D);
        }
        else
        {
            std::cout << "Failed to load texture" << std::endl;
        }
        stbi_image_free(data);


        // draw in wireframe
        //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        // render loop
        // -----------
        while (!glfwWindowShouldClose(window))
        {
            // per-frame time logic
            // --------------
            float currentFrame = static_cast<float>(glfwGetTime());
            deltaTime = currentFrame - lastFrame;
            lastFrame = currentFrame;

            // input
            // -----
            processInput(window);

            // render
            // ------
            glClearColor(1.0f, 1.0f, 0.05f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            // bind Texture
            glBindTexture(GL_TEXTURE_2D, texture);

            // don't forget to enable shader before setting uniforms
            ourShader.use();

            // view/projection transformations
            glm::mat4 projection = glm::perspective(glm::radians(camera.
                Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f
                );
            glm::mat4 view = camera.GetViewMatrix();
            ourShader.setMat4("projection", projection);
```

```
                ourShader.setMat4("view", view);

                // render the loaded model
                glm::mat4 model = glm::mat4(1.0f);
                model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
                    // translate it down so it's at the center of the scene
                model = glm::scale(model, glm::vec3(0.23f, 0.23f, 0.23f)); //
                     if the model is too big for our scene, scale it down
                    here
                ourShader.setMat4("model", model);
                ourModel.Draw(ourShader);

                // glfw: swap buffers and poll IO events (keys pressed/
                    released, mouse moved etc.)
                // -------------
                glfwSwapBuffers(window);
                glfwPollEvents();
        }

        // glfw: terminate, clearing all previously allocated GLFW resources
            .
        // -------------
        glfwTerminate();
        return 0;
}

// process all input: query GLFW whether relevant keys are pressed/released
     this frame and react accordingly
// -------------
void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}

// glfw: whenever the window size changed (by OS or user resize) this
    callback function executes
// -------------
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
```

```
{
        // make sure the viewport matches the new window dimensions; note
            that width and
        // height will be significantly larger than specified on retina
            displays.
        glViewport(0, 0, width, height);
}

// glfw: whenever the mouse moves, this callback is called
// -------------
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);

        if (firstMouse)
        {
                lastX = xpos;
                lastY = ypos;
                firstMouse = false;
        }

        float xoffset = xpos - lastX;
        float yoffset = lastY - ypos; // reversed since y-coordinates go
            from bottom to top

        lastX = xpos;
        lastY = ypos;

        camera.ProcessMouseMovement(xoffset, yoffset);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback is called
// -------------
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
        camera.ProcessMouseScroll(static_cast<float>(yoffset));
}
```

**C++ Code 165:** *main.cpp "3D Object Load with assimp and opengl"*

To compile it, type:
**g++ -o main main.cpp /root/SourceCodes/CPP/src/glad.c -lstdc++ -lassimp -lglfw -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- After we initialize GLAD, we build and compile the shaders (vertex shader and fragment shader), afterwards we load the model **spider.obj** that is located in the current working directory / the same folder as the **main.cpp**, the include file **model.h** that is located in **/usr/include/learnopengl** contain all the codes needed for **Assimp** to import the 3D object file . That is why we have **#include <learnopengl/model.h>** at the beginning of the **main.cpp** code, because all codes to do the loading of the 3D model via **Assimp** are coded inside the **model.h**.

```
// tell stb_image.h to flip loaded texture's on the y-axis (
    before loading model).
stbi_set_flip_vertically_on_load(true);

// configure global opengl state
// -----------------------
glEnable(GL_DEPTH_TEST);

// build and compile shaders
// ------------------------
Shader ourShader("model_loading.vs", "model_loading.fs");

// load models
// -----------
Model ourModel("spider.obj");
```

- This is the part to load texture from **.jpg** image file that is located in **/root/SourceCodes/CPP/Assets/Textures/**. It is very recommended if you read the book on Learning OpenGL [16] to know the details of texture and how to modify it.

```
 // load and create a texture
// ---------------------------
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture); // all upcoming
    GL_TEXTURE_2D operations now have effect on this texture
    object
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    // set texture wrapping to GL_REPEAT (default wrapping
    method)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR
    );
// load image, create texture and generate mipmaps
int width, height, nrChannels;
```

/opt/hamzstlib/LearnOpenGL/includes/learnopengl/model.h - Mousepad

File Edit Search View Document Help

Warning: you are using the root account. You may harm your system.

model.h main.cpp

```
1  #ifndef MODEL_H
2  #define MODEL_H
3
4  #include <glad/glad.h>
5
6  #include <glm/glm.hpp>
7  #include <glm/gtc/matrix_transform.hpp>
8  #include <stb_image.h>
9  #include <assimp/Importer.hpp>
10 #include <assimp/scene.h>
11 #include <assimp/postprocess.h>
12
13 #include <learnopengl/mesh.h>
14 #include <learnopengl/shader.h>
15
16 #include <string>
17 #include <fstream>
18 #include <sstream>
19 #include <iostream>
20 #include <map>
21 #include <vector>
22 using namespace std;
23
24 unsigned int TextureFromFile(const char *path, const string &directory, bool gamma = false);
25
26 class Model
27 {
28 public:
29    // model data
30    vector<Texture> textures_loaded;    // stores all the textures loaded so far, optimization to make sure textures aren't loaded more than once.
31    vector<Mesh>    meshes;
32    string directory;
33    bool gammaCorrection;
34
35    // constructor, expects a filepath to a 3D model.
36    Model(string const &path, bool gamma = false) : gammaCorrection(gamma)
37    {
38       loadModel(path);
39    }
40
41    // draws the model, and thus all its meshes
42    void Draw(Shader &shader)
43    {
44       for(unsigned int i = 0; i < meshes.size(); i++)
45          meshes[i].Draw(shader);
46    }
47
```

**Figure 23.103:** *The include file / header file* ***model.h*** *that is located in* ***/usr/include/learnopengl*** *contains the actual loading and translation code to create a class that represents a model in its entirety / as a whole.*

```
unsigned char *data = stbi_load("/root/SourceCodes/CPP/Assets/
    Textures/matrix.jpg", &width, &height, &nrChannels, 0);
if (data)
{
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
            GL_RGB, GL_UNSIGNED_BYTE, data);
       glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
       std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);
```

- We are going to talk about the loop inside the **while (!glfwWindowShouldClose(window))**.

  We follow the code and logic from [16] book to move around the camera with keyboard and mouse input. Then we bind the texture, enable shader and we use **GLM** to set the camera, the view, model and projection matrix.

  1. The background screen is yellow because when we first render the screen we set this **glClearColor(1.0f, 1.0f, 0.05f, 1.0f)**.
  2. The projection is using perspective projection, this is the more realistic one, compared to orthographic projection, the object that is far from our eye / point of view will looks smaller.
  3. The model will be located in $(0, 0, 0)$ in the 3D world.
  4. We scale down the model, make the 3D object smaller around 23% of its' original size with **glm::scale(model, glm::vec3(0.23f, 0.23f, 0.23f))**.
  5. Remember that OpenGL uses right-handed system, the positive $x$-axis is at the right side of the monitor that we see, the positive $y$-axis is facing to the top of the monitor that we see, and the positive $z$-axis is going toward us from the monitor.

  After finish, then rendered, we draw the shader.

```
while (!glfwWindowShouldClose(window))
{
       // per-frame time logic
       // --------------------
       float currentFrame = static_cast<float>(glfwGetTime());
       deltaTime = currentFrame - lastFrame;
       lastFrame = currentFrame;

       // input
       // -----
       processInput(window);

       // render
       // ------
       glClearColor(1.0f, 1.0f, 0.05f, 1.0f);
```

```
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // bind Texture
        glBindTexture(GL_TEXTURE_2D, texture);

        // don't forget to enable shader before setting uniforms
        ourShader.use();

        // view/projection transformations
        glm::mat4 projection = glm::perspective(glm::radians(
            camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT,
            0.1f, 100.0f);
        glm::mat4 view = camera.GetViewMatrix();
        ourShader.setMat4("projection", projection);
        ourShader.setMat4("view", view);

        // render the loaded model
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f
            )); // translate the object
        model = glm::scale(model, glm::vec3(0.23f, 0.23f, 0.23f)
            ); // if the model is too big for our scene, scale
            it down here
        ourShader.setMat4("model", model);
        ourModel.Draw(ourShader);

        // glfw: swap buffers and poll IO events (keys pressed/
            released, mouse moved etc.)
        // -------------------------------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

- The view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction.
  In **OpenGL** with the help of **GLM** in order to define a camera we need three vectors that define:
  1. The camera' position.
  2. The direction that the camera facing to.
  3. A vector pointing upward (the normal vector) so the camera is standing up.

  Below is an example of creating a view matrix, consists of three vectors above

```
    #include <glm/glm.hpp>
    #include <glm/gtc/matrix_transform.hpp>

    // Set up MVP matrices

    // model matrix
```

```
glm::mat4 model = glm::mat4(1.0f);

// projection matrix
glm::mat4 Projection = glm::perspective(45.0f, 800.0f / 600.0f,
     1.0f, 10.0f);

// camera matrix / view matrix
glm::mat4 View = glm::lookAt(
glm::vec3(4,3,3), // the camera is at (4,3,3) in world space
    coordinate
glm::vec3(0,0,0), // the camera is looking at the origin
glm::vec3(0,1,0) // The camera' head is up
);

glm::mat4 mvp = proj * view * model;

glUseProgram(prog);

GLint uniMvp = glGetUniformLocation(program, "mvp");
glUniformMatrix4fv(uniMvp, 1, GL_FALSE, glm::value_ptr(mvp));
```

The model matrix we created, **glm::mat4 model = glm::mat4(1.0f)**, can then be used furthermore to apply certain matrix transformation to the 3D object / the model.

We use **glUseProgram()** to activate the shader.
We then send the **mvp** matrix (the multiplication of mode, view and projection matrix) to the vertex shader like below

```
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 mvp;

void main()
{
       gl_Position = mvp * vec4(aPos, 1.0);
}
```

Don't send all the 3 model, view, projection matrices separately and calculate it in the shader. A lot of people suggest to pre-calculate $M_{projection} \times M_{view} \times M_{model}$ matrix before send it to shader.

Verify the shader program has compiled and linked correctly and make sure uniforms are valid, which means the location value is / must be $\geq 0$. After compiling the shaders, Make sure check **glGetProgram(GL_LINK_STATUS,...)**.

A binary search with **glGetError()** can also be very useful in these instances when you don't know where the error is.

**Figure 23.104:** *The 3-dimensional object here is a backpack with texture of a container. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Object Load with Assimp and OpenGL/main.cpp**).*

## xliii. C++ Plot: Apply Matrix Transformation Translation and Rotation to 3-Dimensional Object with GLM, Assimp and OpenGL

After we are able to import 3D object into the scene, now we want to do matrix transformation on the object, which is translation and rotation at the same time, transrotation is the cool combination word.

The C++ code and the libraries used are mostly stay the same with the previous section, the only difference is inside the **while (!glfwWindowShouldClose(window))** loop, in this part we just need to adjust the **model matrix**.

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#define STB_IMAGE_IMPLEMENTATION
//#include "stb_image.h" // we do not use this since ' #include "stb_image.h" '
    already included in /usr/include/learnopengl/model.h

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>
#include <learnopengl/camera.h>
```

**Figure 23.105:** *The 3-dimensional object here is a spider with texture of matrix. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Object Load with Assimp and OpenGL/main.cpp**).*

```
#include <learnopengl/model.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 1200;
const unsigned int SCR_HEIGHT = 800;

// camera
Camera camera(glm::vec3(0.0f, 0.0f, 30.0f));
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;

// timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;

int main()
{
	// glfw: initialize and configure
	// ------------------------------
	glfwInit();
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	#ifdef __APPLE__
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	#endif

	// glfw window creation
	// --------------------
	GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "
	    LearnOpenGL", NULL, NULL);
	if (window == NULL)
	{
		std::cout << "Failed to create GLFW window" << std::endl;
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);
	glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

```
        glfwSetCursorPosCallback(window, mouse_callback);
        glfwSetScrollCallback(window, scroll_callback);

        // tell GLFW to capture our mouse
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

        // glad: load all OpenGL function pointers
        // ————————————
        if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
        {
               std::cout << "Failed to initialize GLAD" << std::endl;
               return -1;
        }

        // tell stb_image.h to flip loaded texture's on the y-axis (before
            loading model).
        stbi_set_flip_vertically_on_load(true);

        // configure global opengl state
        // ————————————
        glEnable(GL_DEPTH_TEST);

        // build and compile shaders
        // ————————————
        Shader ourShader("model_loading.vs", "model_loading.fs");

        // load models
        // ————————————
        // The default filesystem is the first place we build LearnOpenGL at
             /sources/Hamzstlab/LearnOpenGL
        //Model ourModel(FileSystem::getPath("backpack.obj"));
        Model ourModel("airplane.obj");

        // load and create a texture
        // ————————————
        unsigned int texture;
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture); // all upcoming GL_TEXTURE_2D
             operations now have effect on this texture object
        // set the texture wrapping parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // set
             texture wrapping to GL_REPEAT (default wrapping method)
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        // set texture filtering parameters
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // load image, create texture and generate mipmaps
```

```
        int width, height, nrChannels;
        unsigned char *data = stbi_load("/root/SourceCodes/CPP/Assets/
            Textures/freya1.jpg", &width, &height, &nrChannels, 0);
        if (data)
        {
               glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, data);
               glGenerateMipmap(GL_TEXTURE_2D);
        }
        else
        {
               std::cout << "Failed to load texture" << std::endl;
        }
        stbi_image_free(data);


        // draw in wireframe
        //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        float X = 0, Y = 0, Z = 0;
        float percent = 0.01f;
        int direction = 1;
        float theta = 0;

        // render loop
        // -----------
        while (!glfwWindowShouldClose(window))
        {
               // per-frame time logic
               // -----------
               float currentFrame = static_cast<float>(glfwGetTime());
               deltaTime = currentFrame - lastFrame;
               lastFrame = currentFrame;

               // input
               // -----
               processInput(window);

               // render
               // ------
               glClearColor(0.30f, 0.55f, 0.65f, 1.0f);
               glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

               // bind Texture
               glBindTexture(GL_TEXTURE_2D, texture);

               // don't forget to enable shader before setting uniforms
               ourShader.use();
```

```
// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(camera.
    Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f
    );
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("projection", projection);
ourShader.setMat4("view", view);

// render the loaded model
glm::mat4 model = glm::mat4(1.0f);

// to scale the model
glm::mat4 scale_mat = glm::mat4(1.0f);
scale_mat = glm::scale(model, glm::vec3(0.005f, 0.005f, 0.005
    f)); // if the model is too big for our scene, scale it
    down here
// for airplane.obj best at glm::scale(model, glm::vec3(0.005
    f, 0.005f, 0.005f))

// to translate the model
glm::mat4 translation_mat = glm::mat4(1.0f);
translation_mat = glm::translate(translation_mat, glm::vec3(X
    , Y, Z)); // translate the object

X += 2.0f * percent * direction;
Y += 1.0f * percent * direction;
Z += 2.0f * percent * direction;

if (X >= 7 || X <=0 )
{
       direction = -direction;
}

std::cout << "X = " << X << " \t --- \t Y = " << Y << "\t
    --- \t Z = " << Z << "\n";

// to rotate the model
theta +=1.7f ;
if (theta > 720)
{
       theta -=720;
}

glm::mat4 rotation_mat = glm::mat4(1.0f);
rotation_mat = glm::rotate(rotation_mat,1.5f*glm::radians(
    theta), glm::vec3(1.0f, 0.5f, 0.0f));
```

```
                // scale the model first, then rotate and translate
                model = translation_mat * rotation_mat * scale_mat ;
                ourShader.setMat4("model", model);
                ourModel.Draw(ourShader);

                // glfw: swap buffers and poll IO events (keys pressed/
                    released, mouse moved etc.)
                // -------------
                glfwSwapBuffers(window);
                glfwPollEvents();
        }

        // glfw: terminate, clearing all previously allocated GLFW resources
            .
        // -------------
        glfwTerminate();
        return 0;
}

// process all input: query GLFW whether relevant keys are pressed/released
     this frame and react accordingly
// -------------
void processInput(GLFWwindow *window)
{
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}

// glfw: whenever the window size changed (by OS or user resize) this
    callback function executes
// -------------
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
        // make sure the viewport matches the new window dimensions; note
            that width and
        // height will be significantly larger than specified on retina
            displays.
        glViewport(0, 0, width, height);
}
```

```
// glfw: whenever the mouse moves, this callback is called
// ------------
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
	float xpos = static_cast<float>(xposIn);
	float ypos = static_cast<float>(yposIn);

	if (firstMouse)
	{
		lastX = xpos;
		lastY = ypos;
		firstMouse = false;
	}

	float xoffset = xpos - lastX;
	float yoffset = lastY - ypos; // reversed since y-coordinates go
	    from bottom to top

	lastX = xpos;
	lastY = ypos;

	camera.ProcessMouseMovement(xoffset, yoffset);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback is called
// ------------
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
	camera.ProcessMouseScroll(static_cast<float>(yoffset));
}
```

**C++ Code 166:** *main.cpp "Apply transrotation to 3d object with opengl"*

To compile it, type:
**g++ -o main main.cpp /root/SourceCodes/CPP/src/glad.c -lstdc++ -lassimp -lglfw -lGL**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Before the **while (!glfwWindowShouldClose(window))** we declare six variables / parameters for the translation and rotation transformation.
  1. The $X, Y, Z$ will be the coordinate of the 3D object.

The increment of each axis position is different and each is multiplied by *percent* and *direction* variables.

2. The *percent* will be the increment of the translation.
3. The *direction* will be the direction of the translation, toward the positive axis means positive 1 here.
   When the 3D object reach the $X = 7$ or $X = 0$ position then the direction will be $- \times (current\ direction)$, meaning it will translate to the opposite direction (to make the airplane move back and forth).
4. The *theta* is the angle of rotation that will have increment of 1.7 till reach 720 then reset to 0 again.

We create 4 matrices

1. **glm::mat4 model = glm::mat4(1.0f);**
2. **glm::mat4 scale_mat = glm::mat4(1.0f);**
   This matrix will be responsible to scale the object, if the object is too big and does not fit the screen, we cannot see anything because it will be clipped.
3. **glm::mat4 translation_mat = glm::mat4(1.0f);**
   This matrix will perform the translation with the **glm::translate(translation_mat,...)** function.
4. **glm::mat4 rotation_mat = glm::mat4(1.0f);**
   This matrix will perform the translation with the **glm::rotate(rotation_mat, ... , ...)** function.

After all the scale, rotate, and translate matrices have been defined and computed, we compute the **model matrix** that will be sent to the vertex shader, the model matrix is a multiplication of all the **scale matrix, rotation matrix,** and **translation matrix**. The order matters since in matrix multiplication, the first transformation will be at the outerright of the multiplication which in this case is the **scale_mat**. **model = translation_mat * rotation_mat * scale_mat**

```
float X = 0, Y = 0, Z = 0;
float percent = 0.01f;
int direction = 1;
float theta = 0;

// render loop
// -------------
while (!glfwWindowShouldClose(window))
{
    // per-frame time logic
    // -------------
    float currentFrame = static_cast<float>(glfwGetTime());
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // input
    // -----
    processInput(window);
```

```
// render
// ------
glClearColor(0.30f, 0.55f, 0.65f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// bind Texture
glBindTexture(GL_TEXTURE_2D, texture);

// don't forget to enable shader before setting uniforms
ourShader.use();

// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(
    camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT,
    0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("projection", projection);
ourShader.setMat4("view", view);

// render the loaded model
glm::mat4 model = glm::mat4(1.0f);

// to scale the model
glm::mat4 scale_mat = glm::mat4(1.0f);
scale_mat = glm::scale(model, glm::vec3(0.005f, 0.005f,
    0.005f)); // if the model is too big for our scene,
    scale it down here
// for airplane.obj best at glm::scale(model, glm::vec3
    (0.005f, 0.005f, 0.005f))

// to translate the model
glm::mat4 translation_mat = glm::mat4(1.0f);
translation_mat = glm::translate(translation_mat, glm::
    vec3(X, Y, Z)); // translate the object

X += 2.0f * percent * direction;
Y += 1.0f * percent * direction;
Z += 2.0f * percent * direction;

if (X >= 7 || X <=0 )
{
       direction = -direction;
}

std::cout << "X = " << X << " \t --- \t Y = " << Y <<
    "\t --- \t Z = " << Z << "\n";
```

```
                // to rotate the model
                theta +=1.7f ;
                if (theta > 720)
                {
                        theta -=720;
                }

                glm::mat4 rotation_mat = glm::mat4(1.0f);
                rotation_mat = glm::rotate(rotation_mat,1.5f*glm::
                    radians(theta), glm::vec3(1.0f, 0.5f, 0.0f));

                // scale the model first, then rotate and translate
                model = translation_mat * rotation_mat * scale_mat ;
                ourShader.setMat4("model", model);
                ourModel.Draw(ourShader);

                // glfw: swap buffers and poll IO events (keys pressed/
                    released, mouse moved etc.)
                // -------------
                glfwSwapBuffers(window);
                glfwPollEvents();
        }

        // glfw: terminate, clearing all previously allocated GLFW
            resources.
        // -------------
        glfwTerminate();
        return 0;
}
```

**Figure 23.106:** *The airplane / 3d object that is imported with* ***Assimp*** *is doing continuous translation and rotation.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Object Load and TransRotating Continuously with Assimp and OpenGL/main.cpp)****.*

## xliv. C++ Computation: Time Share as a Dynamical System with Power Matrix

On 2019, a young Glanz just got kicked out and fired from all her consulting jobs, thus she quits her ice skating lesson, piano lesson, and violin lesson to start a new cycle of life that cost less money, and be closer to Nature. On 2020, she by accident heard that there is a place on top of the mountain with a lot of pine trees that are very breathtaking and the air is very fresh there, the name is Puncak Bintang (PB) thus she went there everyday, and by accident one day she stopped by at Valhalla Projection (VP), she named the forest as VP because since April 2022 she dilligently cut grasses there, help clean up forest in VP at the lower area where people used to throw trashes in forest without knowing bad karma exists and worse than go to jail or get a fine of USD 10,000. Thus VP become a very beautiful place that is now filled with her hand made 7 chakras shape from branches that fallen from trees, a gigantic FREYA made out of grasses, and more, to remember that place is like the reflection of Valhalla up there in the divine realm / heaven that is located on earth.

The problem is she has only 24 hours a day, she cannot allocate her time from morning to night to go outside, thus she has to choose to only go to either VP or PB for a day, she can go to both place but then she will come back home really late, and she won't be focus. She has to study, read, write a book when she is back at home so she can start making passive income as an author and stocks investor. The conditions for PB are

1. On lebaran 2021, she got a lot of wafers, chocolate, kastengels, some still fresh, but looks someone just left it on Puncak Bintang, thus the nature in Puncak Bintang, Kala, says to her it is a gift for her.

2. She is able to create GFreya OS from Linux From Scratch then BLFS book version 11 System V when she went to PB everyday from 2021 to early 2022.

3. She writes Arduino memo and learn Microcontroller and electrical engineering dilligently, wake up at 1 am to read and summarize electrical engineering book when she went to PB everyday.

4. She learns Russian language by herself when she went to PB everyday.

5. Always send flowers (fireworks) when Glanz' dogs passed away.

6. Taught her martial arts and pinca.

Now for VP:

1. On lebaran 2023, she got more than promised cookies, suppose only 7 boxes, she gots more of Coklat Mede and Nastar. Not a leftover by unknown person like how Kala gave her back in 2021.

2. Her Mother suddenly receive a nice car that can be used on January 2024, that is when she still got to VP everyday.

3. She writes Lasthrim Projection, DianFreya Math Physics Simulator with C++ when she goes to VP everyday. She learns JULIA, Python, C++ more focus when she goes to VP everyday.

4. She learns French language by herself when she went to VP everyday.

5. She fixed a lot of problems in GFreya OS from sounds, to cursor problem.
6. She summarize Intelligent Investor by Benjamin Graham and wrote annual report of how she invests in stocks and able to learn from past mistakes thus she can one day make money like Lo Kheng Hong (USD 200 million) then like Warren Buffett (USD 100 billion) in the future while still able to pursue her hobby and love to write STEM books, be engineer, innovator / inventor, and does not need to work as employee anymore.
7. Always give Glanz a land or allocate a small place to bury animals that pass away, from dogs, cats, birds to frogs.
8. Taught her martial arts and handstand.

Suppose that each VP and PB have 50% of Glanz' time that she allocates to visit. Assume that after visiting VP over one-year period, VP can capture Glanz' time that she allocate for PB around 18%, and can retain 97% of its initial 50% time share since VP is closer she does not have to walk further, and all Nature in VP are really protective toward her and her pets, and they can have a big playground in VP forest. While if Glanz is visiting PB, PB can capture Glanz' time that she allocate for VP around 3% and can retain 82% of its initial 50% time share, because PB is located further near the top of mountain, do Yoga / Ashtanga and workout there will sometimes be bothered by celebrity that come occasionally and her Yoga spot will be taken, car comes and park on her Yoga spot, while in VP she gets all the privacy she needs, and in VP she can unleash her creativity to shape the forest too, while in PB she can only do workouts.

Track the time shares to go to VP and PB over a five-year period, 10-year period, and 108-year period. Considering living on mountain and helping Nature will give Glanz benefits more than all employees in the world, long and healthy life can even surpass 1000 years like prophets in the past, and more abundance and wealth coming without seeking.

**Solution:**

Let us begin by introducing the time-dependent variables
$x_1(t)$ =fraction of Glanz' time held by Valhalla Projection (VP) at time $t$.
$x_2(t)$ =fraction of Glanz' time held by Puncak Bintang (PB) at time $t$.

and the column vector

$$\boldsymbol{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$$

The variables $x_1(t)$ and $x_2(t)$ form a dynamical system whose state at time $t$ is the vector $\boldsymbol{x}(t)$, if we take $t = 0$ to be the starting point, then the state of the system at that time is

$$\boldsymbol{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

Now let us try to find the state of the system at time $t = 1$ (one year later), for VP

$$\boldsymbol{x}_1(1) = 0.97(0.5) + 0.18(0.5) = 0.575$$

for PB

$$\boldsymbol{x}_2(1) = 0.03(0.5) + 0.82(0.5) = 0.425$$

Therefore, the state of the system at time $t = 1$ is

$$\boldsymbol{x}(1) = \begin{bmatrix} x_1(1) \\ x_2(1) \end{bmatrix} = \begin{bmatrix} 0.575 \\ 0.425 \end{bmatrix}$$

Now we will track the time shares that Glanz allocate to go to VP and PB over five-year period. The formula is simple

$$\begin{aligned} x_1(k+1) &= (0.97)x_1(k) + (0.18)x_2(k) \\ x_2(k+1) &= (0.03)x_1(k) + (0.82)x_2(k) \end{aligned}$$

in matrix form it can be expressed as

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} 0.97 & 0.18 \\ 0.03 & 0.82 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix}$$

which provides a way of using matrix multiplication to compute the state of the system at time $t = k + 1$ from the state at time $t = k$.

$$\boldsymbol{x}(t+1) = \begin{bmatrix} 0.97 & 0.18 \\ 0.03 & 0.82 \end{bmatrix} \boldsymbol{x}(t)$$

with $t = 0, 1, 2, \ldots$. Therefore

$$\boldsymbol{x}(5) = \begin{bmatrix} 0.97 & 0.18 \\ 0.03 & 0.82 \end{bmatrix} \boldsymbol{x}(4) = \begin{bmatrix} 0.97 & 0.18 \\ 0.03 & 0.82 \end{bmatrix}^5 \boldsymbol{x}(0) = \begin{bmatrix} 0.7472 \\ 0.2528 \end{bmatrix}$$

After 5 years, VP will hold about 75% of Glanz' time and PB will hold about 25% of Glanz' time, for 10-year period

$$\boldsymbol{x}(10) = \begin{bmatrix} 0.8233 \\ 0.1767 \end{bmatrix}$$

After 10 years, VP will hold about 82% of Glanz' time and PB will hold about 18% of Glanz' time, now for 108-year period

$$\boldsymbol{x}(108) = \begin{bmatrix} 0.8571 \\ 0.1429 \end{bmatrix}$$

After 108 years, VP will hold about 85% of Glanz' time and PB will hold about 15% of Glanz' time.

For the C++ codes, we only use **Armadillo** library to compute $n$-th power of a matrix with **powmat** function. The code is really simple, intuitive and this dynamical system can be seen like convergence to certain number when $t$ goes to infinity.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(void)
{
        mat A(2,2,fill::zeros);
        mat x(2,1,fill::zeros);
```

```
        A = { { 0.97, 0.18 },
              { 0.03, 0.82 } };
        x.load("vectorX.txt");

        A.print("A:\n");
        x.print("x:\n");

        mat A2(2,2,fill::zeros);
        mat A5(2,2,fill::zeros);
        mat A10(2,2,fill::zeros);
        mat A108(2,2,fill::zeros);

        A2 = powmat(A,2);
        A5 = powmat(A,5);
        A10 = powmat(A,10);
        A108 = powmat(A,108);

        cout << "A^2 = \n" << A2 << endl;
        cout << "A^5 = \n" << A5 << endl;

        cout << "A*x = \n" << A*x << endl;
        cout << "(A^2)*x = \n" << A2*x << endl;
        cout << "(A^5)*x = \n" << A5*x << endl;
        cout << "(A^10)*x = \n" << A10*x << endl;
        cout << "(A^108)*x = \n" << A108*x << endl;

        return 0;
    }
```

**C++ Code 167:** *main.cpp "Dynamical system with power matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
A:

   0.9700   0.1800
   0.0300   0.8200
x:

   0.5000
   0.5000
A^2 =
   0.9463   0.3222
   0.0537   0.6778

A^5 =
   0.9011   0.5934
   0.0989   0.4066

A*x =
   0.5750
   0.4250

(A^2)*x =
   0.6342
   0.3657

(A^5)*x =
   0.7472
   0.2528

(A^10)*x =
   0.8233
   0.1767

(A^108)*x =
   0.8571
   0.1429
```

**Figure 23.107:** *Dynamical system computation for the time share example with Armadillo library. (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Dynamical System Power Matrix with Armadillo/main.cpp**).*

## xlv. C++ Computation: Determine Steady-State Vector for 2-States Markov Chain

Suppose we have the transition matrix for the Markov chain as

$$P = \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.9 \end{bmatrix}$$

Since the entries of $P$ are positive, the Markov chain is regular and hence has a unique steady-state vector $\boldsymbol{q}$. To find $\boldsymbol{q}$ we will solve the system

$$(I - P)\boldsymbol{q} = \mathbf{0}$$

which we can write as

$$(I - P)q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.9 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0.2 & -0.1 \\ -0.2 & 0.1 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The general solution of this system is

$$q_1 = 0.5s, \quad q_2 = s$$

with $s$ is the parameter.

We can write in vector form as

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0.5s \\ s \end{bmatrix}$$

For $\boldsymbol{q}$ to be a probability vector, we must have

$$1 = q_1 + q_2 = \frac{3}{2}s$$

which implies that $s = \frac{2}{3}$, substituting this will yields the steady-state vector

$$\boldsymbol{q} = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \end{bmatrix}$$

which is consistent with the numerical results if we set $k \to \infty$ when computing $P^k \boldsymbol{x}_0$, it is indeed will converges to $\boldsymbol{q}$.

For the C++ code, we are using **GiNaC** library to do symbolic computation with matrix multiplication, arithmetic operations, and to solve a linear system so we are able to determine the steady-state vector.

```
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

int main()
{
        Digits = 5; // define maximum decimal digits
        symbol q1("q1"), q2("q2");

        matrix P = {{0.8, 0.1}, {0.2, 0.9}};
        matrix Q = {{q1}, {q2}};

        cout << "\n P = " << P << endl;
        cout << "\n q = " << Q << endl;
        cout << "\n I = " << unit_matrix(2) << endl;

        // construct an expression (e/e2) containing matrices with
            arithmetic operators
        ex e = unit_matrix(2) - P;
        ex e2 = (unit_matrix(2) - P)*Q;
        ex e3 = e2.evalm()[0].subs(q1 == 1-q2);
        ex q2_ans = lsolve(e3 == 0 , q2) ;

        //cout << "det(P) = " << determinant(P) << endl;

        cout << endl;
        cout << "I-P = " << e << "=\n" << e.evalm() << endl;
        cout << endl;
```

```
        cout << "(I—P)q = " << e2 << "=\n" << e2.evalm() << endl;
        cout << endl;

        cout << endl;
        cout << "(I—P)q [q1 = 1—q2]= " << e2.evalm()[0].subs(q1 ==1—q2) <<
             endl;
        cout << endl;

        cout << "q2 = " << lsolve(e3 == 0 , q2) << endl;
        cout << "q1 = 1 — q2 = " << 1 — q2_ans << endl;
        cout << endl;

        matrix q = {{1 — q2_ans}, {q2_ans}};
        cout << "q = " << q << endl;

        return 0;
    }
```

**C++ Code 168:** *main.cpp "Determine steady state vector for two states markov chain"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In this part we are constructing 4 expressions from computing $I - P$, then to compute $(I - P)\boldsymbol{q}$ till we are able to compute $q_2$ from already known probability rule that $q_1 + q_2 = 1$. We do not declare $I$ at first, we just use the already established function in **GiNaC** to create special matrix like in this case, a unit matrix of size 2.

```
        ex e = unit_matrix(2) — P;
        ex e2 = (unit_matrix(2) — P)*Q;
        ex e3 = e2.evalm()[0].subs(q1 == 1—q2);
        ex q2_ans = lsolve(e3 == 0 , q2) ;
```

  For **e3** expression, what we mean by **e2.evalm()[0].subs(q1 ==1-q2);** is that we are taking the computed value of expresion **e2** take the first row and substitute the symbolic $q_1$ to become $1 - q_2$.

```
 P = [[0.8,0.1],[0.2,0.9]]

 q = [[q1],[q2]]

 I = [[1,0],[0,1]]

I-P = [[1,0],[0,1]]-[[0.8,0.1],[0.2,0.9]]=
[[0.19999999,-0.1],[-0.2,0.100000024]]

(I-P)q = ([[1,0],[0,1]]-[[0.8,0.1],[0.2,0.9]])*[[q1],[q2]]=
[[-(0.1)*q2+(0.19999999)*q1],[(0.100000024)*q2-(0.2)*q1]]


(I-P)q [q1 = 1-q2]= 0.19999999-(0.29999998)*q2

q2 = 0.6666667
q1 = 1 - q2 = 0.3333333

q = [[0.3333333],[0.6666667]]
```

**Figure 23.108:** *The computation to find the steady-state vector for transition matrix for the Markov chain that only have two states. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Steady-State Vector Markov Chain Transition Matrix with GiNaC/main.cpp**).*

## xlvi. C++ Computation: Wild Glanz Migration as a Three-States Markov Chain

Suppose that Glanz is going to find a nice place to do workout, improve her physical, spiritual and mental power, exercise with Tai Chi, FoY, and Ashtanga Yoga. There are 3 places on the mountain she lives on that can accomodate her needs, they are VP (Valhalla Projection), PB (Puncak Bintang), and Kathmandu. Based on data from 2020 Glanz' migration pattern of moving to certain place can be modeled by a Markov chain with transition matrix

$$P = \begin{bmatrix} 0.97 & 0.51 & 0.6 \\ 0.02 & 0.33 & 0.3 \\ 0.01 & 0.16 & 0.1 \end{bmatrix}$$

That is
$p_{11} = 0.97 =$ probability that Glanz will go to VP tomorrow when she already in VP today.
$p_{12} = 0.51 =$ probability that Glanz will go to VP tomorrow when she already in PB today.
$p_{13} = 0.6 =$ probability that Glanz will go to VP tomorrow when she already in Kathmandu today.
$p_{21} = 0.02 =$ probability that Glanz will go to PB tomorrow when she already in VP today.
$p_{22} = 0.33 =$ probability that Glanz will go to PB tomorrow when she already in PB today.
$p_{23} = 0.3 =$ probability that Glanz will go to PB tomorrow when she already in Kathmandu today.
$p_{31} = 0.01 =$ probability that Glanz will go to Kathmandu tomorrow when she already in VP today.
$p_{32} = 0.16 =$ probability that Glanz will go to Kathmandu tomorrow when she already in PB today.
$p_{33} = 0.1 =$ probability that Glanz will go to Kathmandu tomorrow when she already in Kathmandu today.

Assuming that $t$ is in day and Glanz started all her new cycle of becoming Double-Star Hunter, Nature Golden Child and True Earth Rune bearer from Puncak Bintang at time $t = 0$ (in 2020). Track Glanz' probable locations over a 7-days and after 46-days period. Then determine the steady-state vector $\boldsymbol{q}$, if exist.

**Solution:**

Let $x_1(k), x_2(k)$, and $x_3(k)$ be the probabilities that Glanz is in VP, PB, and Kathmandu, respectively, at time $t = k$, and let

$$\boldsymbol{x}(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \\ x_3(k) \end{bmatrix}$$

be the state vector at that time. Since we know with certainty that Glanz is in PB at time $t = 0$, the initial state vector is

$$\boldsymbol{x}(0) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The state vector over a 7-days period and then after 46-days are

$$\boldsymbol{x}(1) = P\boldsymbol{x}(0) = \begin{bmatrix} 0.51 \\ 0.33 \\ 0.16 \end{bmatrix}, \quad \boldsymbol{x}(2) = P\boldsymbol{x}(1) = \begin{bmatrix} 0.759 \\ 0.1671 \\ 0.0739 \end{bmatrix}, \quad \boldsymbol{x}(3) = P\boldsymbol{x}(2) = \begin{bmatrix} 0.8658 \\ 0.0925 \\ 0.0417 \end{bmatrix}$$

$$\boldsymbol{x}(4) = P\boldsymbol{x}(3) = \begin{bmatrix} 0.912 \\ 0.0604 \\ 0.0276 \end{bmatrix}, \quad \boldsymbol{x}(5) = P\boldsymbol{x}(4) = \begin{bmatrix} 0.9320 \\ 0.0464 \\ 0.0215 \end{bmatrix}, \quad \boldsymbol{x}(6) = P\boldsymbol{x}(5) = \begin{bmatrix} 0.9407 \\ 0.0404 \\ 0.0189 \end{bmatrix}$$

$$\boldsymbol{x}(7) = P\boldsymbol{x}(6) = \begin{bmatrix} 0.9444 \\ 0.0378 \\ 0.0178 \end{bmatrix}, \ldots, \boldsymbol{x}(46) = P\boldsymbol{x}(45) = \begin{bmatrix} 0.9473 \\ 0.0358 \\ 0.0169 \end{bmatrix}$$

Now we take a look again at the entries at matrix $P$

$$P = \begin{bmatrix} 0.97 & 0.51 & 0.6 \\ 0.02 & 0.33 & 0.3 \\ 0.01 & 0.16 & 0.1 \end{bmatrix}$$

Since the entries of $P$ are positive, the Markov chain is regular and hence has a unique steady-state vector $\boldsymbol{q}$. To find $\boldsymbol{q}$ we will solve the system $(I - P)\boldsymbol{q} = \mathbf{0}$, which we can write as

$$(I - P)\boldsymbol{q} = \mathbf{0}$$

$$\begin{bmatrix} 0.03 & -0.51 & -0.6 \\ -0.02 & 0.67 & -0.3 \\ -0.01 & -0.16 & 0.9 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

One-way to determine $q_1, q_2, q_3$ is to reduce the matrix $(I - P)$ into reduced row echelon form then take parameter solution and then use substitution of the knowledge that $q_1 + q_2 + q_3 = 1$.

Since Gauss-Jordan elimination especially for fraction is not an easy task to be done, thus after we retrieve three equations from $(I - P)\boldsymbol{q}$, all equations are homogeneous (equal to 0), then substitute this

$$q_3 = 1 - q_1 - q_2$$

so we have three equations with 2 unknowns. It is easy to be done now, we can just take two out of the three equations (random picking is not a problem) and then we can compute the value

for $q_1$ and $q_2$ with simple elimination method from algebra, after that we can get the value for $q_3$. This is the method of thinking of how we solve with C++ as well.

We will obtain

$$q_1 = 0.947260$$
$$q_2 = 0.035842$$
$$q_3 = 0.016897$$

the steady-state vector is

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0.947260 \\ 0.035842 \\ 0.016897 \end{bmatrix}$$

which is consistent with the results obtained above after 46-days period ($\boldsymbol{x}(46)$), the 46-days state vector converge to the steady-state vector.

We make two C++ codes here:

1. To compute the state vectors over a seven-days period we use **Armadillo** library.
2. To compute steady-state vector, with a little bit of arithmetic manipulation and linear algebra computation we use **GiNaC** library.

For the first code.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(void)
{
        mat P(3,3,fill::zeros);
        mat x(3,1,fill::zeros);
        P = { { 0.97, 0.51, 0.6 },
                { 0.02, 0.33, 0.3},
                { 0.01, 0.16, 0.1 } };
        x.load("vectorX.txt");

        P.print("P:\n");
        x.print("x(0):\n");

        for (int i = 1; i < 8; i++)
        {
                cout << "x(" << i <<") = \n" << powmat(P,i)*x << endl;
        }

        return 0;
}
```

**C++ Code 169:** *main.cpp "State vectors for three states markov chain"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

For the second code.

```
#include <iostream>
#include <ginac/ginac.h>

// Merci beaucoup Freya et Sentinel

using namespace std;
using namespace GiNaC;

int main()
{
        Digits = 5; // define maximum decimal digits
        symbol q1("q1"), q2("q2"), q3("q3");

        matrix P = { { 0.97, 0.51, 0.6 }, { 0.02, 0.33, 0.3}, { 0.01, 0.16,
            0.1 } };
        matrix Q = {{q1}, {q2}, {q3}};

        cout << "\n P = " << P << endl;
        cout << "\n q = " << Q << endl;
        cout << "\n I = " << unit_matrix(3) << endl;

        // construct an expression (e/e2) containing matrices with
            arithmetic operators
        ex e = unit_matrix(3) - P;
        ex e2 = (unit_matrix(3) - P)*Q;
        ex e31 = e2.evalm()[0].subs(q3 ==1-q2-q1);
        ex e32 = e2.evalm()[1].subs(q3 ==1-q2-q1);
        ex e33 = e2.evalm()[2].subs(q3 ==1-q2-q1);

        //cout << "det(P) = " << determinant(P) << endl;

        cout << endl;
        cout << "I-P = " << e << "=\n" << e.evalm() << endl;
        cout << endl;

        cout << "(I-P)q = " << e2 << "=\n" << e2.evalm() << endl;
```

```
        cout << endl;

        cout << endl;
        cout << "Equation 1 = " << e31<< endl;
        cout << endl;
        cout << "Equation 2 = " << e32<< endl;
        cout << endl;
        cout << "Equation 3 = " << e33<< endl;
        cout << endl;

        lst eqns = {e32 == 0, e33 ==0}; // list all equations need to be
            solved
        lst vars = {q1,q2}; // list the variables we want to solve
        ex q1_ans = lsolve(eqns, vars)[0].rhs(); // accessing the right-
            hand side expression
        ex q2_ans = lsolve(eqns, vars)[1].rhs();
        ex q3_ans = 1 - q1_ans - q2_ans;

        cout << "q1 = " << q1_ans <<endl;
        cout << "q2 = " << q2_ans <<endl;
        cout << "q3 = " << q3_ans<< endl;
        cout << endl;

        matrix q = {{q1_ans}, {q2_ans}, {q3_ans}};
        cout << "q = " << q << endl;

        return 0;
    }
```

**C++ Code 170:** *main.cpp "Steady state vector for three states markov chain"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We construct an expression:
  $e = (I - P)$.
  $e2 = (I - P)\boldsymbol{q}$.
  $e31 =$ the first equation of $(I - P)\boldsymbol{q}$ with a substitution of $q3 = 1 - q_1 - q_2$.
  $e32 =$ the second equation of $(I - P)\boldsymbol{q}$ with a substitution of $q3 = 1 - q_1 - q_2$.
  $e33 =$ the third equation of $(I - P)\boldsymbol{q}$ with a substitution of $q3 = 1 - q_1 - q_2$.

```
        ex e = unit_matrix(3) - P;
```

```
ex e2 = (unit_matrix(3) - P)*Q;
ex e31 = e2.evalm()[0].subs(q3 ==1-q2-q1);
ex e32 = e2.evalm()[1].subs(q3 ==1-q2-q1);
ex e33 = e2.evalm()[2].subs(q3 ==1-q2-q1);
```

- We construct a list of equations and variables need to be solved:
  $e32 == 0, e33 == 0$ are the second and third equation from $(I - P)\boldsymbol{q} = \mathbf{0}$ with a substitution of $q3 = 1 - q_1 - q_2$ that already take place.

  We list the variables we want to find the value in **lst vars** as **q1, q2**.

  With the function **lsolve()** we can solve two equations with two unknowns and obtain the result numerically with the method **.rhs()**, this method should be remembered since it will be very handy for the future.

```
lst eqns = {e32 == 0, e33 ==0};
lst vars = {q1,q2};
ex q1_ans = lsolve(eqns, vars)[0].rhs();
ex q2_ans = lsolve(eqns, vars)[1].rhs();
ex q3_ans = 1 - q1_ans - q2_ans;
```

```
P:

   0.9700   0.5100   0.6000
   0.0200   0.3300   0.3000
   0.0100   0.1600   0.1000
x(0):

         0
   1.0000
         0
x(1) =
   0.5100
   0.3300
   0.1600

x(2) =
   0.7590
   0.1671
   0.0739

x(3) =
   0.8658
   0.0925
   0.0417

x(4) =
   0.9120
   0.0604
   0.0276

x(5) =
   0.9320
   0.0464
   0.0215

x(6) =
   0.9407
   0.0404
   0.0189

x(7) =
   0.9444
   0.0378
   0.0178
```

**Figure 23.109:** *The computation to find the state vectors over a seven-days period for the Markov chain that have three states. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Dynamical System Power Matrix 3 States Markov Chain with Armadillo/main.cpp**).*

```
Steady-State Vector Markov Chain Transition Matrix 3 States with GiNaC  ]# ./main

 P = [[0.97,0.51,0.6],[0.02,0.33,0.3],[0.01,0.16,0.1]]

 q = [[q1],[q2],[q3]]

 I = [[1,0,0],[0,1,0],[0,0,1]]

I-P = [[1,0,0],[0,1,0],[0,0,1]]-[[0.97,0.51,0.6],[0.02,0.33,0.3],[0.01,0.16,0.1]]=
[[0.029999971,-0.51,-0.6],[-0.02,0.66999996,-0.3],[-0.01,-0.16,0.9]]

(I-P)q = ([[1,0,0],[0,1,0],[0,0,1]]-[[0.97,0.51,0.6],[0.02,0.33,0.3],[0.01,0.16,0.1]])*[[q1],[q2],[q3]]=
[[(0.029999971)*q1-(0.51)*q2-(0.6)*q3],[-(0.02)*q1+(0.66999996)*q2-(0.3)*q3],[-(0.01)*q1-(0.16)*q2+(0.9)*q3]]


Equation 1 = -0.6+(0.63)*q1+(0.09000003)*q2

Equation 2 = -0.3+(0.28)*q1+(0.96999997)*q2

Equation 3 = 0.9-(0.90999997)*q1-(1.06)*q2

q1 = 0.9472606
q2 = 0.035842292
q3 = 0.01689709

q = [[0.9472606],[0.035842292],[0.01689709]]
```

**Figure 23.110:** *The computation to find the steady-state vector for the regular Markov chain that have three states (if you are a bit confused of the result it is not for the faint-heart). (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Steady-State Vector Markov Chain Transition Matrix 3 States with GiNaC/main.cpp**).*

## VI. Eigenvalues and Eigenvectors

[DF*] The word *eigen* is form the German word, meaning "characteristic". The underlying idea first appeared in the study of rotational motion but was later used to slassify various kinds of surfaces and to desribe solutions to certain differential equations. It has applications in various fields such as computer graphics, mechanical vibrations, heat flow, population dynamics, quantum mechanics, and economics.

[DF*] Let $A$ be a square matrix, $A \in \mathbb{R}^{n\times n}$. Many of the important properties of a matrix are summarized by its eigenvalues and eigenvectors [9].

**Definition 23.44: Eigenvalue and Eigenvector**

Let $\mathbb{C}$ denote the complex plane. $\lambda \in \mathbb{C}$ is an eigenvalue of $A$ (an $n \times n$ matrix) if and only if there is some $\boldsymbol{x} \in \mathbb{C}^n$, $\boldsymbol{x} \neq \mathbf{0}$ ( a nonzero vector $\boldsymbol{x}$ in $\mathbb{R}^n$ is called an eigenvector of $A$ / or the matrix operator $T_A$), such that

$$A\boldsymbol{x} = \lambda \boldsymbol{x} \tag{23.117}$$

for some scalar $\lambda$. The scalar $\lambda$ is called an eigenvalue of $A$.

**Definition 23.45: Spectrum and Spcetral Radius**

Let $\sigma(A)$ denote the set of eigenvalues of $A$; we refer to it as the spectrum of $A$. If $A\boldsymbol{x} = \lambda\boldsymbol{x}$ for a nonzero $\boldsymbol{x}$, then $\boldsymbol{x}$ is an eigenvector corresponding to $\lambda \in \sigma(A)$. The spectral radius of $A$ is

$$\rho(A) \equiv \max\{|\lambda| : \lambda \in \sigma(A)\} \tag{23.118}$$

If $\boldsymbol{x}$ is an eigenvector, then so is any scalar multiple of $\boldsymbol{x}$; hence eigenvectors are defined only up to a proportionality constant. Also $\lambda \in \sigma(A)$ if and only if

$$\det(A - \lambda I) = 0 \tag{23.119}$$

the characteristic equation of $A$, implying that $\lambda \in \sigma(A)$ if and only if $A - \lambda I$ is singular.

[DF*] In general, the image of a vector $\boldsymbol{x}$ under multiplication by a square matrix $A$ differs from $\boldsymbol{x}$ in both magnitude and direction.

However, in the special case where $\boldsymbol{x}$ is an eigenvector of $A$, multiplication by $A$ leaves the direction unchanged.

For example, in $\mathbb{R}^2$ or $\mathbb{R}^3$ multiplication by $A$ maps each eigenvector $\boldsymbol{x}$ of $A$ along the same line through the origin as $\boldsymbol{x}$.

Depending on the sign and magnitude of the eigenvalue $\lambda$ corresponding to $\boldsymbol{x}$, the operation

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

compresses or stretches $\boldsymbol{x}$ by a factor of $\lambda$, with a reversal of direction in the case where $\lambda$ is negative.

[DF*] The vector

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

is an eigenvector of

$$A = \begin{bmatrix} 3 & 0 \\ 8 & -1 \end{bmatrix}$$

corresponding to the eigenvalue $\lambda = 3$, since

$$A\boldsymbol{x} = \begin{bmatrix} 3 & 0 \\ 8 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix} = 3\boldsymbol{x}$$

Geometrically, multiplication by $A$ has stretched the vector $\boldsymbol{x}$ by a factor of 3.

[DF*] Note first that the equation

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

can be rewritten as

$$A\boldsymbol{x} = \lambda I\boldsymbol{x}$$

or equivalently, as

$$(\lambda I - A)\boldsymbol{x} = \boldsymbol{0}$$

For $\lambda$ to be an eigenvalue of $A$ this equation must have a nonzero solution for $\boldsymbol{x}$.

**Theorem 23.82: Characteristic Equation**

If $A$ is an $n \times n$ matrix, then $\lambda$ is an eigenvalue of $A$ if and only if it satisfies the equation

$$\det(\lambda I - A) = 0 \tag{23.120}$$

This is called the characteristic equation of $A$.

[DF*] When the $\det(\lambda I - A)$ is expanded, the result is a polynomial $p(\lambda)$ of degree $n$ that is called the characteristic polynomial of $A$.

Since a polynomial of degree $n$ has atmost $n$ distinct roots, it follows that the equation

$$\lambda^n + c_1\lambda^{n-1} + \cdots + c_n = 0 \tag{23.121}$$

has at most $n$ distinct solutions and consequently that an $n \times n$ matrix has at most $n$ distinct eigenvalues. Some of these solutions may be complex numbers, and it is possible for a matrix to have complex eigenvalues, even if that matrix itself has real entries.

**Theorem 23.83: Eigenvalues for Triangular Matrix**

If $A$ is an $n \times n$ triangular matrix (upper triangular, lower triangular, or diagonal), then the eigenvalues of $A$ are the entries on the main diagonal of $A$.

**Theorem 23.84: Eigenvalues and Eigenvectors**

If $A$ is an $n \times n$ matrix, the following statements are equivalent.

(a) $\lambda$ is an eigenvalue of $A$.

(b) The system of equations $(\lambda I - A)\boldsymbol{x} = \boldsymbol{0}$ has nontrivial solutions.

(c) There is nonzero vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \lambda\boldsymbol{x}$.

(d) $\lambda$ is a solution of the characteristic equation $\det(\lambda I - A) = 0$

[DF*] Eigenvectors corresponding to an eigenvalue $\lambda$ of a matrix $A$ are the nonzero vectors that satisfy the equation

$$(\lambda I - A)\boldsymbol{x} = \boldsymbol{0}$$

these eigenvectors are the nonzero vectors in the null space of the matrix $\lambda I - A$. We call this null space the eigenspace of $A$ corresponding to $\lambda$.

The eigenspace of $A$ corresponding to the eigenvalue $\lambda$ is the solution space of the homogeneous system $(\lambda I - A)\boldsymbol{x} = \boldsymbol{0}$.

[DF*] Once the eigenvalues and eigenvectors of a matrix $A$ are found, it is a simple matter to find the eigenvalues and eigenvectors of any positive integer power of $A$.

If $\lambda$ is an eigenvalue of $A$ and $\boldsymbol{x}$ is a corresponding eigenvector, then

$$A^2\boldsymbol{x} = A(A\boldsymbol{x}) = A(\lambda\boldsymbol{x}) = \lambda(A\boldsymbol{x}) = \lambda(\lambda\boldsymbol{x}) = \lambda^2\boldsymbol{x}$$

which shows that $\lambda^2$ is an eigenvalue of $A^2$ and that $\boldsymbol{x}$ is a corresponding eigenvector.

**Theorem 23.85: Eigenvalue for Powers of a Matrix**

If $k$ is a positive integer, $\lambda$ is an eigenvalue of a matrix $A$, and $\boldsymbol{x}$ is a corresponding eigenvector, then $\lambda^k$ is an eigenvalue of $A^k$ and $\boldsymbol{x}$ is a corresponding eigenvector.

**Theorem 23.86: Eigenvalue and Invertibility**

A square matrix $A$ is invertible if and only if $\lambda = 0$ is not an eigenvalue of $A$.

[DF*] It is often useful to decompose a matrix into the product of other matrices. A particularly useful decomposition is the Jordan decomposition.

Suppose that $A$ is a $n \times n$ nonsingular square matrix and has $n$ distinct eigenvalues. Then

$$A = NDN^{-1} \tag{23.122}$$

where $D$ is a diagonal matrix with the distinct eigenvalues on the diagonal, and the columns of $N$ are right eigenvectors of $A$.

$$D = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \vdots & 0 & \dots & \ddots & 0 \\ 0 & 0 & 0 & \dots & \lambda_n \end{bmatrix} \tag{23.123}$$

Equivalently we can express

$$A = N^{-1}DN \tag{23.124}$$

where the diagonal elements of $D$ are the left eigenvalues of $A$ and the rows of $N$ are left eigenvectors of $A$.

[DF*] This theorem of equivalent statements relates all of the major topics that have been studied until this section.

**Theorem 23.87: Equivalent Statements for Square Matrix**

If $A$ is an $n \times n$ matrix, then the following statements are equivalent

(a) $A$ is invertible.

(b) $A\boldsymbol{x} = \mathbf{0}$ has only the trivial solution.

(c) The reduced row echelon form of $A$ is $I_n$.

(d) $A$ is expressible as a product of elementary matrices.

(e) $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for every $n \times 1$ matrix $\boldsymbol{b}$.

(f) $A\boldsymbol{x} = \boldsymbol{b}$ has exactly one solution for every $n \times 1$ matrix $\boldsymbol{b}$.

(g) $\det(A) \neq 0$.

(h) The column vectors of $A$ are linearly independent.

(i) The row vectors of $A$ are linearly independent.

(j) The column vectors of $A$ span $\mathbb{R}^n$.

(k) The row vectors of $A$ span $\mathbb{R}^n$.

(l) The column vectors of $A$ form a basis for $\mathbb{R}^n$.

(m) The row vectors of $A$ form a basis for $\mathbb{R}^n$.

(n) $A$ has rank $n$.

(o) $A$ has nullity 0.

(p) The orthogonal complement of the null space of $A$ is $\mathbb{R}^n$.

(q) The orthogonal complement of the row space of $A$ is $\{\mathbf{0}\}$.

(r) The range of $T_A$ is $\mathbb{R}^n$.

(s) $T_A$ is one-to-one.

(t) $\lambda = 0$ is not an eigenvalue of $A$.

**Computational Guide 23.4: Compute Basis for the Eigenspace Corresponding to an Eigenvalue**

These are the methods / procedure to compute basis for the eigenspace corresponding to $\lambda_i$.

Suppose we have square matrix A with size of $n \times n$, and then we want to compute its basis for the eigenspace.

1. Compute the eigenvalues, with formula

$$\det(\lambda I - A) = 0$$

   we will obtain characteristic equation that will result in polynomial of degree $n$, thus with the help of root finding formula (Bisection, Newton-Raphson) or root quadratic formula we will obtain at most $n$ distint eigenvalues.

2. After we obtain $\lambda_1, \lambda_2, \ldots, \lambda_n$, each value of the eigenvalue if substituted into the equation

$$(\lambda_i I - A)\boldsymbol{x} = \mathbf{0}$$

   can be used to determine the basis for eigenspace corresponding to $\lambda_i$, with $i = 1, 2, \ldots, n$ and $\boldsymbol{x}$ is an eigenvector of $A$ corresponding to $\lambda_i$.

3. Compute from $i = 1, 2, \ldots, n$ the matrix

$$(\lambda_i I - A)\boldsymbol{x} = \mathbf{0}$$

   Solve the system using Gaussian elimination to obtain reduced row echelon form for matrix

$$(\lambda_i I - A)$$

   we will obtain the solution for $\boldsymbol{x}$ in parameter term, for example like this:

$$\begin{aligned} x_1 &= 2s \\ x_2 &= 3t - s \\ x_3 &= t \\ &\vdots \\ x_n &= s \end{aligned}$$

   with $s, t$ are example of parameter.

**Computational Guide 23.5: Compute Basis for the Eigenspace Corresponding to an Eigenvalue**

4. The eigenvectors $\boldsymbol{x}$ of $A$ corresponding to $\lambda_i$ are the nonzero vector of the form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2s \\ 3t - s \\ t \\ \vdots \\ s \end{bmatrix} = \begin{bmatrix} 2s \\ -s \\ 0 \\ \vdots \\ s \end{bmatrix} + \begin{bmatrix} 0 \\ 3t \\ t \\ \vdots \\ 0 \end{bmatrix} = s \begin{bmatrix} 2 \\ -1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} + t \begin{bmatrix} 0 \\ 3 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

thus we obtain that these eigenvectors below

$$\begin{bmatrix} 2 \\ -1 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 3 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

are linearly independent and form a basis for the eigenspace corresponding to $\lambda_i$.

5. Do the computation till $i = n$.

[DF*] **Example:**
Let $A$ be a $2 \times 2$ matrix, and call a line through the origin of $\mathbb{R}^2$ invariant under $A$ if $A\boldsymbol{x}$ lies on the line when $\boldsymbol{x}$ does. Find equations for all lines in $\mathbb{R}^2$, if any, that are invariant under the given matrix

(a)

$$A = \begin{bmatrix} 4 & -1 \\ 2 & 1 \end{bmatrix}$$

(b)

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

(c)

$$A = \begin{bmatrix} 2 & 3 \\ 0 & 2 \end{bmatrix}$$

**Solution:**

(a) If a line is invariant under $A$, then it can be expressed in terms of an eigenvector of $A$.

$$\begin{aligned} \det(\lambda I - A) &= \begin{vmatrix} \lambda - 4 & 1 \\ -2 & \lambda - 1 \end{vmatrix} \\ &= (\lambda - 4)(\lambda - 1) + 2 \\ &= \lambda^2 - 5\lambda + 6 \\ &= (\lambda - 3)(\lambda - 2) \end{aligned}$$

To determine the eigenvector for eigenspace corresponding to eigenvalue $\lambda = 3$ and $\lambda = 2$ we will use this equation

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

$$\begin{bmatrix} \lambda - 4 & 1 \\ -2 & \lambda - 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

For $\lambda = 3$, it gives

$$\begin{bmatrix} -1 & 1 \\ -2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the reduces row echelon form for $(\lambda I - A)$ is

$$\begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}$$

remember that leading 1's has to be 1, not $-1$. So a general solution is

$$y = x$$

The line $y = x$ is invariant under $A$.

For $\lambda = 2$, it gives

$$\begin{bmatrix} -2 & 1 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the reduces row echelon form for $(\lambda I - A)$ is

$$\begin{bmatrix} 1 & -\frac{1}{2} \\ 0 & 0 \end{bmatrix}$$

the general solution is

$$y = 2x$$

the line $y = 2x$ is invariant under $A$.

(b) We will check the eigenvalue for the matrix $A$

$$\det(\lambda I - A) = \begin{vmatrix} \lambda & -1 \\ 1 & \lambda \end{vmatrix} = \lambda^2 + 1$$

Since the characteristic equation

$$\lambda^2 + 1 = 0$$

has no real solutions ($\lambda = \sqrt{-1} = \pm i$), there are no lines that are invariant under $A$.

(c)

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - 2 & -3 \\ 0 & \lambda - 2 \end{vmatrix} = (\lambda - 2)^2$$

The eigenvalues are real and repeated, $\lambda = 2$.

For $\lambda = 2$, it gives

$$\begin{bmatrix} 0 & -3 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the reduces row echelon form for $(\lambda I - A)$ is

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

the general solution is

$$y = 0$$

and the line $y = 0$ is invariant under $A$. The line $y = 0$ is the $x$ axis that went through the origin.

[DF*] **Example:**
Show that if

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then the solutions of the characteristic equation of $A$ are

$$\lambda = \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right]$$

Use this result to show that $A$ has

(a) Two distinct real eigenvalues if $(a-d)^2 + 4bc > 0$

(b) Two repeated real eigenvalues if $(a-d)^2 + 4bc = 0$

(c) Complex conjugate eigenvalues if $(a-d)^2 + 4bc < 0$

**Solution:**
First we will compute

$$\det(\lambda I - A) = 0$$

to determine the characteristic equation

$$\begin{aligned} \det(\lambda I - A) &= 0 \\ (\lambda - a)(\lambda - d) - bc &= 0 \\ \lambda^2 - (a+d)\lambda + ad - bc &= 0 \end{aligned}$$

from the quadratic equation to determine the root we will have

$$\begin{aligned} \lambda_{1,2} &= \frac{(a+d) \pm \sqrt{(a+d)^2 - 4(1)(ad-bc)}}{2} \\ &= \frac{(a+d) \pm \sqrt{(a^2+d^2+2ad) - 4ad + 4bc}}{2} \\ &= \frac{(a+d) \pm \sqrt{(a^2+d^2-2ad) + 4bc}}{2} \\ &= \frac{(a+d) \pm \sqrt{(a-d)^2 + 4bc}}{2} \\ &= \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right] \end{aligned}$$

(a) Now if we let

$$(a-d)^2 + 4bc > 0$$

the eigenvalues will be

$$\lambda_{1,2} = \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right]$$
$$\lambda_{1,2} = \frac{(a+d) \pm C}{2}$$

with $C$ is a positive real number, thus we will have two distinct real eigenvalues.

(b) Now if we let

$$(a-d)^2 + 4bc = 0$$

the eigenvalues will be

$$\lambda_{1,2} = \frac{1}{2}\left[(a+d) \pm \sqrt{0}\right]$$
$$\lambda_{1,2} = \frac{(a+d) \pm 0}{2}$$
$$\lambda_1 = \lambda_2 = \frac{(a+d)}{2}$$

thus we will have two repeated real eigenvalues $\lambda_1 = \lambda_2 = \frac{(a+d)}{2}$.

(c) Now if we let

$$(a-d)^2 + 4bc < 0$$

the eigenvalues will be

$$\lambda_{1,2} = \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right]$$
$$\lambda_{1,2} = \frac{(a+d) \pm iD}{2}$$

with $D$ is a positive real number and $i = \sqrt{-1}$ is a complex number, thus we will have two complex conjugate eigenvalues if $(a-d)^2 + bc < 0$, that is

$$\lambda_1 = \frac{(a+d) + iD}{2}$$
$$\lambda_2 = \frac{(a+d) - iD}{2}$$

[DF*] **Example:**
Let

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Show that if $b \neq 0$, then

$$\boldsymbol{x}_1 = \begin{bmatrix} -b \\ a - \lambda_1 \end{bmatrix}, \quad \boldsymbol{x}_2 = \begin{bmatrix} -b \\ a - \lambda_2 \end{bmatrix}$$

are eigenvectors of $A$ that correspond, respectively, to the eigenvalues

$$\lambda_1 = \frac{1}{2}\left[(a+d) + \sqrt{(a-d)^2 + 4bc}\right]$$
$$\lambda_2 = \frac{1}{2}\left[(a+d) - \sqrt{(a-d)^2 + 4bc}\right]$$

**Solution:**
We know that

$$\boldsymbol{x}_1 = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda_1$ if and only if $\boldsymbol{x}$ is a nontrivial solution of $(\lambda_1 I - A)\boldsymbol{x} = \mathbf{0}$ / the eigenvectors are the nonzero vectors in the null space of the matrix $\lambda_1 I - A$, or in matrix form

$$\begin{bmatrix} \lambda_1 - a & -b \\ -c & \lambda_1 - d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solving this system symbolically using Gaussian elimination yields

$$\begin{bmatrix} \lambda_1 - a & -b \\ 0 & \frac{\lambda_1 - d}{c}(\lambda_1 - a) - b \end{bmatrix}$$

For the second row we will analyze it later on, so we just let the second row be for now.

On the first row we will have

$$(\lambda_1 - a)x_1 - bx_2 = 0$$
$$x_1 = \frac{b}{\lambda_1 - a}x_2$$

now we do the parameterization to obtain the general solution

$$x_2 = s$$
$$x_1 = \frac{b}{\lambda_1 - a}s$$

Thus, the eigenvector of $A$ corresponding to $\lambda_1$ are the nonzero vectors of the form

$$\boldsymbol{x}_1 = \begin{bmatrix} \frac{b}{\lambda_1 - a}s \\ s \end{bmatrix} = s \begin{bmatrix} \frac{b}{\lambda_1 - a} \\ 1 \end{bmatrix}$$

so if we multiply the entries in $\boldsymbol{x}_1$ by $-(\lambda_1 - a)$

$$\boldsymbol{x}_1 = \begin{bmatrix} -b \\ a - \lambda_1 \end{bmatrix}$$

notice that by multiplying a vector, we are only scaling it (make it longer or smaller), and multiplying with $-1$ is only changing the direction to the opposite direction of the vector, the slope stays the same. Thus these two eigenvectors are the same eigenvector

$$\boldsymbol{x}_1 = \begin{bmatrix} -b \\ a - \lambda_1 \end{bmatrix} = \begin{bmatrix} b \\ \lambda_1 - a \end{bmatrix}$$

These vector is a basis for the eigenspace corresponding to $\lambda_1$, in other words $\boldsymbol{x}_1$ is the eigenvector of $A$ corresponding to $\lambda_1$.

Now we will analyze the seond row of

$$\begin{bmatrix} \lambda_1 - a & -b \\ 0 & \frac{\lambda_1 - d}{c}(\lambda_1 - a) - b \end{bmatrix}$$

to prove that $b \neq 0$, we will start with a converse statement.

If $b = 0$, here is what will happens. We will take a look at the second row since we know that eigenvector is the nonzero vector, the second row already stated that $x_1 = 0$ thus

$$x_2 \neq 0$$

so

$$\begin{aligned} \left(\frac{\lambda_1 - d}{c}(\lambda_1 - a) - b\right) x_2 &= 0 \\ \frac{\lambda_1 - d}{c}(\lambda_1 - a) - b &= 0 \\ (\lambda_1 - d)(\lambda_1 - a) &= bc \\ \lambda_1^2 - (a + d)\lambda_1 + ad &= bc \end{aligned}$$

if $b = 0$ we will have

$$\lambda_1^2 - (a + d)\lambda_1 + ad = 0$$

this quadratic characteristic polynomial will have the root / eigenvalues of

$$\begin{aligned} \lambda_1 &= \frac{1}{2}\left[(a + d) + \sqrt{(a + d)^2 - 4(1)(ad)}\right] \\ \lambda_1 &= \frac{1}{2}\left[(a + d) + \sqrt{(a - d)^2}\right] \end{aligned}$$

which is not the eigenvalues that is set at the beginning:

$$\begin{aligned} \lambda_1 &= \frac{1}{2}\left[(a + d) + \sqrt{(a - d)^2 + 4bc}\right] \\ \lambda_2 &= \frac{1}{2}\left[(a + d) - \sqrt{(a - d)^2 + 4bc}\right] \end{aligned}$$

thus, it is a contradiction and $b \neq 0$ to fulfill the condition.
We repeat the same method for $\lambda_2$, we will obtain

$$\boldsymbol{x}_2 = \begin{bmatrix} -b \\ a - \lambda_2 \end{bmatrix} = \begin{bmatrix} b \\ \lambda_2 - a \end{bmatrix}$$

as the eigenvector of $A$ corresponding to $\lambda_2$.

```
A:
[a b]
[c d]

λI - A:
[λ-a  -b ]
[ -c  λ-d]

 λI - A (in reduced row form) :
[        λ-a                  -b          ]
[          0          λ-d-b*c*(λ-a)^(-1)]
```

**Figure 23.111:** *The computation to perform Gaussian elimination for symbolic matrix $\lambda I - A$ with C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Symbolic Gaussian Elimination of 2 X 2 Matrix with SymbolicC++/main.cpp**).*

[DF*] **Example:**
Prove: If $\lambda$ is an eigenvalue of an invertible matrix $A$, and $\boldsymbol{x}$ is a corresponding eigenvector, then $1/\lambda$ is an eigenvalue of $A^{-1}$, and $\boldsymbol{x}$ is a corresponding eigenvector.
**Solution:**
Let

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

be an invertible matrix, then

$$\lambda I - A = \begin{bmatrix} \lambda - a & -b \\ -c & \lambda - d \end{bmatrix}$$

and the determinant of $\lambda I - A$ to obtain the characteristic polynomial is

$$\begin{aligned} \det(\lambda I - A) &= 0 \\ (\lambda - a)(\lambda - d) - bc &= 0 \\ \lambda^2 - (a+d)\lambda + ad - bc &= 0 \end{aligned}$$

we will have the eigenvalue as

$$\lambda = \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right]$$

Now, we will determine the eigenvalue of $A^{-1}$,

$$A^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

then

$$\lambda I - A^{-1} = \begin{bmatrix} \lambda - \frac{d}{ad-bc} & \frac{b}{ad-bc} \\ \frac{c}{ad-bc} & \lambda - \frac{a}{ad-bc} \end{bmatrix}$$

thus

$$\begin{aligned} \det(\lambda I - A^{-1}) &= 0 \\ \left(\lambda - \frac{d}{ad-bc}\right)\left(\lambda - \frac{a}{ad-bc}\right) - \frac{bc}{(ad-bc)^2} &= 0 \\ \lambda^2 - \left(\frac{a}{ad-bc} + \frac{d}{ad-bc}\right)\lambda + \frac{ad}{(ad-bc)^2} - \frac{bc}{(ad-bc)^2} &= 0 \end{aligned}$$

we will have the eigenvalue for $A^{-1}$ as

$$\begin{aligned}
\lambda &= \frac{(\frac{a+d}{ad-bc}) \pm \sqrt{(\frac{a+d}{ad-bc})^2 - 4(1)(\frac{ad-bc}{(ad-bc)^2})}}{2} \\
&= \frac{(\frac{a+d}{ad-bc}) \pm \sqrt{\frac{(a+d)^2}{(ad-bc)^2} - 4(\frac{1}{ad-bc})}}{2} \\
&= \frac{(\frac{a+d}{ad-bc}) \pm \sqrt{\frac{(a+d)^2}{(ad-bc)^2} - \frac{4(ad-bc)}{(ad-bc)^2}}}{2} \\
&= \frac{(\frac{a+d}{ad-bc}) \pm \sqrt{\frac{(a+d)^2-4(ad-bc)}{(ad-bc)^2}}}{2} \\
&= \frac{(\frac{a+d}{ad-bc}) \pm \sqrt{\frac{(a-d)^2+4bc}{(ad-bc)^2}}}{2} \\
&= \frac{1}{2(ad-bc)} \left[(a+d) \pm \sqrt{(a-d)^2+4bc}\right]
\end{aligned}$$

From this we can say that

$$\lambda_{A^{-1}} = \frac{1}{ad-bc}\lambda_A$$

if $\lambda$ is an eigenvalue of an invertible matrix $A$ and $\boldsymbol{x}$ is a corresponding eigenvector, then $\frac{1}{ad-bc}\lambda$ is an eigenvalue of $A^{-1}$ and $\boldsymbol{x}$ is a corresponding eigenvector.

[DF*] **Example:**

Prove: If $\lambda$ is an eigenvalue of an invertible matrix $A$, and $\boldsymbol{x}$ is a corresponding eigenvector, then $\lambda - s$ is an eigenvalue of $A - sI$, and $\boldsymbol{x}$ is a corresponding eigenvector.

**Solution:**

Let

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

be an invertible matrix, then

$$\lambda I - A = \begin{bmatrix} \lambda - a & -b \\ -c & \lambda - d \end{bmatrix}$$

and the determinant of $\lambda I - A$ to obtain the characteristic polynomial is

$$\begin{aligned}
\det(\lambda I - A) &= 0 \\
(\lambda - a)(\lambda - d) - bc &= 0 \\
\lambda^2 - (a+d)\lambda + ad - bc &= 0
\end{aligned}$$

we will have the eigenvalue as

$$\lambda = \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2+4bc}\right]$$

Now, we will determine the eigenvalue of $A - sI$,

$$A - sI = \begin{bmatrix} a-s & b \\ c & d-s \end{bmatrix}$$

then

$$\lambda I - (A - sI) = \begin{bmatrix} \lambda - a + s & -b \\ -c & \lambda - d + s \end{bmatrix}$$

and the determinant of $\lambda I - (A - sI)$ to obtain the characteristic polynomial is

$$\begin{aligned} \det(\lambda I - (A - sI)) &= 0 \\ (\lambda - (a-s))(\lambda - (d-s)) - bc &= 0 \\ \lambda^2 - ((a-s) + (d-s))\lambda + (a-s)(d-s) - bc &= 0 \end{aligned}$$

we will have the eigenvalue for $A - sI$ as

$$\begin{aligned} \lambda &= \frac{1}{2}\left[(a-s) + (d-s) \pm \sqrt{((a-s)+(d-s))^2 - 4(1)((a-s)(d-s) - bc)}\right] \\ &= \frac{1}{2}\left[(a+d) - (2s) \pm \sqrt{(a+d-2s)^2 - 4((ad - as - ds + s^2) - bc)}\right] \\ &= \frac{1}{2}\left[(a+d) - (2s) \pm \sqrt{(a^2 + 2ad - 4as - 4ds + d^2 + 4s^2) - 4ad + 4as + 4ds - 4s^2 + 4bc)}\right] \\ &= \frac{1}{2}\left[(a+d) - (2s) \pm \sqrt{a^2 - 2ad + d^2 + 4bc}\right] \\ &= \frac{1}{2}\left[(a+d) \pm \sqrt{(a-d)^2 + 4bc}\right] - s \end{aligned}$$

so it is proven that

$$\lambda_{A-sI} = \lambda_A - s$$

[DF*] **Example:**

(a) Prove that if $A$ is a square matrix, then $A$ and $A^T$ have the same eigenvalues.

(b) Show that $A$ and $A^T$ need not have the same eigenspaces.

**Solution:**

(a) If $A$ is a square matrix of size $n$, then

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

and

$$A^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{bmatrix}$$

transposing a square matrix won't change the diagonal entries. Thus

$$\det(\lambda I - A) = \det(\lambda I - A^T)$$

If $\det(\lambda I - A) = \det(\lambda I - A^T)$ then the characteristic polynomial of $A$ and $A^T$ will be the same and hence it will make the eigenvalues of $A$ the same as the eigenvalues of $A^T$.

(b) We refresh our memory on the definition:
Eigenspace is the null space of the matrix $\lambda I - A$ corresponding to its eigenvalue $\lambda$.

Eigenspace is the null space of the matrix $\lambda I - A^T$ corresponding to its eigenvalue $\lambda$.

Let

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

The characteristic equation for $A$ is

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - 1 & -2 \\ -3 & \lambda - 4 \end{vmatrix} = (\lambda - 1)(\lambda - 4) - 6$$

The characteristic equation for $A^T$ is

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - 1 & -3 \\ -2 & \lambda - 4 \end{vmatrix} = (\lambda - 1)(\lambda - 4) - 6$$

Both matrix $A$ and $A^T$ will share the same eigenvalue, since the characteristic polynomial is the same one, even if the entries of matrix $A$ are all different.

Now to examine the eigenspace for $A$, let $\lambda$ be the eigenvalue for $A$ and $A^T$, by definition

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of $(\lambda I - A)\boldsymbol{x} = \mathbf{0}$, or in matrix form

$$\begin{bmatrix} \lambda - 1 & -2 \\ -3 & \lambda - 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solving the system above with Gaussian elimination yields

$$x_1 = \frac{2s}{\lambda - 1}, \quad x_2 = s$$

thus, the eigenvector of $A$ corresponding to $\lambda$ are the nonzero vectors of the form

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{2s}{\lambda-1} \\ s \end{bmatrix} = s \begin{bmatrix} \frac{2}{\lambda-1} \\ 1 \end{bmatrix}$$

the vector

$$\begin{bmatrix} \frac{2}{\lambda-1} \\ 1 \end{bmatrix}$$

form a basis for the eigenspace of $A$ corresponding to $\lambda$.

Now to examine the eigenspace for $A^T$, let $\lambda$ be the eigenvalue for $A$ and $A^T$, by definition

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

is an eigenvector of $A^T$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of $(\lambda I - A^T)\boldsymbol{x} = \mathbf{0}$, or in matrix form

$$\begin{bmatrix} \lambda - 1 & -3 \\ -2 & \lambda - 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solving the system above with Gaussian elimination yields

$$x_1 = \frac{3s}{\lambda - 1}, \quad x_2 = s$$

thus, the eigenvector of $A^T$ corresponding to $\lambda$ are the nonzero vectors of the form

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{3s}{\lambda-1} \\ s \end{bmatrix} = s \begin{bmatrix} \frac{3}{\lambda-1} \\ 1 \end{bmatrix}$$

the vector

$$\begin{bmatrix} \frac{3}{\lambda-1} \\ 1 \end{bmatrix}$$

form a basis for the eigenspace of $A^T$ corresponding to $\lambda$.

Now we have proven that both $A$ and $A^T$ have different eigenspace even if they are sharing the same eigenvalue $\lambda$.

[DF*] **Example:**
The eigenvectors that we have been studying are sometimes called right eigenvectors to distinguish them from left eigenvectors, which are $n \times 1$ column matrices $\boldsymbol{x}$ that satisfy the equation $\boldsymbol{x}^T A = \mu \boldsymbol{x}^T$ for some scalar $\mu$. What is the relationship, if any, between the right eigenvectors and corresponding eigenvalues $\lambda$ of $A$ and the left eigenvectors and corresponding eigenvalues $\mu$ of $A$.
**Solution:**
We will derivate the equation $\boldsymbol{x}^T A = \mu \boldsymbol{x}^T$ to obtain the left eigenvectors formula, first

$$\begin{aligned} \boldsymbol{x}^T A &= \mu \boldsymbol{x}^T \\ \boldsymbol{x}^T A &= \boldsymbol{x}^T \mu \\ \boldsymbol{x}^T \mu I - \boldsymbol{x}^T A &= \mathbf{0} \\ \boldsymbol{x}^T (\mu I - A) &= \mathbf{0} \end{aligned}$$

the equation above must have a nonzero solution for $\boldsymbol{x}^T$. This is so if and only if the coefficient matrix $\mu I - A$ has a zero determinant, or in equation term it satisfies the equation

$$\det(\mu I - A) = 0$$

Based on the characteristic equation above, we can tell that $\mu = \lambda$ since for right eigenvectors we will have the characteristic equation of $A$ as

$$\det(\lambda I - A) = 0$$

Now in determining the left eigenvectors $\boldsymbol{x}^T$, we will have

$$\begin{aligned}
\boldsymbol{x}^T(\mu I - A) &= \mathbf{0} \\
\boldsymbol{x}^T(\lambda I - A) &= \mathbf{0} \\
\boldsymbol{x}^T(\lambda I - A) &= (\lambda I - A^T)\boldsymbol{x}
\end{aligned}$$

In conclusion:
The left eigenvectors of $A$ is the right eigenvectors of $A^T$.
The left eigenvectors that satisfy the equation $\boldsymbol{x}^T A = \mu \boldsymbol{x}^T$ is connected with the eigenvalue of the right eigenvectors, as $\lambda = \mu$. Thus, $\boldsymbol{x}^T A = \mu \boldsymbol{x}^T = \lambda \boldsymbol{x}^T$.

[DF*] A basis for $\mathbb{R}^n$ that consists of eigenvectors of an $n \times n$ matrix $A$ can be used to study geometric properties of $A$ and to simplify various numerical computations.

[DF*] The matrix product

$$P^{-1}AP$$

is called a similarity transformation of the matrix $A$. Such products are important in the study of eigenvectors and eigenvalues.

**Definition 23.46: Similarity**

If $A$ and $B$ are square matrices, then we say that $B$ is similar to $A$ if there is an invertible matrix $P$ such that

$$B = P^{-1}AP$$

[DF*] Similar matrices have many properties in common. For example, if $B = P^{-1}AP$, then it follows that $A$ and $B$ have the same determinant, since

$$\begin{aligned}
\det(B) &= \det(P^{-1}AP) \\
&= \det(P^{-1})\det(A)\det(P) \\
&= \frac{1}{\det(P)}\det(A)\det(P) \\
&= \det(A)
\end{aligned}$$

In general, any property that is shared by all similar matrices is called a similarity invariant. The list of similarity invariants:

(a) $A$ and $P^{-1}AP$ have the same determinant.
(b) $A$ is invertible if and only if $P^{-1}AP$ is invertible.
(c) $A$ and $P^{-1}AP$ have the same rank.
(d) $A$ and $P^{-1}AP$ have the same nullity.
(e) $A$ and $P^{-1}AP$ have the same trace.
(f) $A$ and $P^{-1}AP$ have the same characteristic polynomial.
(g) $A$ and $P^{-1}AP$ have the same eigenvalues.

(h) If $\lambda$ is an eigenvalue of $A$ and hence of $P^{-1}AP$, then the eigenspace of $A$ corresponding to $\lambda$ and the eigenspace of $P^{-1}AP$ corresponding to $\lambda$ have the same dimension.

**Definition 23.47: Diagonalizable**

A square matrix $A$ is said to be diagonalizable if it is similar to some diagonal matrix; that is, if there exists an invertible matrix $P$ such that

$$P^{-1}AP$$

is diagonal. In this case the matrix $P$ is said to diagonalize $A$.

**Theorem 23.88: Diagonalizable and Eigenvectors**

If $A$ is an $n \times n$ matrix, the following statements are equivalent.

(a) $A$ is diagonalizable.

(b) $A$ has $n$ linearly independent eigenvectors.

[DF*] Matrix $A$ cannot always be diagonalized.

After we obtain the eigenvalues for matrix $A$ with size of $n \times n$, we compute these matrices

$$\begin{aligned} A_1 &= \lambda_1 I - A \\ A_2 &= \lambda_2 I - A \\ &\vdots \\ A_n &= \lambda_n I - A \end{aligned}$$

We know that the eigenvectors corresponding to an eigenvalue $\lambda_i$ of a matrix $A$ are the nonzero vectors that satisfy the equation

$$(\lambda_i I - A)\boldsymbol{x} = \mathbf{0}$$

these eigenvectors are the nonzero vectors in the null space of the matrix $\lambda_i I - A$. With $i, j = 1, 2, \ldots, n$ and $i \neq j$, thus $\lambda_i \neq \lambda_j$, for repeated root / repeated eigenvalue it will share the same set of eigenvectors.

But, instead of computing the eigenvectors for each matrix $A_1, A_2, \ldots, A_n$ we can just compute the rank of each matrix above, then we can obtain the nullity (the dimension of the null space) and find the sum of all nullity.

$$n = \text{rank}(A) + \text{nullity}(A)$$

If

$$\sum_{i=1}^{n} \text{nullity}(A_i) \neq n$$

then there is no matrix $P$ that diagonalizes $A$.

**Theorem 23.89: Linearly Independent Eigenvectors**

If $v_1, v_2, \ldots, v_k$ are eigenvectors of a matrix $A$ corresponding to distinct eigenvalues, then $\{v_1, v_2, \ldots, v_k\}$ is a linearly independent set.

**Theorem 23.90: Distinct Eigenvalues**

If an $n \times n$ matrix $A$ has $n$ distinct eigenvalues, then $A$ is diagonalizable.

[DF*] We can apply diagonalization to compute high powers of a square matrix $A$.

If $A$ happens to be diagonalizable, then the computations can be simplified by diagonalizing $A$.

Suppose that $A$ is a diagonalizable $n \times n$ matrix, that $P$ diagonalizes $A$, and that

$$P^{-1}AP = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} = D$$

Squaring both sides of this equation yields

$$(P^{-1}AP)^2 = \begin{bmatrix} \lambda_1^2 & 0 & \ldots & 0 \\ 0 & \lambda_2^2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n^2 \end{bmatrix} = D^2$$

We can rewrite the left side of this equation as

$$(P^{-1}AP)^2 = P^{-1}APP^{-1}AP = P^{-1}AIAP = P^{-1}A^2P$$

from which we obtain the relationship

$$P^{-1}A^2P = D^2$$

More generally, if $k$ is a positive integer, then a similar computation will show that

$$P^{-1}A^kP = D^k = \begin{bmatrix} \lambda_1^k & 0 & \ldots & 0 \\ 0 & \lambda_2^k & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n^k \end{bmatrix}$$

which we can rewrite as

$$A^k = PD^kP^{-1} = P \begin{bmatrix} \lambda_1^k & 0 & \ldots & 0 \\ 0 & \lambda_2^k & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \lambda_n^k \end{bmatrix} P^{-1} \tag{23.125}$$

Computing the right side of this formula involves only three matrix multiplication and the powers of the diagonal entries of $D$. For matrices of large size and high powers of $\lambda$, this involves substantially fewer operations than computing $A^k$ directly.

**Theorem 23.91: Eigenvalues of Powers of a Matrix**

If $\lambda$ is an eigenvalue of a square matrix $A$ and $\boldsymbol{x}$ is a corresponding eigenvector, and if $k$ is any positive integer, then $\lambda^k$ is an eigenvalue of $A^k$ and $\boldsymbol{x}$ is a corresponding eigenvector.

[DF*] If we have a matrix of size $3 \times 3$ with characteristic polynomial of

$$(\lambda - 1)(\lambda - 2)^2$$

thus, the eigenspace corresponding to $\lambda = 1$ is at most one-dimensional, and the eigenspace corresponding to $\lambda = 2$ is at most two-dimensional.

If $\lambda_0$ is an eigenvalue of an $n \times n$ matrix $A$, then the dimension of the eigenspace corresponding to $\lambda_0$ is called the geometric multiplicity of $\lambda_0$, and the number of times that $\lambda - \lambda_0$ appears as a factor in the characteristic polynomial of $A$ is called the algebraic multiplicity of $\lambda_0$.

**Theorem 23.92: Geometric and Algebraic Multiplicity**

If $A$ is a square matrix, then:

(a) For every eigenvalue of $A$, the geometric multiplicity is less than or equal to the algebraic multiplicity.

(b) $A$ is diagonalizable if and only if the geometric multiplicity of every eigenvalue is equal to the algebraic multiplicity.

[DF*] **Example:**
Suppose that the characteristic polynomial of some matrix $A$ is found to be $p(\lambda) = (\lambda - 1)(\lambda - 3)^2(\lambda - 4)^3$.

(a) What can you say about the dimensions of the eigenspaces of $A$?

(b) What can you say about the dimensions of the eigenspaces if you know that $A$ is diagonalizable?

(c) If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is a linearly independent set of eigenvectors of $A$ all of which correspond to the same eigenvalue of $A$, what can you say about the eigenvalue?

**Solution:**

(a) The dimensions of the eigenspaces of $A$ will be less than or equal than the sum of all geometric multiplicities for each distinct eigenvalue which is less than or equal to 6.

(b) If $A$ is diagonalizable then the dimensions of the eigenspaces of $A$ will be 6.

(c) If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is a linearly independent set of eigenvectors of $A$ then it is correspond to the eigenvalue of $\lambda = 4$, the only eigenvalue that has algebraic multiplicity of 3 and able to have the dimensions of the eigenspace / the geometric multiplicity of 3.

[DF*] **Example:**

This problem will lead you through a proof of the fact that the algebraic multiplicity of an eigenvalue of an $n \times n$ matrix $A$ is greater than or equal to the geometric multiplicity. For this purpose, assume that $\lambda_0$ is an eigenvalue with geometric multiplicity $k$.

(a) Prove that there is a basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ for $\mathbb{R}^n$ in which the first $k$ vectors of $B$ form a basis for the eigenspace corresponding to $\lambda_0$.

(b) Let $P$ be the matrix having the vectors in $B$ as columns. Prove that the product $AP$ can be expressed as

$$AP = P \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

Compare the first $k$ columns vectors on both sides.

(c) Use the result in part (b) to prove that $A$ is similar to

$$C = \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

and hence that $A$ and $C$ have the same characteristic polynomial.

(d) By considering $\det(\lambda I - C)$, prove that the characteristic polynomial of $C$ (and hence $A$) contains the factor $(\lambda - \lambda_0)$ at least $k$ times, thereby proving that the algebraic multiplicity of $\lambda_0$ is greater than or equal to the geometric multiplicity $k$.

**Solution:**

(a) To prove that there is a basis $B$ that is consisting of $n$ linearly independent eigenvectors it means that $A$ is diagonalizable by theorem.

By assuming that $\lambda_0$ is an eigenvalue with geometric multiplicity $k$, it means that there are $k$ linearly independent eigenvectors correspond to $\lambda_0$, which we can say they are

$$\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k\}$$

By theorem, the geometric multiplicity is less than or equal to the algebraic multiplicity.

By theorem we know that $A$ is diagonalizable if and only if the geometric multiplicity of every eigenvalue is equal to the algebraic multiplicity. So, the characteristic polynomial of $A$ can be represented as

$$p(\lambda) = (\lambda - \lambda_0)^k (\lambda - \lambda_1)^{r_1} (\lambda - \lambda_2)^{r_2} \ldots (\lambda - \lambda_{n-1})^{r_{n-1}}$$

with $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ are eigenvalues of $A$ and $k, r_1, r_2, \ldots, r_n$ are integers that each is less or equal to $n$ thus $k + r_1 + r_2 + \cdots + r_{n-1} = n$.

Generally, in order to obtain the eigenvectors of $A$ corresponding to each eigenvalues we need to solve these systems below:

$$\begin{aligned} (\lambda_0 I - A)\boldsymbol{x} &= \boldsymbol{0} \\ (\lambda_1 I - A)\boldsymbol{x} &= \boldsymbol{0} \\ &\vdots \\ (\lambda_{n-1} I - A)\boldsymbol{x} &= \boldsymbol{0} \end{aligned}$$

by solving each of the system above using Gaussian elimination we will obtain the eigenvectors corresponding to $\lambda_i$ as the nonzero vectors in the null space of the matrix $\lambda_i I - A$ with $i = 0, 1, 2 \ldots, n$.

Matrix $P$ that diagonalizes matrix $A$ can be represented as the basis vectors / eigenvectors of $A$ with no preferred order.

$$\begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_n \end{bmatrix}$$

matrix $P$ is filled with basis $B$.

Now we will focus on the particular eigenvalue $\lambda_0$. If there are $k+1$ linearly independent eigenvectors corresponding to $\lambda_0$ then

$$(k+1) + r_1 + r_2 + \cdots + r_{n-1} \neq n$$

this is a contradiction, thus there will only be $k$ eigenvectors corresponding to $\lambda_0$ that will make the first $k$ vectors of $B$.

(b) To prove that the product $AP$ can be expressed as

$$AP = P \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

we multiply both sides with $P^{-1}$

$$P^{-1}AP = P^{-1}P \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

with $P^{-1}AP$ is a diagonal matrix, the left hand side and right hand side will share the same size of a matrix which is of size $n \times n$

$$P^{-1}AP = I_n \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

From this we can see that the diagonal matrix will contain the first $k$ diagonal, the eigenvalue $\lambda_0$, which is correct by definition and by theorem, and the rest of the diagonal will have entries of the rest of the eigenvalues.

Remember that the columns for matrix $P$ is filled with eigenvectors of matrix $A$ and the diagonal matrix $P^{-1}AP$ will have the $i$-th diagonal entry an eigenvalue for the $i$-th column vector of $P$.

Since we set the first $k$ eigenvectors corresponding to $\lambda_0$ as basis for $B$ thus the first $k$ column vectors of $P$ are eigenvectors corresponding to $\lambda_0$.

(c) It has been proven that since $A$ is diagonalizable, thus it is similar to

$$C = P^{-1}AP = I_n \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix} = \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

Since $A$ and $C$ is similar thus they have the same eigenvalues (by similarity invatiants) hence the characteristic polynomial for $A$ and $C$ will be the same.

(d) Consider this

$$\det(\lambda I - C) = \begin{vmatrix} \lambda - \lambda_0 I_k & X \\ 0 & \lambda - Y \end{vmatrix} = \begin{vmatrix} \lambda - \lambda_0 I_k & 0 \\ 0 & \lambda - Y \end{vmatrix}$$

$Y$ are eigenvalues of $A$ excluding $\lambda_0$, and $X$ should have values of 0 since the diagonal matrix will have zeroes in entries non other than the diagonal.

For matrix $A$ we can say that $X$ are entries with nonzero values

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - \lambda_0 I_k & X \\ 0 & \lambda - Y \end{vmatrix}, \quad X \neq 0$$

Even if the entries are different, due to diagonalization and similar invariants, $A$ and $C$ are similar thus the eigenvalues are the same, the eigenspace are the same as well. On $\det(\lambda I - C)$ we observe that $\lambda - \lambda_0$ will appear $k$ times at the diagonal entries. Thus when we have diagonal matrix and want to compute the determinant we will obtain the characteristic polynomial for $C$ as

$$p(\lambda) = (\lambda - \lambda_0)^k (\lambda - \lambda_1)^{r_1} (\lambda - \lambda_2)^{r_2} \dots (\lambda - \lambda_{n-1})^{r_{n-1}}$$

The factor for $(\lambda - \lambda_0)$ is $k$, this is the algebraic multiplicity of $\lambda_0$. When we compute from $\det(\lambda I - A)$ and $\det(\lambda I - C)$ will result in the same algebraic multiplicity for $\lambda_0$, that is $k$.

Since $A$ and $C$ are square matrix of size $n$, we will have

$$k + r_1 + r_2 + \cdots + r_{n-1} = n$$

**For** $\det(\lambda I - C)$ **case**
To compute the geometric multiplicity we will examine the nontrivial solution of

$$(\lambda_0 I - C)\boldsymbol{x} = \mathbf{0}$$

To obtain the eigenvectors corresponding to $\lambda_0$ we will perform Gaussian elimination toward matrix $\lambda_0 I - C$, notice that the first $k$ rows of $\lambda_0 I - C$ will be zeroes, while the rows below cannot be reduced anymore, thus

$$\text{nullity}(\lambda_0 I - C) = k$$

This is the geometric multiplicity of $\lambda_0$, which is $k$.

**For** $\det(\lambda I - A)$ **case**
To compute the geometric multiplicity we will examine the nontrivial solution of

$$(\lambda_0 I - A)\boldsymbol{x} = \mathbf{0}$$

When we compute $\lambda_0 I - A$ with $X \neq 0$ it means that we will have less null space dimension / less row of zeroes, thus

$$\text{nullity}(\lambda_0 I - C) < k$$

This is the geometric multiplicity of $\lambda_0$, which is less than $k$.

So we can say that the algebraic multiplicity of $\lambda_0$ is greater than or equal to the geometric multiplicity.

## i. C++ Computation: Eigenvalues, Eigenvectors, and Characteristic Polynomial of a $3 \times 3$ Matrix

Suppose we have a matrix

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 4 & -17 & 8 \end{bmatrix}$$

We want to compute the eigenvalues and eigenvectors of matrix $A$.

We will use the characteristic equation of $A$.

$$\det(\lambda I - A) = 0$$

$$\det(\lambda I - A) = \det \begin{bmatrix} \lambda & -1 & 0 \\ 0 & \lambda & -1 \\ -4 & 17 & \lambda - 8 \end{bmatrix} = \lambda^3 - 8\lambda^2 + 17\lambda - 4$$

The eigenvalues of $A$ must therefore satisfy the cubic equation

$$\lambda^3 - 8\lambda^2 + 17\lambda - 4 = 0$$

Solving the equation above to compute all values of $\lambda$ is not an easy task. Most of the solutions may be complex numbers.

To solve the cubic equation above, we will begin by searching for integer solutions. This task can be simplified by exploiting the fact that all integer solutions (if there are any) of a polynomial equation with integer coefficients

$$\lambda^n + c_1\lambda^{n-1} + \cdots + c_n = 0$$

must be divisors of the constant term $c_n$. Thus, the only possible integer solutions of the cubic equation above are the divisors of $-4$, that is $\pm 1, \pm 2, \pm 4$. Successively substituting these values in the equation shows that $\lambda = 4$ is an integer solution.

As a consequence, $\lambda - 4$ must be a factor of the left side of the equation. Dividing $\lambda - 4$ into $\lambda^3 - 8\lambda^2 + 17\lambda - 4$ shows that

$$(\lambda - 4)(\lambda^2 - 4\lambda + 1) = 0$$

We know the first eigenvalue is $\lambda_1 = 4$. The rest can be found by using quadratic root equations for $\lambda^2 - 4\lambda + 1) = 0$.

$$\begin{aligned} \lambda_{2,3} &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{-(-4) \pm \sqrt{(-4)^2 - 4(1)(1)}}{2(1)} \\ &= \frac{4 \pm \sqrt{12}}{2} \\ \lambda_{2,3} &= 2 \pm \sqrt{3} \end{aligned}$$

thus the second and third eigenvalues are

$$\lambda_2 = 2 + \sqrt{3}$$

$$\lambda_3 = 2 - \sqrt{3}$$

To determine the eigenvectors, we will start by the definition of eigenvector.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

For the C++ code we are going to use **GiNaC** to compute characteristic polynomial only, then use a nice library (headers only) called **Eigen**, it is a very useful library to do Numerical Linear Algebra computation around the eigen world.

The basic code are obtained from:
**http://yang.amp.i.kyoto-u.ac.jp/ yyama/computer/FAQ/eigen/eigensystem.html**
**https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html**
On GFreya OS and Ubuntu 14.04, the header file should be included as **#include <eigen3/Eigen/Eigenvalues>**, but "eigen3" may be skipped in other platforms, thus you can use this instead **#include <Eigen/Eigenvalues>**.

```
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 5; // define maximum decimal digits
       symbol l("l");

       matrix A = { { 0, 1, 0 }, { 0, 0, 1}, { 4, -17, 8 } };
       matrix I = { { l, 0, 0 }, { 0, l, 0}, { 0, 0, l } };
       cout << "\n A = " << A << endl;
       cout << "\n I = " << I << endl;

       matrix result = I.sub(A);

       cout << endl;

       cout << "lI-A = " << result <<endl;
       cout << endl;
       cout << "Characteristic Polynomial : det(lI - A) = " << determinant
           (result) << endl;
       cout << endl;

       return 0;
```

```
}
```

**C++ Code 171:** *main.cpp "Compute characteristic polynomial"*

To compile it, type:
**g++ main.cpp -o main -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
A = [[0,1,0],[0,0,1],[4,-17,8]]
I = [[λ,0,0],[0,λ,0],[0,0,λ]]
λI-A = [[λ,-1,0],[0,λ,-1],[-4,17,-8+λ]]
Characteristic Polynomial : det(λI - A) = -4+17*λ-8*λ^2+λ^3
```

**Figure 23.112:** *The computation to find characteristic polynomial of a* $3 \times 3$ *matrix with GiNaC library. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Characteristic Polynomial of 3 X 3 Matrix with GiNaC/main.cpp**).*

Now for the C++ code to compute the eigenvalues and eigenvectors with **Eigen**.

```
#include <eigen3/Eigen/Eigenvalues> // header file
#include <iostream>

using namespace std;

int main()
{
        Eigen::Matrix<double, 3, 3> A; // declare a real (double) 3x3 matrix
        // defined the matrix A
        A << 0,1,0,
        0,0,1,
        4,-17,8;
        /*
        A(0,0) = 0.0;
        A(0,1) = 1.0;
        A(0,2) = 0.0;

        A(1,0) = 0.0;
        A(1,1) = 0.0;
        A(1,2) = 1.0;

        A(2,0) = 4.0;
        A(2,1) = -17.0;
        A(2,2) = 8.0;
```

```
        */

        Eigen::EigenSolver<Eigen::Matrix<double, 3,3> > s(A); // the
            instance s(A) includes the eigensystem
        cout << "Matrix A:\n" << A << endl;
        cout << endl;
        cout << "Eigenvalues:" << endl;

        int n = 3;
        for (int i = 1; i <= n; i++)
        {
               cout << "l_" << i << " = " << real(s.eigenvalues()(i-1)) <<
                   " + " << imag(s.eigenvalues()(i-1)) << "i" << endl;
        }

        //cout << s.eigenvalues() << endl;

        cout << endl;
        cout << "Eigenvectors:" << endl;

        //cout << s.eigenvectors() << endl;

        for (int i = 1; i <= n; i++)
        {
               cout << "x[l_" << i << "] = (" << real(s.eigenvectors()(0,i
                   -1)) << " + " << imag(s.eigenvectors()(0,i-1)) << "i" ;
               cout << ", " << real(s.eigenvectors()(1,i-1)) << " + " <<
                   imag(s.eigenvectors()(1,i-1)) << "i" ;
               cout << ", " << real(s.eigenvectors()(2,i-1)) << " + " <<
                   imag(s.eigenvectors()(2,i-1)) << "i )" << endl;
               cout << endl;
        }

        return(0);
    }
```

**C++ Code 172:** *main.cpp "Compute eigenvalues and eigenvectors"*

To compile it, type:
**g++ main.cpp -o main**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We show the eigenvalues and eigenvectors in real and imaginary terms, in this case, the

eigenvalues and eigenvectors are all real numbers, so the imaginary parts are all 0. Remember that complex numbers can be written as

$$a + bi$$

with $a$ and $b$ are real numbers / constant, and $i = \sqrt{-1}$. You can read and comprehend more about complex numbers at the chapter dedicated for this, **Complex Numbers**, it is few chapters after this chapter.

```
cout << "Eigenvalues:" << endl;

int n = 3;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " << real(s.eigenvalues()(i-1))
            << " + " << imag(s.eigenvalues()(i-1)) << "i" <<
           endl;
}

cout << endl;
cout << "Eigenvectors:" << endl;

for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] = (" << real(s.eigenvectors()
           (0,i-1)) << " + " << imag(s.eigenvectors()(0,i-1))
           << "i" ;
       cout << ", " << real(s.eigenvectors()(1,i-1)) << " + "
           << imag(s.eigenvectors()(1,i-1)) << "i" ;
       cout << ", " << real(s.eigenvectors()(2,i-1)) << " + "
           << imag(s.eigenvectors()(2,i-1)) << "i )" << endl;
       cout << endl;
}
```

```
Matrix A:
  0   1   0
  0   0   1
  4 -17   8

Eigenvalues:
λ_1 = 0.267949 + 0i
λ_2 = 3.73205 + 0i
λ_3 = 4 + 0i

Eigenvectors:
x[λ_1] = (-0.963611 + 0i, -0.258199 + 0i, -0.0691842 + 0i )

x[λ_2] = (-0.0691842 + 0i, -0.258199 + 0i, -0.963611 + 0i )

x[λ_3] = (-0.0605228 + 0i, -0.242091 + 0i, -0.968364 + 0i )
```

**Figure 23.113:** *The computation to find eigenvalues and eigenvectors of a* $3 \times 3$ *matrix with Eigen library. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Eigenvectors with Eigen/main.cpp**).*

## ii. C++ Computation: Eigenvectors and Bases for Eigenspaces

In the previous section we are computing eigenvalues and eigenvectors with **Eigen**, now we are going to compute eigenvalues, eigenvectors, and the eigenspaces with **Armadillo** library and then compare the result with code that use **Eigen**.

Suppose we have a $3 \times 3$ matrix

$$A = \begin{bmatrix} 0 & 0 & -2 \\ 1 & 2 & 1 \\ 1 & 0 & 3 \end{bmatrix}$$

we want to find the bases for the eigenspaces of matrix $A$ above. In other words, the eigenvectors of $A$ are the bases for the eigenspaces of matrix $A$ corresponding to the eigenvalue $\lambda$. The eigenspace of $A$ is the solution space of the homogeneous system

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

The characteristic equation of $A$ is

$$\lambda^3 - 5\lambda^2 + 8\lambda - 4 = 0$$

or in factored form

$$(\lambda - 1)(\lambda - 2)^2 = 0$$

Thus, the eigenvalues are

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 2 \\ \lambda_3 &= 2 \end{aligned}$$

By definition,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of $(\lambda I - A)\boldsymbol{x} = \mathbf{0}$, or in matrix form

$$\begin{bmatrix} \lambda & 0 & 2 \\ -1 & \lambda - 2 & -1 \\ -1 & 0 & \lambda - 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

In the case where $\lambda = 2$

$$\begin{bmatrix} 2 & 0 & 2 \\ -1 & 0 & -1 \\ -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this system using Gaussian elimination yields

$$x_1 = -s, \quad x_2 = t, \quad x_3 = s$$

Thus, the eigenvectors of $A$ corresponding to $\lambda = 2$ are the nonzero vectors of the form

$$\boldsymbol{x} = \begin{bmatrix} -s \\ t \\ s \end{bmatrix} = \begin{bmatrix} -s \\ 0 \\ s \end{bmatrix} + \begin{bmatrix} 0 \\ t \\ 0 \end{bmatrix} = s \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} + t \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Since

$$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

are linearly independent, these vectors form a basis for the eigenspace corresponding to $\lambda = 2$.

For the case where $\lambda = 1$, we will have

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & -1 & -1 \\ -1 & 0 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this system yields

$$x_1 = -2s, \quad x_2 = s, \quad x_3 = s$$

Thus, the eigenvectors corresponding to $\lambda = 1$ are the nonzero vectors of the form

$$\begin{bmatrix} -2s \\ s \\ s \end{bmatrix} = s \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

so that

$$\begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 1$.

When we use **Armadillo** library, we know how fun it is, we can just create a textfile **matrixA.txt** as the input for the matrix, this automatic method is really useful and handy.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        cx_mat eigvec;
        cx_vec eigval;
```

```
        mat I(3,3,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 3;
        for (int i = 1; i <= n; i++)
        {
               cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                   imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        cout <<"Matrix l_1I - A:" << "\n" << I*eigval[0] - A <<endl;
        cout <<"Matrix l_2I - A:" << "\n" << I*eigval[1] - A <<endl;
        cout <<"Matrix l_3I - A:" << "\n" << I*eigval[2] - A <<endl;
        mat A1 = I*real(eigval[0]) - A;
        mat A2 = I*real(eigval[1]) - A;
        mat A3 = I*real(eigval[2]) - A;

        cout << endl;
        cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
        cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
        cout << "Eigenspaces for l_3: " << n - arma::rank(A3) << endl;
        cout << endl;

        cout << "Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
               cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                   endl;
        }

        // show eigenvectors in matrix form
        // cout <<"Eigenvectors:" << "\n" << eigvec <<endl;
        // cout <<"Eigenvectors:" << "\n" << real(eigvec) <<endl;
        return 0;
    }
```

**C++ Code 173:** *main.cpp "Eigenvectors or bases for eigenspaces"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- To compute the dimension for the null space remember the formula is

$$\text{nullity}(A) = m - \text{rank}(A)$$

with $m$ is the number of column of matrix $A$.

Thus, for the code, we need to compute the matrix $\lambda I - A$ for every eigenvalues $\lambda_1, \lambda_2$, and $\lambda_3$, then for each corresponding matrix $\lambda I - A$ we will obtain the rank and nullity for the matrix, the dimension of the null space / nullity is the number of bases for the eigenspaces for the corresponding $\lambda$. Watch out that $\lambda_1 = \lambda_3 = 2$ and they both represent only one eigenvalue $\lambda = 2$ with the corresponding eigenvectors:

$$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The code produces eigenvector for $\lambda_3$ as

$$\begin{bmatrix} 0.7071 \\ 0 \\ -0.7071 \end{bmatrix}$$

this is representing the same eigenvector with different length or toward opposite direction, and basically it is the same, if you divide all entries with $-0.7071$ you will obtain

$$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

since we obtain eigenvectors by taking out the parameter of $s$ or $t$, therefore the length itself can varies, and the direction can be toward the opposite or toward the same direction. This is the eigenvector for $\lambda_3$ in parameter term

$$\boldsymbol{x}[\lambda_3] = \begin{bmatrix} -s \\ 0 \\ s \end{bmatrix} = s \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

We take a deep look, we can see that the eigenvector will still the same line, just give random input for $s$, and we realize that it can have different length or facing the opposite or the same direction.

```
cout <<"Matrix l_1I - A:" << "\n" << I*eigval[0] - A <<endl;
cout <<"Matrix l_2I - A:" << "\n" << I*eigval[1] - A <<endl;
cout <<"Matrix l_3I - A:" << "\n" << I*eigval[2] - A <<endl;
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
```

```
            mat A3 = I*real(eigval[2]) - A;

            cout << endl;
            cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
            cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
            cout << "Eigenspaces for l_3: " << n - arma::rank(A3) << endl;
```

```
Matrix A:
        0        0  -2.0000
   1.0000   2.0000   1.0000
   1.0000        0   3.0000

Eigenvalues:
λ_1 = 2 + 0i
λ_2 = 1 + 0i
λ_3 = 2 + 0i

Matrix λ_1I - A:
    (+2.000e+00,+0.000e+00)                     (0,0)    (+2.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)                     (0,0)    (-1.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)                     (0,0)    (-1.000e+00,+0.000e+00)

Matrix λ_2I - A:
    (+1.000e+00,+0.000e+00)                     (0,0)    (+2.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)    (-1.000e+00,+0.000e+00)    (-1.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)                     (0,0)    (-2.000e+00,+0.000e+00)

Matrix λ_3I - A:
    (+2.000e+00,+0.000e+00)                     (0,0)    (+2.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)                     (0,0)    (-1.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)                     (0,0)    (-1.000e+00,+0.000e+00)


Eigenspaces for λ_1: 2
Eigenspaces for λ_2: 1
Eigenspaces for λ_3: 2

Eigenvectors:
x[λ_1] :
         0
   1.0000
        0

x[λ_2] :
   -0.8165
   0.4082
   0.4082

x[λ_3] :
    0.7071
         0
  -0.7071
```

**Figure 23.114:** *The computation to find eigenvalues and eigenvectors of a $3 \times 3$ matrix with Armadillo library. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Eigenvectors with Armadillo/main.cpp**).*

Now for the comparison, when we calculate with **Eigen**, the eigenvectors that it compute is a bit different. It must be the numerical method used by **Eigen** library is different than the one used by **Armadillo** to compute the approximation of eigenvector.

```
Matrix A:
 0  0 -2
 1  2  1
 1  0  3

Eigenvalues:
λ_1 = 1 + 0i
λ_2 = 2 + 0i
λ_3 = 2 + 0i

Eigenvectors:
x[λ_1] = (-0.816497 + 0i, 0.408248 + 0i, 0.408248 + 0i )

x[λ_2] = (-0.57735 + 0i, 0.57735 + 0i, 0.57735 + 0i )

x[λ_3] = (-0.0879732 + 0i, -0.992231 + 0i, 0.0879732 + 0i )
```

**Figure 23.115:** *The computation to find eigenvalues and eigenvectors of the same $3 \times 3$ matrix with Eigen library resulting in different eigenvectors.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Eigenvectors with Eigen/main.cpp)****.*

## iii. C++ Computation: Determine Rank, Column Space and Null Space for Symmetric Matrix

In the previous section (General Vector Spaces section) we have learned the manual and rigorous way to determine rank, kernel, null space, column space. Suppose we have a symmetric matrix

$$A = \begin{bmatrix} -1 & 2 & 5 \\ 2 & 1 & 4 \\ 3 & 0 & 3 \end{bmatrix}$$

and we want to compute :

1. rank($A$)
2. nullity($A$)
3. Basis for the null space of $A$
4. Basis for the column space of $A$

What makes it more interesting is we are going to use **Eigen** library in this section, the first time we are dwelling on this library, because it is really famous and popular, even CERN is using this library. This library is amazing, it is header only, so we do not need to compile with **-leigen** like we always do with **Armadillo, GiNac, symbolicc++**, we only need to put **#include <eigen3/Eigen/Dense>** at the beginning of our code.

If you read back at the section where I am coding with **Armadillo** and use labourous, manual way to determine basis for the null space and column space of any kind of matrix (symmetric and non-symmetric), this one you can get the basis for null space and column space automatically, given that the matrix is symmetric.

```
#include <eigen3/Eigen/Dense> // header file
#include <iostream>

int main()
{
```

```
        Eigen::Matrix3f A;
        A << -1, 2, 5,
        2, 1, 4,
        3, 0, 3;

        int n = 3; // the size of the matrix A

        std::cout << "Here is the matrix A:\n" << A << std::endl;
        Eigen::FullPivLU<Eigen::Matrix3f> lu_decomp(A);
        std::cout << "The rank of A is " << lu_decomp.rank() << std::endl;
        std::cout << "The nullity of A is " << n - lu_decomp.rank() << std
            ::endl;
        std::cout << "Here is a matrix whose columns form a basis of the
            null-space of A:\n"
        << lu_decomp.kernel() << std::endl;
        std::cout << "Here is a matrix whose columns form a basis of the
            column-space of A:\n"
        << lu_decomp.image(A) << std::endl; // yes, have to pass the
            original A
    }
```

**C++ Code 174:** *main.cpp "Solving Linear System with Equations Less than Unknowns"*

To compile it, type:
**g++ -o main main.cpp**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- You define the matrix with Eigen' class of **Matrix3F** as a symmetric matrix with size of 3. Then you give input of the entries.

```
        Eigen::Matrix3f A;
        A << -1, 2, 5,
        2, 1, 4,
        3, 0, 3;
```

```
Here is the matrix A:
-1  2  5
 2  1  4
 3  0  3
The rank of A is 3
The nullity of A is 0
Here is a matrix whose columns form a basis of the null-space of A:
0
0
0
Here is a matrix whose columns form a basis of the column-space of A:
 5 -1  2
 4  2  1
 3  3  0
```

**Figure 23.116:** *The computation to find the rank, basis for null space and column space for symmetric matrix A with Eigen library. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Rank, Null Space and Column Space with Eigen/main.cpp**).*

## iv. C++ Computation: Polynomial Division and Compute Eigenvalues with Newton-Raphson and Bisection Method

This computation is coming from example number 16 from chapter 5.1 of [11] book.

Find $\det(A)$ given that $A$ has $p(\lambda)$ as its characteristic polynomial.

1. $p(\lambda) = \lambda^3 - 2\lambda^2 + \lambda + 5$
2. $p(\lambda) = \lambda^4 - \lambda^3 + 7$

**Solution:**

1. From the theorem of invertibility, we know that for characteristic equation

$$\lambda^n + c_1\lambda^{n-1} + \cdots + c_n = 0$$

to have the solution of $\lambda = 0$, it will occurs if and only if $c_n = 0$. Thus, it suffices to prove that $A$ is invertible if and only if $c_n \neq 0$. From the determinant formula we will have

$$\det(\lambda I - A) = \lambda^n + c_1\lambda^{n-1} + \cdots + c_n$$

now when we set $\lambda = 0$

$$\begin{aligned} \det(-A) &= c_n \\ (-1)^n \det(A) &= c_n \end{aligned}$$

if $c_n = 0$ then $\det(A) = 0$ and that means $A$ is not invertible, it is a contradiction. Hence $A$ is invertible if and only if $c_n \neq 0$.

Now, we can use the formula above to determine $\det(A)$ knowing that we know $p(\lambda)$.

$$\begin{aligned} (-1)^n \det(A) &= c_n \\ (-1)^3 \det(A) &= 5 \\ \det(A) &= -5 \end{aligned}$$

2. For $p(\lambda) = \lambda^4 - \lambda^3 + 7$, we can use the formula above to determine $\det(A)$ knowing that we know $p(\lambda)$.

$$\begin{aligned}(-1)^n \det(A) &= c_n \\ (-1)^4 \det(A) &= 7 \\ \det(A) &= 7\end{aligned}$$

To answer the questions above by following the proof of theorem is quite easy, but we want to dig deeper and try to find the eigenvalue / root of the characteristic polynomial. Thus, we create a lot of C++ codes here, which can be merge into one if anyone want it:

1. C++ code to do polynomial factorization
   To decompose $\lambda^n + c_1\lambda^{n-1} + \cdots + c_n = 0$ into $(\lambda + c_n)(\lambda^{n-1} + d_1\lambda^{n-2} + \cdots + d_n)$, with $c_1, c_2, \ldots, c_n$ and $d_1, d_2, \ldots, d_n$ are constant.

2. C++ code to plot two univariate functions:

$$p(\lambda) = \lambda^n + c_1\lambda^{n-1} + \cdots + c_n = 0$$

   and

$$\lambda^n + c_1\lambda^{n-1} + \cdots + c_n = \text{reminder}\left(\frac{p(\lambda)}{\lambda + c_n}\right)$$

3. The C++ code to compute the eigenvalue / the root of $p(\lambda)$ with Bisection method and Newton-Raphson method. We can compare the result between two methods.

```
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

int main()
{
        Digits = 5; // define maximum decimal digits
        symbol l("l");

        // construct an expression with arithmetic operators
        ex characteristic_polynomial = pow(l,3) - 2 * (pow(l,2)) + l + 5;
        //ex characteristic_polynomial = pow(l,3) - 8 * (pow(l,2)) + 17*l -
             4;

        cout << endl;
        cout << "p(l) = " << characteristic_polynomial << endl;
        cout << endl;
        // cout << factor(characteristic_polynomial, factor_options::all) <<
             endl;
        // quo = compute the quotient of univariate polynomials in
            polynomial division
```

```
        // rem = compute the reminder of univariate polynomials in
            polynomial division

        ex a = quo(characteristic_polynomial, l + 5, l) ;
        ex b = rem(characteristic_polynomial, l + 5, l);
        ex c = a + b;

        cout << "Quotient of p(l) / (l+5) = " << a << endl;
        cout << "Reminder of p(l) / (l+5) = " << b << endl;
        cout << endl;

        cout << "Quotient * (l+5) = " << expand((l+5) * a) << endl;

        cout << "[ Quotient * (l+5) ] + reminder = " << expand((l+5) * a) +
            b << endl;

        return 0;
    }
```

**C++ Code 175:** *main.cpp "Polynomial factorization"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
p(λ) = 5-2*λ^2+λ^3+λ

Quotient of p(λ) / (λ+5) = 36+λ^2-7*λ
Reminder of p(λ) / (λ+5) = -175

Quotient * (λ+5) = 180-2*λ^2+λ^3+λ
[ Quotient * (λ+5) ] + reminder = 5-2*λ^2+λ^3+λ
```

**Figure 23.117:** *The computation to obtain $\lambda^3 - 2\lambda^2 + \lambda + 5 = -175$ if we divide $\lambda^3 - 2\lambda^2 + \lambda + 5 = 0$ by $\lambda + 5$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Polynomial Factorization with GiNaC/main.cpp**).*

```
    #include <vector>
    #include <cmath>
    #include <utility>
    #include <boost/tuple/tuple.hpp>

    #include "gnuplot-iostream.h"

    int main() {
        Gnuplot gp;
```

```
        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-10:10]\nset yrange [-960:960]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = (x**3)-(2*x**2) + x + 5\n";
        gp << "g(x) = (x**3)-(2*x**2) + x + 180\n";
        gp << "plot f(x) title 'l^{3} - 2l^{2} + l + 5 = 0', g(x) title 'l
            ^{3} - 2l^{2} + l + 5 = -175'\n";

        return 0;
}
```

**C++ Code 176:** *main.cpp "Plot two Polynomial functions"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

The C++ code below is actually part of Numerical Methods chapter, but we modify and join two root-finding methods: Bisection and newton-Raphson, so we can compare the results from each method in this Linear Algebra section, this time in particular this characteristic polynomial can get a lot of help from numerical method to approximate the root / solution of a univariate function. If you want to know more about root-finding methods you can read the chapter dedicated for Numerical Methods.

```
#include <iostream>
#include <ginac/ginac.h>

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#define pi 3.1415926535897

using namespace std;
using namespace GiNaC;

int main()
{
        Digits = 10; // define maximum decimal digits
        symbol l("l");
        int N = 17;
        ex f, fd, fp, fpd, fpb, fa, fb;
        ex pn;
        ex p0 = (pi/4);

        f = pow(l,3) - 2*pow(l,2) + l + 5;
```

**Figure 23.118:** *The plot of* $\lambda^3 - 2\lambda^2 + \lambda + 5 = 0$ *and* $\lambda^3 - 2\lambda^2 + \lambda + 5 = -175$. ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Polynomial Functions/main.cpp)***.

```
// Newton—Raphson formula
fd = diff(f,l);
fp = subs(f,l==p0);
fpd = subs(fd,l==p0);


// Bisection parameters and formula
float a = —3;
float b = 2;
float p = a + (b—a)/2 ;

fa = subs(f,l==a);
fb = subs(f,l==b);
fpb = subs(f,l==p);

cout << "f(x) = " << f <<endl;
cout << endl;
cout << "f'(x) = " << fd <<endl;
cout << endl;
cout << "p_{0} = " << p0 <<endl;
cout << endl;
cout << "a = " << a <<endl;
cout << "b = " << b <<endl;
cout << "p = " << p <<endl;
cout << "f(a) = " << fa << endl;
cout << "f(b) = " << fb << endl;
cout << "f(p) = " << fp << endl;

cout << setw(6) << "n" << "\t\t\t" << "p_{n} (Newton—Raphson)" << "
    \t\t\t" << "p_{n} (Bisection)" << "\n\n";
cout << setw(6) << "0" << "\t\t\t" << p0 << "\t\t\t" << p<< "\n";
for (int i = 1; i <=N; i++)
{
       fp = subs(f,l==p0);
       fpd = subs(fd,l==p0);
       pn = p0 — (fp/fpd);
       // Bisection
       p = a + (b—a)/2 ;
       fa = subs(f,l==a);
       fpb = subs(f,l==p);

       if (fa * fpb > 0)
       {
              cout << setw(6) << i << "\t\t\t" << pn << "\t\t\t" <<
                  p << "\n";
              a = p;
              if ((b—a)/2 < pow(10,—4))
```

```
                        {
                                cout << setw(6) << "Bisection method converges"
                                     << "\n";
                        }
                }
                else
                {
                        cout << setw(6) << i << "\t\t\t" << pn << "\t\t\t" <<
                            p << "\n";
                        b = p;
                        if ((b-a)/2 < pow(10,-4))
                        {
                                cout << setw(6) << "Bisection method converges"
                                     << "\n";
                        }
                }

                if (abs(p0 - pn) < pow(10,-5))
                {
                        cout << "Newton-Raphson method converges" << endl;
                        //break;
                }
                p0 = pn;
        }
        return 0;
}
```

**C++ Code 177:** *main.cpp "Bisection and Newton Raphson method"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
thods/Bisection and Newton-Raphson Method with GiNaC ]# ./main
f(x) = 5+λ+λ^3-2*λ^2

f'(x) = 1-4*λ+3*λ^2

p_{0} = 0.785398163397425

a = -3
b = 2
p = -0.5
f(a) = -43.0
f(b) = 7.0
f(p) = 5.03617068639097
     n                  p_{n} (Newton-Raphson)                  p_{n} (Bisection)

     0                  0.785398163397425                       -0.5
     1                  18.089339367750178                      -0.5
     2                  12.280460973630667                      -1.75
     3                  8.403029532006478                       -1.125
     4                  5.805522620328828                       -0.8125
     5                  4.042727858442355                       -0.96875
     6                  2.7896767300603553                      -1.04688
     7                  1.7330356293088183                      -1.08594
     8                  -0.19388071998137635                    -1.10547
     9                  -2.695427984098808                      -1.11523
    10                  -1.7480949718727015                     -1.12012
    11                  -1.2701397169706574                     -1.11768
    12                  -1.1285955006997808                     -1.11646
    13                  -1.1164297053802903                     -1.11584
    14                  -1.1163433029629386                     -1.11615
    15                  -1.1163432986242117                     -1.1163
Bisection method converges
Newton-Raphson method converges
    16                  -1.1163432986242117                     -1.11638
Bisection method converges
Newton-Raphson method converges
    17                  -1.1163432986242117                     -1.11634
Bisection method converges
Newton-Raphson method converges
```

**Figure 23.119:** *The computation to determine the root / eigenvalue of $\lambda^3 - 2\lambda^2 + \lambda + 5 = 0$ with root finding method Bisection and Newton-Raphson, both methods converges after 15 iterations, the eigenvalue obtained is $\lambda = -1.11634$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Bisection and Newton-Raphson Method with GiNaC/main.cpp**).*

## v. C++ Computation: Finding a Matrix $P$ that Diagonalizes a Matrix $A$ and the Diagonal Matrix $D = P^{-1}AP$

Find a matrix $P$ that diagonalizes

$$A = \begin{bmatrix} 0 & 0 & -2 \\ 1 & 2 & 1 \\ 1 & 0 & 3 \end{bmatrix}$$

then find the diagonal matrix $D = P^{-1}AP$.

**Solution:**

First we will compute the

$$\det(\lambda I - A) = 0$$

to obtain the characteristic equation which is

$$(\lambda - 1)(\lambda - 2)^2 = 0$$

and we found the following bases for the eigenspaces:
For $\lambda = 2$:

$$\boldsymbol{p}_1 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{p}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

For $\lambda = 1$:

$$\boldsymbol{p}_3 = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

There are three basis vectors / eigenvectors for $A$ in total, so the matrix

$$P = \begin{bmatrix} -1 & 0 & -2 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

diagonalizes $A$.

Now, for the diagonal matrix $D$ we can compute it with

$$D = P^{-1}AP = \begin{bmatrix} 1 & 0 & 2 \\ 1 & 1 & 1 \\ -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 0 & -2 \\ 1 & 2 & 1 \\ 1 & 0 & 3 \end{bmatrix} \begin{bmatrix} -1 & 0 & -2 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, there is no preferred order for the columns of $P$. Since the $i$th diagonal entry of $P^{-1}AP$ is an eigenvalue for the $i$th column vector of $P$, changing the order of the columns of $P$ just change the order of the eigenvalues on the diagonal of $P^{-1}AP$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
```

```
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        cx_mat eigvec;
        cx_vec eigval;

        mat I(3,3,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 3;
        for (int i = 1; i <= n; i++)
        {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
        }

        mat A1 = I*real(eigval[0]) - A;
        mat A2 = I*real(eigval[1]) - A;
        mat A3 = I*real(eigval[2]) - A;

        cout << endl;
        cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
        cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
        cout << "Eigenspaces for l_3: " << n - arma::rank(A3) << endl;
        cout << endl;

        cout << "Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
                cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                    endl;
        }

        mat P(3,3,fill::eye);
        mat PI(3,3,fill::eye);
        P = real(eigvec);
        PI = inv(P);
        cout <<"Matrix P:" << "\n" << P <<endl;
        cout <<"Matrix P^{-1}:" << "\n" << PI <<endl;
```

```
        cout <<"Matrix P^{-1}AP:" << "\n" << PI*A*P <<endl;

        return 0;
    }
```

**C++ Code 178:** *main.cpp "Finding a matrix P that diagonalizes matrix A and the diagonal matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- From the previous sections we have learned how to compute eigenvalues and eigenvectors with **Armadillo**. Now, we construct 2 new matrices as matrix $P$ and $P^{-1}$, matrix $P$ can be obtained by inserting the column vectors as the eigenvectors of matrix $A$, the order of the iegenvector does not matter, since what will be obtained is a diagonal matrix $D = P^{-1}AP$ that has the eigenvalues as its' diagonal entries.

```
        mat P(3,3,fill::eye);
        mat PI(3,3,fill::eye);
        P = real(eigvec);
        PI = inv(P);
        cout <<"Matrix P:" << "\n" << P <<endl;
        cout <<"Matrix P^{-1}:" << "\n" << PI <<endl;
        cout <<"Matrix P^{-1}AP:" << "\n" << PI*A*P <<endl;
```

```
Matrix A:
         0         0   -2.0000
    1.0000    2.0000    1.0000
    1.0000         0    3.0000

Eigenvalues:
λ_1 = 2 + 0i
λ_2 = 1 + 0i
λ_3 = 2 + 0i

Eigenspaces for λ_1: 2
Eigenspaces for λ_2: 1
Eigenspaces for λ_3: 2

Eigenvectors:
x[λ_1] :
          0
    1.0000
         0

x[λ_2] :
   -0.8165
    0.4082
    0.4082

x[λ_3] :
    0.7071
         0
   -0.7071

Matrix P:
         0   -0.8165    0.7071
    1.0000    0.4082         0
         0    0.4082   -0.7071

Matrix P^{-1}AP:
    2.0000         0         0
         0    1.0000         0
         0         0    2.0000
```

**Figure 23.120:** *The computation to find a matrix $P$ that diagonalizes a matrix $A$ and then to compute the diagonal matrix $D = P^{-1}AP$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix P that Diagonalizes Matrix A with Armadillo/-main.cpp**).*

## vi. C++ Computation: Powers of a Matrix

Compute $A^{10}$ with

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & -3 & 1 \end{bmatrix}$$

**Solution:**

Matrix $A$ is diagonalizable, and it is diagonalized by

$$P = \begin{bmatrix} 1 & 0.4472 & -0.3015 \\ 0 & 0 & 0.3015 \\ 0 & 0.8944 & -0.9045 \end{bmatrix}$$

and

$$D = P^{-1}AP = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

this shows that $A$ has three distinct eigenvalues

$$\lambda_1 = -1$$
$$\lambda_2 = 1$$
$$\lambda_3 = 2$$

Thus,

$$A^{10} = PD^{10}P^{-1} = \begin{bmatrix} 1 & 0.4472 & -0.3015 \\ 0 & 0 & 0.3015 \\ 0 & 0.8944 & -0.9045 \end{bmatrix} \begin{bmatrix} -1^{10} & 0 & 0 \\ 0 & 1^{10} & 0 \\ 0 & 0 & 2^{10} \end{bmatrix} \begin{bmatrix} 1 & -0.5 & -0.5 \\ 0 & 3.3541 & 1.1180 \\ 0 & 3.3166 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1023 & 0 \\ 0 & 1024 & 0 \\ 0 & -3069 & 1 \end{bmatrix}$$

For the C++ code, we are using **Armadillo** library, we use textfile **matrixA.txt** to take the input of matrix $A$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        cx_mat eigvec;
        cx_vec eigval;
```

```
mat I(3,3,fill::eye);
cout <<"Matrix A:" << "\n" << A <<endl;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense
    general square matrix

cout << "Eigenvalues:" << endl;

int n = 3;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " +
           " << imag(eigval[i-1]) << "i" << endl;
}

mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
mat A3 = I*real(eigval[2]) - A;

cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) <<
    endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) <<
    endl;
cout << "Eigenspaces for l_3: " << n - arma::rank(A3) <<
    endl;
cout << endl;

cout << "Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i
           -1)) << endl;
}

mat P(3,3,fill::eye);
mat PI(3,3,fill::eye);
P = real(eigvec);
PI = inv(P);

mat D(3,3,fill::eye);
D = PI*A*P;
for (int i = 0; i < n; i++)
{
       for (int j = 0; j < n; j++)
       {
              if (i !=j)
              {
```

```
                                        D(i,j) = 0;
                                }
                        }
                }

                cout <<"Matrix P:" << "\n" << P <<endl;
                cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;

                return 0;
        }
```

**C++ Code 179:** *main.cpp "Powers of a diagonalizable matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We declare three new matrices $P, P^{-1}$, and $D$. We give entries of $P$ from the real values of eigenvectors of matrix $A$, but beware if matrix $A$ is not diagonalizable, it will return with an error that saying the matrix $P$ is singular. If it is diagonalizable then we proceed to compute the $P^{-1}$, with **Armadillo** we just need to call the function **inv(**$P$**)**. Then the last is $D = P^{-1}AP$ that we will use to do the computation of powers of matrix.

```
        mat P(3,3,fill::eye);
        mat PI(3,3,fill::eye);
        P = real(eigvec);
        PI = inv(P);

        mat D(3,3,fill::eye);
        D = PI*A*P;
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        if (i !=j)
                        {
                                D(i,j) = 0;
                        }
                }
        }

        cout <<"Matrix P:" << "\n" << P <<endl;
        cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;
```

```
Matrix A:
  -1.0000        0   1.0000
        0   2.0000        0
        0  -3.0000   1.0000

Eigenvalues:
λ_1 = -1 + 0i
λ_2 = 1 + 0i
λ_3 = 2 + 0i

Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1
Eigenspaces for λ_3: 1

Matrix P:
   1.0000   0.4472  -0.3015
        0        0   0.3015
        0   0.8944  -0.9045

Matrix P^{-1}:
   1.0000  -0.5000  -0.5000
        0   3.3541   1.1180
        0   3.3166        0

Matrix P^{-1}AP:
  -1.0000        0        0
        0   1.0000        0
        0        0   2.0000

Matrix A^{10} = P D^{10} P^{-1} :
   1.0000e+00  -1.0230e+03            0
            0   1.0240e+03            0
            0  -3.0690e+03   1.0000e+00

Matrix A^{10} manual way:
   1.0000e+00  -1.0230e+03            0
            0   1.0240e+03            0
            0  -3.0690e+03   1.0000e+00
```

**Figure 23.121:** *The computation to find the powers of a matrix $A$ that is diagonalizable with $D = P^{-1}AP$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Powers of a Matrix A that is Diagonalizable with Armadillo/main.cpp**).*

## VII. Complex Vector Spaces

[DF*] The characteristic equation of any square matrix can have complex solutions, the notion of complex eigenvalues and eigenvectors arise naturally, even within the context of matrices with real entries.

[DF*] If $z = a + bi$ is a complex number, then:

- $\text{Re}(z) = a$ and $\text{Im}(z) = b$ are called the real part of $z$ and the imaginary part of $z$, respectively.
- $|z| = \sqrt{a^2 + b^2}$ is called the modulus (or absolute value) of $z$.
- $\bar{z} = a - bi$ is called the complex conjugate of $z$.
- $z\bar{z} = a^2 + b^2 = |z|^2$
- The angle $\phi$ is called an argument of $z$.
- $\text{Re}(z) = |z| \cos\ \phi$
- $\text{Im}(z) = |z| \sin\ \phi$
- $e^{i\phi} = \cos\ \phi + i\ \sin\ \phi$
- $z = |z|(\cos\ \phi + i\ \sin\ \phi)$ is called the polar form of $z$.

**Figure 23.122:** *The complex number can be represented as $z = a + bi = |z|e^{i\phi}$, with the angle $\phi$ is the argument of $z$.*

[DF*] It is possible for the characteristic equation of a matrix $A$ with real entries to have imaginary solutions, for example

$$A = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}$$

$$p(\lambda) = \det(\lambda I - A) = \lambda^2 + 1 = 0$$

the characteristic polynomial above has solutions

$$\lambda^2 + 1 = 0$$
$$\lambda = \pm i$$

[DF*] A vector space in which scalars are allowed to be complex numbers is called a complex vector space.

**Definition 23.48: Complex Vector Space**

If $n$ is a positive integer, then a complex $n$-tuple is a sequence of $n$ complex numbers $(v_1, v_2, \ldots, v_n)$. The set of all complex $n$-tuples is called complex $n$-space and is denoted by $\mathbb{C}^n$. Scalars are complex numbers, and the operations of addition, subtraction, and scalar multiplication are performed componentwise.

[DF*] If $v_1, v_2, \ldots, v_n$ are complex numbers, then we call $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ a vector in $\mathbb{C}^n$ and $v_1, v_2, \ldots, v_n$ its components. Some examples of vectors in $\mathbb{C}^3$ are

$$\boldsymbol{u} = (1 + 8i, 5, -3i)$$
$$\boldsymbol{v} = (1, 0, 2i)$$
$$\boldsymbol{w} = (\pi i, 6 - \sqrt{10}i, \frac{1}{2}i)$$

Every vector

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_n) = (a_1 + b_1 i, a_2 + b_2 i, \ldots, a_n + b_n i)$$

in $\mathbb{C}^n$ can be split into real and imaginary parts as

$$\boldsymbol{v} = (a_1, a_2, \ldots, a_n) + i(b_1, b_2, \ldots, b_n)$$

which can be denoted as

$$\boldsymbol{v} = \mathrm{Re}(\boldsymbol{v}) + i\ \mathrm{Im}(\boldsymbol{v})$$

where

$$\mathrm{Re}(\boldsymbol{v}) = (a_1, a_2, \ldots, a_n)$$
$$\mathrm{Im}(\boldsymbol{v}) = (b_1, b_2, \ldots, b_n)$$

The vector

$$\overline{\boldsymbol{v}} = (\overline{v_1}, \overline{v_2}, \ldots, \overline{v_n}) = (a_1 - b_1 i, a_2 - b_2 i, \ldots, a_n - b_n i)$$

is called the complex conjugate of $\boldsymbol{v}$ and can be expressed in terms of $\mathrm{Re}(\boldsymbol{v})$ and $\mathrm{Im}(\boldsymbol{v})$ as

$$\overline{\boldsymbol{v}} = (a_1, a_2, \ldots, a_n) - i(b_1, b_2, \ldots, b_n) = \mathrm{Re}(\boldsymbol{v}) - i\mathrm{Im}(\boldsymbol{v}) \tag{23.126}$$

It follows that the vectors in $\mathbb{R}^n$ can be viewed as those vectors in $\mathbb{C}^n$ whose imaginary part is zero.

A vector $\boldsymbol{v}$ in $\mathbb{C}^n$ is in $\mathbb{R}^n$ if and only if

$$\overline{\boldsymbol{v}} = \boldsymbol{v}$$

[DF*] We call a matrix $A$ a real matrix if its entries are required to be real numbers and a complex matrix if its entries are allowed to be complex numbers. The standard operations on real matrices carry over to complex matrices without change, and all of familiar properties of matrices continue to hold.
If $A$ is a complex matrix, then $\mathrm{Re}(A)$ and $\mathrm{Im}(A)$ are the matrices formed from the real and imaginary parts of the entries of $A$, and the $\overline{A}$ is the matrix formed by taking the complex conjugate of each entry in $A$.

**Theorem 23.93: Algebraic Properties of the Complex Conjugate for Vectors**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{C}^n$, and if $k$ is a scalar, then:

(a) $\overline{\overline{\boldsymbol{u}}} = \boldsymbol{u}$

(b) $\overline{k\boldsymbol{u}} = k\overline{\boldsymbol{u}}$

(c) $\overline{\boldsymbol{u} + \boldsymbol{v}} = \overline{\boldsymbol{u}} + \overline{\boldsymbol{v}}$

(d) $\overline{\boldsymbol{u} - \boldsymbol{v}} = \overline{\boldsymbol{u}} - \overline{\boldsymbol{v}}$

**Theorem 23.94: Algebraic Properties of the Complex Conjugate for Matrices**

If $A$ is an $m \times k$ complex matrix and $B$ is a $k \times n$ complex matrix, then:

(a) $\overline{\overline{A}} = A$

(b) $(\overline{A^T}) = (\overline{A})^T$

(c) $\overline{AB} = \overline{A}\,\overline{B}$

**Definition 23.49: Complex Euclidean Inner Product**

If $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ are vectors in $\mathbb{C}^n$, then the complex Euclidean inner product of $\boldsymbol{u}$ and $\boldsymbol{v}$ (also called complex dot product) is denoted by $\boldsymbol{u} \cdot \boldsymbol{v}$ and is defined as

$$\boldsymbol{u} \cdot \boldsymbol{u} = u_1\overline{v_1} + u_2\overline{v_2} + \cdots + u_n\overline{v_n} + \tag{23.127}$$

We also define the Euclidean norm on $\mathbb{C}^n$ to be

$$||\boldsymbol{v}|| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}} = \sqrt{|v_1|^2 + |v_2|^2 + \cdots + |v_n|^2} \tag{23.128}$$

As in the real case, we call $\boldsymbol{v}$ a unit vector in $\mathbb{C}^n$ if $||\boldsymbol{v}|| = 1$, and we say two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal if $\boldsymbol{u} \cdot \boldsymbol{v} = 0$.

[DF*] If $\boldsymbol{u}$ and $\boldsymbol{v}$ are column vectors in $\mathbb{R}^n$, then their dot product can be expressed as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}^T\boldsymbol{v} = \boldsymbol{v}^T\boldsymbol{u}$$

The analogous formulas in $\mathbb{C}^n$ are

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}^T\overline{\boldsymbol{v}} = \overline{\boldsymbol{v}}^T\boldsymbol{u} \tag{23.129}$$

**Theorem 23.95: Properties of Complex Euclidean Inner Product**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in $\mathbb{C}^n$, and if $k$ is a scalar, then the complex Euclidean inner product has the following properties:

(a) $\boldsymbol{u} \cdot \boldsymbol{v} = \overline{\boldsymbol{v} \cdot \boldsymbol{u}}$ (Antisymmetry property)

(b) $\boldsymbol{u} \cdot (\boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{u} \cdot \boldsymbol{w}$ (Distributive property)

(c) $k(\boldsymbol{u} \cdot \boldsymbol{v}) = (k\boldsymbol{u}) \cdot \boldsymbol{v}$ (Homogeneity property)

(d) $\boldsymbol{u} \cdot k\boldsymbol{v} = \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v})$ (Antihomogeneity property)

(e) $\boldsymbol{v} \cdot \boldsymbol{v} \geq 0$ and $\boldsymbol{v} \cdot \boldsymbol{v} = 0$ if and only if $\boldsymbol{v} = \mathbf{0}$ (Positivity property)

[DF*] Except for the use of complex scalars, the notions of linear combination, linear independence, subspace, spanning, basis, and dimension carry over without change to $\mathbb{C}^n$.

[DF*] Eigenvalues and eigenvectors are defined for complex matrices exactly as for real matrices.

If $A$ is an $n \times n$ matrix with complex entries, then the complex roots of the characteristic equation

$$\det(\lambda I - A) = 0$$

are called complex eigenvalues of $A$.

As in the real case, $\lambda$ is a complex eigenvalue of $A$ if and only if there exists a nonzero vector $\boldsymbol{x}$ in $\mathbb{C}^n$ such that

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

Each such $\boldsymbol{x}$ is called a complex eigenvector of $A$ corresponding to $\lambda$.

The complex eigenvectors of $A$ corresponding to $\lambda$ are the nonzero solutions of the linear system

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

and the set of all such solutions is a subspace of $\mathbb{C}^n$, called the eigenspace of $A$ corresponding to $\lambda$.

**Theorem 23.96: Conjugate Pairs in Eigenvectors and Eigenvalues**

If $\lambda$ is an eigenvalue of a real $n \times n$ matrix $A$, and if $\boldsymbol{x}$ is a corresponding eigenvector, then $\overline{\lambda}$ is also an eigenvalue of $A$, and $\overline{\boldsymbol{x}}$ is a corresponding eigenvector.

[DF*] **Example:**
Find the eigenvalues and bases for the eigenspaces of

$$A = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}$$

**Solution:**
The characteristic polynomial of $A$ is

$$\begin{vmatrix} \lambda + 2 & 1 \\ -5 & \lambda - 2 \end{vmatrix} = \lambda^2 + 1 = (\lambda - i)(\lambda + i)$$

so the eigenvalues of $A$ are $\lambda = i$ and $\lambda = -i$. Note that these eigenvalues are complex conjugates. To find the eigenvectors we must solve the system

$$\begin{bmatrix} \lambda + 2 & 1 \\ -5 & \lambda - 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

substitute with $\lambda = i$, the system becomes

$$\begin{bmatrix} i + 2 & 1 \\ -5 & i - 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We could solve this system by reducing the augmented matrix

$$\begin{bmatrix} i + 2 & 1 & 0 \\ -5 & i - 2 & 0 \end{bmatrix}$$

to reduced row echelon form by Gauss-Jordan elimination. Observe that the reduced row echelon form must have a row of zeros in order to have a basis for the corresponding $\lambda = i$,

thus each row of the matrix above must be a scalar multiple of the other.

We can just set one of the row to zero and then do the parameterization. Now we set the first row to zero, so we will have

$$\begin{bmatrix} 0 & 0 & 0 \\ -5 & i-2 & 0 \end{bmatrix}$$

We divide the second row with $-\frac{1}{5}$ then interchange the rows

$$\begin{bmatrix} 1 & \frac{2}{5}-\frac{i}{5} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

then do the parameterization to get the general solution

$$\begin{aligned} x_1 &= \left(-\frac{2}{5}+\frac{i}{5}\right)t \\ x_2 &= t \end{aligned}$$

you can check by substituting the values of $x_1$ and $x_2$ into the first row

$$\begin{bmatrix} i+2 & 1 & 0 \\ -5 & i-2 & 0 \end{bmatrix}$$

The equation will hold for

$$(i+2)x_1 + x_2 = 0$$

you can verify this by hand.

So, the eigenspace corresponding to $\lambda = i$ is one-dimensional and consists of all complex scalar multiples of the basis vector

$$\boldsymbol{x} = \begin{bmatrix} -\frac{2}{5}+\frac{i}{5} \\ 1 \end{bmatrix}$$

As a check, let us confirm that

$$A\boldsymbol{x} = i\boldsymbol{x}$$

$$A\boldsymbol{x} = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}\begin{bmatrix} -\frac{2}{5}+\frac{i}{5} \\ 1 \end{bmatrix} = \begin{bmatrix} -2\left(-\frac{2}{5}+\frac{i}{5}\right)-1 \\ 5\left(-\frac{2}{5}+\frac{i}{5}\right)+2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{5}-\frac{2i}{5} \\ i \end{bmatrix} = i\boldsymbol{x}$$

We could find the basis for the eigenspace corresponding to $\lambda = -i$ in a similar way, but to save time and be more efficient we have a theorem that implies

$$\overline{\boldsymbol{x}} = \begin{bmatrix} -\frac{2}{5}-\frac{i}{5} \\ 1 \end{bmatrix}$$

must be a basis for this eigenspace. The following computations confirm that $\overline{\boldsymbol{x}}$ is an eigenvector of $A$ corresponding to $\lambda = -i$:

$$A\overline{\boldsymbol{x}} = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}\begin{bmatrix} -\frac{2}{5}-\frac{i}{5} \\ 1 \end{bmatrix} = \begin{bmatrix} -2\left(-\frac{2}{5}-\frac{i}{5}\right)-1 \\ 5\left(-\frac{2}{5}-\frac{i}{5}\right)+2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{5}+\frac{2i}{5} \\ -i \end{bmatrix} = -i\overline{\boldsymbol{x}}$$

**Theorem 23.97: Determining Eigenvalues of a $2 \times 2$ Matrix**

If $A$ is a $2 \times 2$ matrix with real entries, then the characteristic equation of $A$ is

$$\lambda^2 - \text{tr}(A)\lambda + \det(A) = 0 \tag{23.130}$$

and

(a) $A$ has two distinct real eigenvalues if $\text{tr}(A)^2 - 4\det(A) > 0$;

(b) $A$ has one repeated real eigenvalue if $\text{tr}(A)^2 - 4\det(A) = 0$;

(c) $A$ has two complex conjugate eigenvalues if $\text{tr}(A)^2 - 4\det(A) < 0$

[DF*] **Example:**
Use the characteristic equation to find the eigenvalues of

$$A = \begin{bmatrix} 2 & 3 \\ -3 & 2 \end{bmatrix}$$

**Solution:** We have

$$\text{tr}(A) = 4$$
$$\det(A) = 13$$

so the characteristic equation of $A$ is

$$\lambda^2 - 4\lambda + 13 = 0$$

Solving this equation by the quadratic formula yields

$$\lambda = \frac{4 \pm \sqrt{(-4)^2 - 4(13)}}{2} = \frac{4 \pm \sqrt{-36}}{2} = 2 \pm 3i$$

Thus, the eigenvalues of $A$ are $\lambda = 2 + 3i$ and $\lambda = 2 - 3i$.

**Theorem 23.98: Symmetric Matrices Have Real Eigenvalues**

If $A$ is a real symmetric matrix, then $A$ has real eigenvalues.

**Theorem 23.99: Geometric Interpretation of Complex Eigenvalues**

The eigenvalues of the real matrix

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{23.131}$$

are $\lambda = a \pm bi$. If $a$ and $b$ are not both zero, then this matrix can be factored as

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \tag{23.132}$$

where $\phi$ is the angle from the positive $x$-axis to the ray that joins the origin to the point $(a, b)$.

**Figure 23.123:** *The geometric interpretation of complex eigenvalue $\lambda = a + bi$ with $\phi$ is the angle from the positive x-axis to the ray that joins the origin to the point $(a, b)$.*

**Figure 23.124:** *The geometric interpretation of vector $\boldsymbol{x}$ being multiplied by matrix $C$ with characteristic equation $(\lambda - a)^2 + b^2 = 0$, the multiplication can be viewed as a rotation through the angle $\phi$ followed by a scaling with factor $|\lambda|$.*

[DF*] Geometrically, a multiplication by a matrix of form

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{23.133}$$

can be viewed as a rotation through the angle $\phi$ followed by a scaling with factor $|\lambda|$. Since the characteristic equation of $C$ is

$$(\lambda - a)^2 + b^2 = 0$$

from which it follows that the eigenvalues of $C$ are

$$\lambda = a \pm bi$$

Assuming that $a$ and $b$ are not both zero, let $\phi$ be the angle from the positive $x$-axis to the ray that joins the origin to the point $(a, b)$. The angle $\phi$ is an argument of the eigenvalue $\lambda = a + bi$, therefore

$$a = |\lambda| \ \cos \ \phi$$
$$b = |\lambda| \ \sin \ \phi$$

It follows from this that the matrix in (23.131) can be written as

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \frac{a}{|\lambda|} & -\frac{b}{|\lambda|} \\ \frac{b}{|\lambda|} & \frac{a}{|\lambda|} \end{bmatrix} = \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \cos \ \phi & -\sin \ \phi \\ \sin \ \phi & \cos \ \phi \end{bmatrix}$$

**Theorem 23.100: Similar Matrix to Real Matrix with Cosmplex Eigenvalues**

Let $A$ be a real $2 \times 2$ matrix with complex eigenvalues $\lambda = a \pm bi$ (where $b \neq 0$). If $\boldsymbol{x}$ is an eigenvector of $A$ corresponding to $\lambda = a - bi$, then the matrix

$$P = [\mathrm{Re}(\boldsymbol{x}) \quad \mathrm{Im}(\boldsymbol{x})]$$

is invertible and

$$A = P \begin{bmatrix} a & -b \\ b & a \end{bmatrix} P^{-1}$$

[DF*] Let

$$A = PCP^{-1} = PSR_\theta P^{-1} = P \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \cos \ \theta & -\sin \ \theta \\ \sin \ \theta & \cos \ \theta \end{bmatrix} P^{-1} \tag{23.134}$$

If we now view $P$ as the transition matrix from the basis $B = \{\mathrm{Re}(\boldsymbol{x}), \mathrm{Im}(\boldsymbol{x})\}$ to the standard basis then computing a product $A\boldsymbol{x}_0$ can be broken down into a three-step process:

1. Map $\boldsymbol{x}_0$ from standard coordinates into $B$-coordinates by forming the product $P^{-1}\boldsymbol{x}_0$.
2. Rotate and scale the vector $P^{-1}\boldsymbol{x}_0$ by forming the product $SR_\theta P^{-1}\boldsymbol{x}_0$.
3. Map the rotated and scaled vector back to standard coordinates to obtain $A\boldsymbol{x_0} = PSR_\theta P^{-1}\boldsymbol{x}_0$.

[DF*] **Example:**
Find $|\lambda|$ and $\phi$ for which $-\pi < \phi \leq \pi$, with

$$C = \begin{bmatrix} \sqrt{2} & \sqrt{2} \\ -\sqrt{2} & \sqrt{2} \end{bmatrix}$$

**Solution:**
We know that

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} \sqrt{2} & \sqrt{2} \\ -\sqrt{2} & \sqrt{2} \end{bmatrix}$$

thus

$$\begin{aligned} a &= \sqrt{2} \\ b &= -\sqrt{2} \end{aligned}$$

and

$$\begin{aligned} |\lambda| &= \sqrt{a^2 + b^2} \\ &= \sqrt{(\sqrt{2})^2 + (-\sqrt{2})^2} \\ &= \sqrt{2+2} \\ &= 2 \end{aligned}$$

we know that these conditions fulfilled

$$\sin\ \phi < 0 \quad \text{and} \quad -\pi < \phi \leq \pi$$

when the angle $\phi$ is in quadrant IV, thus

$$\begin{aligned} \cos\ \phi &= \frac{a}{|\lambda|} \\ &= \frac{\sqrt{2}}{2} \\ \phi &= \cos^{-1}\left(\frac{\sqrt{2}}{2}\right) = \frac{7\pi}{4} \end{aligned}$$

If we factor matrix $C$ it will be

$$C = SR_\phi$$

with $S$ is the scaling matrix, and $R_\phi$ the rotation matrix

$$\begin{bmatrix} \sqrt{2} & \sqrt{2} \\ -\sqrt{2} & \sqrt{2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} \cos\ \frac{7\pi}{4} & -\sin\ \frac{7\pi}{4} \\ \sin\ \frac{7\pi}{4} & \cos\ \frac{7\pi}{4} \end{bmatrix}$$

[DF*] **Example:**
The matrices

$$\sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

called Pauli spin matrices, are used in quantum mechanics to study particle spin. The Dirac matrices, which are also used in quantum mechanics, are expressed in terms of the Pauli spin matrices and the $2 \times 2$ identity matrix $I_2$ as

$$\beta = \begin{bmatrix} I_2 & 0 \\ 0 & I_2 \end{bmatrix}, \quad \alpha_x = \begin{bmatrix} 0 & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix}$$

$$\alpha_y = \begin{bmatrix} 0 & \sigma_2 \\ \sigma_2 & 0 \end{bmatrix}, \quad \alpha_z = \begin{bmatrix} 0 & \sigma_3 \\ \sigma_3 & 0 \end{bmatrix}$$

(a) Show that $\beta^2 = \alpha_x^2 = \alpha_y^2 = \alpha_z^2$.

(b) Matrices $A$ and $B$ for which $AB = -BA$ are said to be anticommutative. Show that the Dirac matrices are anticommutative.

**Solution:**

(a) The computation for Dirac matrices

$$\beta^2 = \begin{bmatrix} I_2 & 0 \\ 0 & -I_2 \end{bmatrix} \begin{bmatrix} I_2 & 0 \\ 0 & -I_2 \end{bmatrix} = \begin{bmatrix} I_2^2 & 0 \\ 0 & I_2^2 \end{bmatrix} = \begin{bmatrix} I_2 & 0 \\ 0 & I_2 \end{bmatrix}$$

$$\alpha_x^2 = \begin{bmatrix} 0 & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix} \begin{bmatrix} 0 & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_1^2 \end{bmatrix}$$

$$\alpha_y^2 = \begin{bmatrix} 0 & \sigma_2 \\ \sigma_2 & 0 \end{bmatrix} \begin{bmatrix} 0 & \sigma_2 \\ \sigma_2 & 0 \end{bmatrix} = \begin{bmatrix} \sigma_2^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

$$\alpha_z^2 = \begin{bmatrix} 0 & \sigma_3 \\ \sigma_3 & 0 \end{bmatrix} \begin{bmatrix} 0 & \sigma_3 \\ \sigma_3 & 0 \end{bmatrix} = \begin{bmatrix} \sigma_3^2 & 0 \\ 0 & \sigma_3^2 \end{bmatrix}$$

Since

$$\sigma_1^2 = \sigma_2^2 = \sigma_3^2 = I_2$$

then

$$\beta^2 = \alpha_x^2 = \alpha_y^2 = \alpha_z^2$$

(b) If we choose

$$A = \beta = \begin{bmatrix} I_2 & 0 \\ 0 & -I_2 \end{bmatrix}$$

and

$$B = \alpha_x = \begin{bmatrix} 0 & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix}$$

we will have

$$AB = \beta\alpha_x = \begin{bmatrix} I_2 & 0 \\ 0 & -I_2 \end{bmatrix} \begin{bmatrix} 0 & \sigma_1 \\ \sigma_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & I_2\sigma_1 \\ -I_2\sigma_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \sigma_1 \\ -\sigma_1 & 0 \end{bmatrix}$$

now, we check for

$$-BA = -\alpha_x\beta = \begin{bmatrix} 0 & -\sigma_1 \\ -\sigma_1 & 0 \end{bmatrix} \begin{bmatrix} I_2 & 0 \\ 0 & -I_2 \end{bmatrix} = \begin{bmatrix} 0 & I_2\sigma_1 \\ -I_2\sigma_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \sigma_1 \\ -\sigma_1 & 0 \end{bmatrix}$$

In conclusion Dirac matries are anti-commutative as long as we choose $\beta$ in the Dirac matrices as either $A$ or $B$, then the other matrix shall be either $\alpha_x, \alpha_y, \alpha_z$ due to the similar placement of nonzero and zeros entries in them.

[DF*] **Example:**
Prove that if $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{C}^n$, then

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{v} = &\frac{1}{4}||\boldsymbol{u} + \boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u} - \boldsymbol{v}||^2 \\ &+ \frac{i}{4}||\boldsymbol{u} + i\boldsymbol{v}||^2 - \frac{i}{4}||\boldsymbol{u} - i\boldsymbol{v}||^2 \end{aligned}$$

**Solution:**
Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be vectors in $\mathbb{C}^n$

$$\begin{aligned}\boldsymbol{u} &= (a_1 + b_1 i, a_2 + b_2 i, \ldots, a_n + b_n i)\\ \boldsymbol{v} &= (c_1 + d_1 i, c_2 + d_2 i, \ldots, c_n + d_n i)\end{aligned}$$

thus

$$\begin{aligned}\boldsymbol{u}\cdot\boldsymbol{v} &= u_1\overline{v_1} + u_2\overline{v_2} + \cdots + u_n\overline{v_n}\\ &= (a_1 + b_1 i)(c_1 - d_1 i) + (a_2 + b_2 i)(c_2 - d_2 i) + \cdots + (a_n + b_n i)(c_n - d_n i)\\ &= [(a_1c_1 + b_1d_1) + (b_1c_1 - a_1d_1)i] + [(a_2c_2 + b_2d_2) + (b_2c_2 - a_2d_2)i]\\ &+ \cdots + [(a_nc_n + b_nd_n) + (b_nc_n - a_nd_n)i]\end{aligned}$$

Now, we will examine the right side of the equation, we will break down all the four terms one by one, by definition we will have

$$\begin{aligned}\frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 &= \frac{1}{4}\sqrt{|u_1+v_1|^2 + |u_2+v_2|^2 + \cdots + |u_n+v_n|^2}^2\\ &= \frac{1}{4}\sqrt{\sum_{i=1}^{n}\sqrt{(a_i+c_i)^2 + (b_i+d_i)^2}^2}^2\\ &= \frac{1}{4}\left[(a_1+c_1)^2 + (b_1+d_1)^2 + (a_2+c_2)^2 + (b_2+d_2)^2 + \cdots + (a_n+c_n)^2 + (b_n+d_n)^2\right]\end{aligned}$$

$$\begin{aligned}\frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2 &= \frac{1}{4}\sqrt{|u_1-v_1|^2 + |u_2-v_2|^2 + \cdots + |u_n-v_n|^2}^2\\ &= \frac{1}{4}\sqrt{\sum_{i=1}^{n}\sqrt{(a_i-c_i)^2 + (b_i-d_i)^2}^2}^2\\ &= \frac{1}{4}\left[(a_1-c_1)^2 + (b_1-d_1)^2 + (a_2-c_2)^2 + (b_2-d_2)^2 + \cdots + (a_n-c_n)^2 + (b_n-d_n)^2\right]\end{aligned}$$

$$\frac{i}{4}||\boldsymbol{u}+i\boldsymbol{v}||^2 = \frac{i}{4}\left[(a_1-d_1)^2 + (b_1+c_1)^2 + (a_2-d_2)^2 + (b_2+c_2)^2 + \cdots + (a_n-d_n)^2 + (b_n+c_n)^2\right]$$

$$\frac{i}{4}||\boldsymbol{u}-i\boldsymbol{v}||^2 = \frac{i}{4}\left[(a_1+d_1)^2 + (b_1-c_1)^2 + (a_2+d_2)^2 + (b_2-c_2)^2 + \cdots + (a_n+d_n)^2 + (b_n-c_n)^2\right]$$

from algebra, we know that $(a+b)^2 = a^2 + 2ab + b^2$ and $(a-b)^2 = a^2 - 2ab + b^2$, hence

$$\frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2 = \frac{1}{4}\left[4a_1c_1 + 4b_1d_1 + 4a_2c_2 + 4b_2d_2 + \cdots + 4a_nc_n + 4b_nd_n\right]$$

$$\frac{i}{4}||\boldsymbol{u}+i\boldsymbol{v}||^2 - \frac{i}{4}||\boldsymbol{u}-i\boldsymbol{v}||^2 = \frac{i}{4}\left[-4a_1d_1 + 4b_1c_1 - 4a_2d_2 + 4b_2c_2 - \cdots + \cdots - 4a_nd_n + 4b_nc_n\right]s$$

it is proven that

$$\begin{aligned}\boldsymbol{u}\cdot\boldsymbol{v} &= \frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2\\ &+ \frac{i}{4}||\boldsymbol{u}+i\boldsymbol{v}||^2 - \frac{i}{4}||\boldsymbol{u}-i\boldsymbol{v}||^2\end{aligned}$$

[DF*] **Example:**
The eigenvalues of the rotation matrix

$$R_\phi \begin{bmatrix} \cos\ \phi & -\sin\ \phi \\ \sin\ \phi & \cos\ \phi \end{bmatrix}$$

are $\lambda = \cos\ \phi \pm i \sin\ \phi$. Prove that if $\boldsymbol{x}$ is an eigenvector corresponding to either eigenvalue, then $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ are orthogonal and have the same length. This implies that $P = [\mathrm{Re}(\boldsymbol{x}) \mid \mathrm{Im}(\boldsymbol{x})]$ is a scalar multiple of an orthogonal matrix.
**Solution:**
First, as a good practice we still need to prove that $\lambda = \cos\ \phi \pm i \sin\ \phi$, thus

$$\begin{vmatrix} \lambda - \cos\ \phi & \sin\ \phi \\ -\sin\ \phi & \lambda - \cos\ \phi \end{vmatrix} = 0$$

$$\begin{aligned} \lambda^2 - 2\lambda \cos\ \phi + \cos^2\phi + \sin^2\phi &= 0 \\ \lambda^2 - 2\lambda \cos\ \phi + 1 &= 0 \\ a &= 1 \\ b &= -2\ \cos\phi \\ c &= 1 \end{aligned}$$

so the eigenvalues can be determined with quadratic root equation

$$\begin{aligned} \lambda_{1,2} &= \frac{2\ \cos\phi \pm \sqrt{4\cos^2\phi - 4(1)(1)}}{2(1)} \\ &= \frac{2\ \cos\phi \pm \sqrt{4(\cos^2\phi - 1)}}{2} \\ &= \frac{2\ \cos\phi \pm \sqrt{4(-\sin^2\phi)}}{2} \\ &= \frac{2\ \cos\phi \pm 2\sqrt{(-1)\sin^2\phi}}{2} \\ &= \frac{2\ \cos\phi \pm 2\sin\phi\sqrt{(-1)}}{2} \\ \lambda_{1,2} &= \cos\phi \pm i\ \sin\phi \end{aligned}$$

Now, to compute the eigenvector corresponding to $\lambda = \cos\phi + i\ \sin\phi$ we weill have to find the solution for the null space of

$$(\lambda I - R_\phi)\boldsymbol{x} = \mathbf{0}$$

$$\begin{bmatrix} i\ \sin\phi & \sin\phi \\ -\sin\phi & i\ \sin\phi \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the augmented matrix will become

$$\begin{bmatrix} i\ \sin\phi & \sin\phi & 0 \\ -\sin\phi & i\ \sin\phi & 0 \end{bmatrix}$$

we solve the system above to reduced row echelon form by Gauss-Jordan elimination. Assuming that the reduced row echelon form will have a row of zeros because $(\lambda I - R_\phi)\boldsymbol{x} = \mathbf{0}$

has nontrivial solutions. Then, each row must be a scalar multiple of the other, we will make row 1 to be zeros and use row 2

$$\begin{aligned} x_1(-\sin\phi) + x_2 i\ \sin\phi &= 0 \\ x_1 &= \frac{-i\ \sin\phi}{-\sin\phi} x_2 \\ x_1 &= i x_2 \end{aligned}$$

the general solution with parameterization is

$$\begin{aligned} x_1 &= it \\ x_2 &= t \end{aligned}$$

the eigenvector corresponding to $\lambda = \cos\phi + i\ \sin\phi$ is

$$\boldsymbol{x} = \begin{bmatrix} i \\ 1 \end{bmatrix}$$

For this $\lambda = \cos\phi + i\ \sin\phi$ we will have

$$\mathrm{Re}(\boldsymbol{x}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathrm{Im}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

To check whether $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ are orthogonal and have the same length we will use dot product and norm formula:

$$\begin{aligned} \mathrm{Re}(\boldsymbol{x}) \cdot \mathrm{Im}(\boldsymbol{x}) &= (0,1) \cdot (1,0) \\ &= 0 \end{aligned}$$

it is proven that $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ are orthogonal.

$$\begin{aligned} ||\mathrm{Re}(\boldsymbol{x})|| &= \sqrt{0^2 + 1^2} = 1 \\ ||\mathrm{Im}(\boldsymbol{x})|| &= \sqrt{1^2 + 0^2} = 1 \end{aligned}$$

The length for $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ are the same.

Now for the second eigenvalue, by theorem, we will know directly that the eigenvector corresponding to $\lambda = \cos\phi - i\ \sin\phi$ is

$$\boldsymbol{x} = \begin{bmatrix} -i \\ 1 \end{bmatrix}$$

For this $\lambda = \cos\phi - i\ \sin\phi$ we will have

$$\mathrm{Re}(\boldsymbol{x}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathrm{Im}(\boldsymbol{x}) = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

To check whether $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ are orthogonal and have the same length we will use dot product and norm formula:

$$\begin{aligned} \mathrm{Re}(\boldsymbol{x}) \cdot \mathrm{Im}(\boldsymbol{x}) &= (0,1) \cdot (-1,0) \\ &= 0 \end{aligned}$$

it is proven that $\text{Re}(\boldsymbol{x})$ and $\text{Im}(\boldsymbol{x})$ are orthogonal.

$$||\text{Re}(\boldsymbol{x})|| = \sqrt{0^2 + 1^2} = 1$$
$$||\text{Im}(\boldsymbol{x})|| = \sqrt{-1^2 + 0^2} = 1$$

The length for $\text{Re}(\boldsymbol{x})$ and $\text{Im}(\boldsymbol{x})$ are the same.

[DF*] **Example:**

(a) Let

$$M = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

and let $\boldsymbol{u} = \text{Re}(\boldsymbol{x})$ and $\boldsymbol{v} = \text{Im}(\boldsymbol{x})$, so $P = [\boldsymbol{u} \mid \boldsymbol{v}]$. Show that the relationship $A\boldsymbol{x} = \lambda\boldsymbol{x}$ implies that

$$A\boldsymbol{x} = (a\boldsymbol{u} + b\boldsymbol{v}) + i(-b\boldsymbol{u} + a\boldsymbol{v})$$

and then equate real and imaginary parts in this equation to show that

$$AP = [A\boldsymbol{u} \mid A\boldsymbol{v}] = [a\boldsymbol{u} + b\boldsymbol{v} \mid -b\boldsymbol{u} + a\boldsymbol{v}] = PM$$

(b) Show that $P$ is invertible, thereby completing the proof, since the result in part (a) implies that $A = PMP^{-1}$. If $P$ is not invertible, the one of its column vectors is a real scalar multiple of the other, say $\boldsymbol{v} = c\boldsymbol{u}$. Substitute this into equations $A\boldsymbol{u} = a\boldsymbol{u} + b\boldsymbol{v}$ and $A\boldsymbol{v} = -b\boldsymbol{u} + a\boldsymbol{v}$ obtained in part (a), and show that $(1 + c^2)b\boldsymbol{u} = \boldsymbol{0}$. Finally, show that this leads to a contradiction, thereby proving that $P$ is invertible

**Solution:**

(a) Let $A$ be a matrix $2 \times 2$ with eigenvalues $\lambda = a \pm bi$, $b \neq 0$. To obtain such eigenvalues then

$$A = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

so

$$\det(\lambda I - A) = 0$$
$$(\lambda - a)^2 + b^2 = 0$$
$$\lambda^2 - 2a\lambda + a^2 + b^2$$

with quadratic root equation we will have

$$\begin{aligned}
\lambda_{1,2} &= \frac{2a \pm \sqrt{4a^2 - 4(1)(a^2 + b^2)}}{2(1)} \\
&= \frac{2a \pm \sqrt{4a^2 - 4a^2 - 4b^2)}}{2(1)} \\
&= \frac{2a \pm \sqrt{-4b^2)}}{2(1)} \\
&= \frac{2a \pm 2b\sqrt{-1}}{2(1)} \\
\lambda_{1,2} &= a \pm bi
\end{aligned}$$

If $\boldsymbol{x}$ is an eigenvector of $A$ corresponding to $\lambda = a - bi$, then

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

$$\begin{bmatrix} (a-bi)-a & b \\ -b & (a-bi)-a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the augmented matrix for the system above is

$$\begin{bmatrix} -bi & b & 0 \\ -b & -bi & 0 \end{bmatrix}$$

to obtain nontrivial solution, we know that one row is a scalar multiple of the other, thus we will use row 2 to get the general solution with parameterization

$$\begin{aligned} -b(x_1) - bi(x_2) &= 0 \\ x_2 &= t \\ x_1 &= \frac{bi}{-b}t = -it \end{aligned}$$

thus the eigenvector $\boldsymbol{x}$ of $A$ corresponding to $\lambda = a - bi$ is

$$\boldsymbol{x} = \begin{bmatrix} -it \\ t \end{bmatrix} = \begin{bmatrix} -i \\ 1 \end{bmatrix} t$$

if we break the eigenvector into the form of $a + bi$

$$\boldsymbol{x} = \begin{bmatrix} -i \\ 1 \end{bmatrix} t = \begin{bmatrix} 0 \\ 1 \end{bmatrix} t + \begin{bmatrix} -1 \\ 0 \end{bmatrix} i$$

with

$$\boldsymbol{u} = \mathrm{Re}(\boldsymbol{x}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{v} = \mathrm{Im}(\boldsymbol{x}) = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Now we will show the implication of $A\boldsymbol{x} = \lambda\boldsymbol{x}$, with our chosen $\lambda = a - bi$

$$A\boldsymbol{x} = \lambda x = (a - bi)\begin{bmatrix} -i \\ 1 \end{bmatrix} = \begin{bmatrix} (a-bi)(-i) \\ (a-bi)(1) \end{bmatrix} = \begin{bmatrix} -b - ai \\ a - bi \end{bmatrix}$$

$$\begin{bmatrix} -b - ai \\ a - bi \end{bmatrix} = \begin{bmatrix} -b \\ a \end{bmatrix} + \begin{bmatrix} -ai \\ -bi \end{bmatrix} = (a\boldsymbol{u} + b\boldsymbol{v}) + i(-b\boldsymbol{u} + a\boldsymbol{v})$$

Then we can construct matrix $P$

$$P = [\boldsymbol{u} \mid \boldsymbol{v}] = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

thus

$$AP = [A\boldsymbol{u} \mid A\boldsymbol{v}] = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = [a\boldsymbol{u} + b\boldsymbol{v} \mid -b\boldsymbol{u} + a\boldsymbol{v}]$$

$$[a\boldsymbol{u} + b\boldsymbol{v} \mid -b\boldsymbol{u} + a\boldsymbol{v}] = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a & -b \\ b & a \end{bmatrix} = PM$$

(b) We will prove that $P$ is invertible, by contradiction we assume that $P$ is not invertible. If $P$ is not invertible, then one of its column vectors is a real scalar multiple of the other

$$\boldsymbol{v} = c\boldsymbol{u}$$

We substitute into the equations

$$\begin{aligned} A\boldsymbol{u} &= a\boldsymbol{u} + b\boldsymbol{v} \\ A\boldsymbol{v} &= -b\boldsymbol{u} + a\boldsymbol{v} \end{aligned}$$

they will become

$$\begin{aligned} A\boldsymbol{u} &= a\boldsymbol{u} + b(c\boldsymbol{u}) = (a + bc)\boldsymbol{u} \\ A\boldsymbol{v} &= -b\boldsymbol{u} + a(c\boldsymbol{u}) = (ac - b)\boldsymbol{u} \end{aligned}$$

now we substitute $\boldsymbol{v} = c\boldsymbol{u}$ into $A\boldsymbol{v}$

$$\begin{aligned} A\boldsymbol{v} &= (ac - b)\boldsymbol{u} \\ A(c\boldsymbol{u}) &= (ac - b)\boldsymbol{u} \\ cA\boldsymbol{u} &= (ac - b)\boldsymbol{u} \\ c((a + bc)\boldsymbol{u}) &= (ac - b)\boldsymbol{u} \end{aligned}$$

so

$$\begin{aligned} c(a + bc) &= ac - b \\ bc^2 &= -b \\ bc^2 + b &= 0 \\ (1 + c^2)b &= 0 \end{aligned}$$

from the definition of complex eigenvalue we know that $b \neq 0$, so

$$c = \sqrt{-1} = i$$

it means that $c$ is a complex number, a contradiction, hence $P$ is invertible (if $P$ is not invertible then $c$ has to be a real number).

[DF*] **Example:**

(a) Prove: if $k$ is a complex number, and $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{C}^n$, then

$$(\boldsymbol{u} - k\boldsymbol{v}) \cdot (\boldsymbol{u} - k\boldsymbol{v}) = \boldsymbol{u} \cdot \boldsymbol{u} - \bar{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k\overline{(\boldsymbol{u} \cdot \boldsymbol{v})} + k\bar{k}(\boldsymbol{v} \cdot \boldsymbol{v})$$

(b) Use the result in part (a) to prove that

$$0 \leq \boldsymbol{u} \cdot \boldsymbol{u} - \bar{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k\overline{(\boldsymbol{u} \cdot \boldsymbol{v})} + k\bar{k}(\boldsymbol{v} \cdot \boldsymbol{v})$$

(c) Take $k = (\boldsymbol{u} \cdot \boldsymbol{v})/(\boldsymbol{v} \cdot \boldsymbol{v})$ in part (b) to prove that

$$|\boldsymbol{u} \cdot \boldsymbol{v}| \leq ||\boldsymbol{u}||\ ||\boldsymbol{v}||$$

**Solution:**

(a) Let $k$ be a complex number, and $\boldsymbol{u}$ and $\boldsymbol{v}$ be vectors in $\mathbb{C}^n$

$$\begin{aligned} k &= c + di \\ \boldsymbol{u} &= (a_1 + b_1 i, a_2 + b_2 i, \ldots, a_n + b_n i) \\ \boldsymbol{v} &= (c_1 + d_1 i, c_2 + d_2 i, \ldots, c_n + d_n i) \end{aligned}$$

with $c, d, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_n \in \mathbb{R}$, thus we will use the theorem to compute the dot product and make it into simpler form

$$\begin{aligned} (\boldsymbol{u} - k\boldsymbol{v}) \cdot (\boldsymbol{u} - k\boldsymbol{v}) &= \boldsymbol{u} \cdot \boldsymbol{u} - \boldsymbol{u} \cdot (k\boldsymbol{v}) - (k\boldsymbol{v}) \cdot \boldsymbol{u} + (k\boldsymbol{v})(k\boldsymbol{v}) \\ &= \boldsymbol{u} \cdot \boldsymbol{u} - \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k(\overline{\boldsymbol{u} \cdot \boldsymbol{v}}) + k\overline{k}(\boldsymbol{v} \cdot \boldsymbol{v}) \end{aligned}$$

(b) To prove that

$$0 \leq \boldsymbol{u} \cdot \boldsymbol{u} - \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k(\overline{\boldsymbol{u} \cdot \boldsymbol{v}}) + k\overline{k}(\boldsymbol{v} \cdot \boldsymbol{v})$$

with $k$ is a complex number we assume to be in the form of $c + di$, $c, d \in \mathbb{R}$.

We will extract the right hand side equation one by one, based on theorem we already know that

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{u} &\geq 0 \\ \boldsymbol{v} \cdot \boldsymbol{v} &\geq 0 \\ k\overline{k}(\boldsymbol{v} \cdot \boldsymbol{v}) &= (c + di)(c - di)(\boldsymbol{v} \cdot \boldsymbol{v}) \\ k\overline{k}(\boldsymbol{v} \cdot \boldsymbol{v}) &= (c^2 + d^2)(\boldsymbol{v} \cdot \boldsymbol{v}) \geq 0 \end{aligned}$$

We assume the general form for the result of the dot product $(\boldsymbol{u} \cdot \boldsymbol{v}) = a + bi$, $a, b \in \mathbb{R}$ thus

$$\begin{aligned} \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) &= (c - di)(a + bi) \\ &= ac + bci - adi + bd \\ &= (ac + bd) + (bc - ad)i \end{aligned}$$

$$\begin{aligned} k\overline{(\boldsymbol{u} \cdot \boldsymbol{v})} &= (c + di)(a - bi) \\ &= ac - bci + adi + bd \\ &= (ac + bd) + (ad - bc)i \end{aligned}$$

hence

$$\begin{aligned} -\overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k\overline{(\boldsymbol{u} \cdot \boldsymbol{v})} &= -(ac + bd) + (bc - ad)i - (ac + bd) + (ad - bc)i \\ &= -2(ac + bd) \end{aligned}$$

We substitute all the computations above to the inequality

$$\begin{aligned} 0 &\leq \boldsymbol{u} \cdot \boldsymbol{u} - 2(ac + bd) + (c^2 + d^2)(\boldsymbol{v} \cdot \boldsymbol{v}) \\ 2(ac + bd) &\leq \boldsymbol{u} \cdot \boldsymbol{u} + (c^2 + d^2)(\boldsymbol{v} \cdot \boldsymbol{v}) \end{aligned}$$

the condition needs to be fulfilled is that $2(ac + bd) \leq \boldsymbol{u} \cdot \boldsymbol{u} + (c^2 + d^2)(\boldsymbol{v} \cdot \boldsymbol{v})$ so the inequality can be proven.

Remember again that

$$(\boldsymbol{u} - k\boldsymbol{v}) \cdot (\boldsymbol{u} - k\boldsymbol{v}) \geq 0$$

since $\boldsymbol{u} - k\boldsymbol{v} \in \mathbb{C}^n$ and based on definition

$$(\boldsymbol{u} - k\boldsymbol{v}) \cdot (\boldsymbol{u} - k\boldsymbol{v}) = ||(\boldsymbol{u} - k\boldsymbol{v})||^2 \geq 0$$

(c) Let $k = (\boldsymbol{u} \cdot \boldsymbol{v})/(\boldsymbol{v} \cdot \boldsymbol{v})$, then

$$\begin{aligned}
0 &\leq \boldsymbol{u} \cdot \boldsymbol{u} - \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k\overline{(\boldsymbol{u} \cdot \boldsymbol{v})} + k\overline{k}(\boldsymbol{v} \cdot \boldsymbol{v}) \\
0 &\leq ||\boldsymbol{u}||^2 - \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k(\boldsymbol{v} \cdot \boldsymbol{u}) + k\frac{(\boldsymbol{v} \cdot \boldsymbol{u})}{(\boldsymbol{v} \cdot \boldsymbol{v})}\boldsymbol{v} \cdot \boldsymbol{v} \\
0 &\leq ||\boldsymbol{u}||^2 - \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) - k(\boldsymbol{v} \cdot \boldsymbol{u}) + \overline{k}(\boldsymbol{u} \cdot \boldsymbol{v}) \\
0 &\leq ||\boldsymbol{u}||^2 - k(\boldsymbol{v} \cdot \boldsymbol{u}) \\
k(\boldsymbol{v} \cdot \boldsymbol{u}) &\leq ||\boldsymbol{u}||^2 \\
\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\boldsymbol{v} \cdot \boldsymbol{v}}(\boldsymbol{v} \cdot \boldsymbol{u}) &\leq ||\boldsymbol{u}||^2 \\
(\boldsymbol{u} \cdot \boldsymbol{v})(\boldsymbol{v} \cdot \boldsymbol{u}) &\leq ||\boldsymbol{u}||^2(\boldsymbol{v} \cdot \boldsymbol{v}) \\
|\boldsymbol{u} \cdot \boldsymbol{v}|^2 &\leq ||\boldsymbol{u}||^2||\boldsymbol{v}||^2 \\
|\boldsymbol{u} \cdot \boldsymbol{v}| &\leq ||\boldsymbol{u}||||\boldsymbol{v}||
\end{aligned}$$

it is proven that $|\boldsymbol{u} \cdot \boldsymbol{v}| \leq ||\boldsymbol{u}||||\boldsymbol{v}||$, if you have doubt, read again on the theorems and definitions, always remember that $\overline{(\boldsymbol{v} \cdot \boldsymbol{v})} = \boldsymbol{v} \cdot \boldsymbol{v}$, then the general result for $(\boldsymbol{u} \cdot \boldsymbol{v}) = a + bi$ and $(\boldsymbol{v} \cdot \boldsymbol{u}) = a - bi$ thus

$$\begin{aligned}
(\boldsymbol{u} \cdot \boldsymbol{v})(\boldsymbol{v} \cdot \boldsymbol{u}) &= (a + bi)(a - bi) \\
&= a^2 + b^2 = |\boldsymbol{u} \cdot \boldsymbol{v}|^2
\end{aligned}$$

**Computational Guide 23.6: Matrix Factorization Using Complex Eigenvalues**

Suppose we have a matrix $A$ of size $2 \times 2$

$$A = \begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}$$

To be able to obtain the factorization, matrix $C, P$ and $P^{-1}$ the procedures are as follow:

1. Compute the characteristic polynomial of $A$

$$\det(\lambda I - A) = \lambda^2 + 1$$

2. We obtain the eigenvalues which are

$$\lambda_1 = i$$
$$\lambda_2 = -i$$

The theorem stated that we need to use this form of eigenvalue $\lambda = a - bi$ to determine matrix $P$. So we will choose $\lambda_2 = -i$ with

$$a = 0, \quad b = 1$$

3. Examine the matrix

$$(\lambda I - A)$$

To find the eigenvectors we need to solve the system

$$\begin{bmatrix} \lambda + 2 & 1 \\ -5 & \lambda - 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

we substitute the value of $\lambda = \lambda_2$ then we perform Gauss-Jordan elimination to obtain the row echelon form.

A nice trick is we know that the reduced row echelon form of matrix $(\lambda I - A)$ will have row of zeros, thus we can just assume that each row is a scalar multiple of the other, and hence we can make the first row into a row of zeros.

4. After we make the first row into row of zeros we will obtain for $\lambda = -i$

$$(\lambda_2 I - A) = \begin{bmatrix} 0 & 0 & 0 \\ -5 & -i - 2 & 0 \end{bmatrix}$$

By interchanging the first row and the second row, then divide the first row by $(-5)$ we will obtain

$$(1)x_1 + \left(\frac{2}{5} + \frac{i}{5}\right) x_2 = 0$$

5. With parameterization, we will obtain the general solution of the system as

$$x_1 = \left(-\frac{2}{5} - \frac{i}{5}\right) t$$
$$x_2 = t$$

**Computational Guide 23.7: Matrix Factorization Using Complex Eigenvalues**

6. Now we have obtained the vectors

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -\frac{2}{5} \\ 1 \end{bmatrix} t + \begin{bmatrix} -\frac{1}{5} \\ 0 \end{bmatrix} ti$$

thus

$$\mathrm{Re}(\boldsymbol{x}) = \begin{bmatrix} -\frac{2}{5} \\ 1 \end{bmatrix}, \quad \mathrm{Im}(\boldsymbol{x}) = \begin{bmatrix} -\frac{1}{5} \\ 0 \end{bmatrix}$$

and we will obtain the matrix $P$ corresponding to $\lambda = -i$ as

$$P = [\mathrm{Re}(\boldsymbol{x}) \quad \mathrm{Im}(\boldsymbol{x})] \begin{bmatrix} -\frac{2}{5} & -\frac{1}{5} \\ 1 & 0 \end{bmatrix}$$

we can easily compute the inverse for $P$

$$P^{-1} = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}$$

7. Remember that by choosing the eigenvalue $\lambda = \lambda_2 = -i$, we will have $a = 0$ and $b = 1$ as a result we will obtain matrix $C$ as

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

8. So matrix $A$ can be factored as

$$A = PCP^{-1}$$

$$\begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} -\frac{2}{5} & -\frac{1}{5} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}$$

## i. C++ Computation: Complex Conjugate, Real and Imaginary Parts of Vectors and Matrices

Let

$$v = (3+i, -2i, 5)$$

and

$$A = \begin{bmatrix} 1+i & -i \\ 4 & 6-2i \end{bmatrix}$$

Then

$$\begin{aligned} \overline{v} &= (3-i, 2i, 5) \\ \mathrm{Re}(v) &= (3, 0, 5) \\ \mathrm{Im}(v) &= (1, -2, 0) \end{aligned}$$

$$\overline{A} = \begin{bmatrix} 1-i & i \\ 4 & 6+2i \end{bmatrix}$$

$$\mathrm{Re}(A) = \begin{bmatrix} 1 & 0 \\ 4 & 6 \end{bmatrix}, \quad \mathrm{Im}(A) = \begin{bmatrix} 1 & -1 \\ 0 & -2 \end{bmatrix}$$

$$\det(A) = \begin{vmatrix} 1+i & -i \\ 4 & 6-2i \end{vmatrix} = (1+i)(6-2i) - (-i)(4) = 8+8i$$

The C++ code is using **Armadillo** library, we take the vector and matrix from textfiles, since it is complex vector and complex matrix, thus for constructing a complex vector we will need two real vectors that will act as the real part and imaginary part, the same goes for complex matrix that consist of two real matrices, one as the real part and the other as the imaginary part.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat Areal;
       mat Acomplex;
       vec vreal;
       vec vcomplex;
       Areal.load("matrixAreal.txt");
       Acomplex.load("matrixAcomplex.txt");
       vreal.load("vectorvreal.txt");
       vcomplex.load("vectorvcomplex.txt");
       cx_mat A(Areal, Acomplex); // To directly construct a complex matrix
            out of two real matrices
       cx_mat complexconjugateA(Areal, -Acomplex);
```

```
        cx_mat eigvec;
        cx_vec eigval;
        cx_vec v(vreal,vcomplex);

        mat I(2,2,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;
        cout <<"Re(A):" << "\n" << real(A) <<endl;
        cout <<"Im(A):" << "\n" << imag(A) <<endl;
        cout <<"Complex conjugate of A:" << "\n" << complexconjugateA <<endl
            ;
        //cout <<"Complex conjugate of A:" << "\n" << (A.t()).st() <<endl;
        cout <<"det(A):" << "\n" << det(A) <<endl;
        cout << endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
               cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                   imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        cout <<"Vector v:" << "\n" << v <<endl;
        cout <<"Re(v):" << "\n" << real(v) <<endl;
        cout <<"Im(v):" << "\n" << imag(v) <<endl;
        cout <<"Complex conjugate of v:" << "\n" << (v.t()).st() <<endl;

        return 0;
    }
```

**C++ Code 180:** *main.cpp "Complex conjugate real and imaginary parts of vectors and matrices"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- This is how we construct a complex vector with **cx_vec** from two vectors **(vreal, vcomplex)** and to construct a complex matrix $A$ from two matrices **Areal, Acomplex**.

```
mat Areal;
mat Acomplex;
vec vreal;
vec vcomplex;
Areal.load("matrixAreal.txt");
Acomplex.load("matrixAcomplex.txt");
vreal.load("vectorvreal.txt");
vcomplex.load("vectorvcomplex.txt");
cx_mat A(Areal, Acomplex); // To directly construct a complex
    matrix out of two real matrices
cx_mat complexconjugateA(Areal, -Acomplex);
cx_mat eigvec;
cx_vec eigval;
cx_vec v(vreal,vcomplex);
```

There are two known ways to compute complex conjugate the first one is with defining a new complex matrix with the negative of the original imaginary part:

**cx_mat complexconjugateA(Areal, -Acomplex)**

The other way is to do Hermitian transpose (**.t()**) that will flip the sign of the imaginary part but it will also transpose the matrix, thus we will perform another simple transpose (**.st()**) :

**(A.t()).st()**

```
Matrix A:
    (+1.000e+00,+1.000e+00)    (+0.000e+00,-1.000e+00)
    (+4.000e+00,+0.000e+00)    (+6.000e+00,-2.000e+00)

Re(A):
   1.0000        0
   4.0000   6.0000

Im(A):
   1.0000  -1.0000
        0  -2.0000

Complex conjugate of A:
    (+1.000e+00,-1.000e+00)    (+0.000e+00,+1.000e+00)
    (+4.000e+00,-0.000e+00)    (+6.000e+00,+2.000e+00)

det(A):
(8,8)

Eigenvalues:
λ_1 = 0.656077 + 1.52186i
λ_2 = 6.34392 + -2.52186i

Vector v:
    (+3.000e+00,+1.000e+00)
    (+0.000e+00,-2.000e+00)
    (+5.000e+00,+0.000e+00)

Re(v):
   3.0000
        0
   5.0000

Im(v):
   1.0000
  -2.0000
        0

Complex conjugate of v:
    (+3.000e+00,-1.000e+00)
    (+0.000e+00,+2.000e+00)
    (+5.000e+00,-0.000e+00)
```

**Figure 23.125:** *The computation to find the complex conjugate, real and imaginary parts of vector* $\boldsymbol{v}$ *and matrix* $A$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Complex Conjugate, Real and Imaginary Parts with Armadillo/main.cpp**).*

## ii. C++ Computation: Complex Euclidean Inner Product

Let

$$\boldsymbol{u} = (1+i, i, 3-i)$$

and

$$\boldsymbol{v} = (1+i, 2, 4i)$$

Find $\boldsymbol{u} \cdot \boldsymbol{v}$, $\boldsymbol{v} \cdot \boldsymbol{u}$, $||\boldsymbol{u}||$, $||\boldsymbol{v}||$.

**Solution:**

$$\begin{aligned}
\boldsymbol{u} \cdot \boldsymbol{v} &= (1+i)(1-i) + i(2) + (3-i)(-4i) = -2 - 10i \\
\boldsymbol{v} \cdot \boldsymbol{u} &= (1+i)(1-i) + 2(-i) + (4i)(3+i) = -2 + 10i \\
||\boldsymbol{u}|| &= \sqrt{|1+i|^2 + |i|^2 + |3-i|^2} = \sqrt{13} = 3.60555 \\
||\boldsymbol{v}|| &= \sqrt{|1+i|^2 + |2|^2 + |4i|^2} = \sqrt{22} = 4.69042
\end{aligned}$$

Remember that $|z| = |a+bi| = \sqrt{a^2+b^2}$.

The C++ code is using **Armadillo** library and the complex vector $\boldsymbol{u}$ consists of two real vectors that are taken from textfiles **vectorureal.txt** and **vectorucomplex.txt**, the complex vector $\boldsymbol{v}$ also consists of two real vectors that are taken from textfiles **vectorvreal.txt** and **vectorvcomplex.txt**

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{

       vec vreal, vcomplex, ureal, ucomplex;
       vreal.load("vectorvreal.txt");
       vcomplex.load("vectorvcomplex.txt");
       ureal.load("vectorureal.txt");
       ucomplex.load("vectorucomplex.txt");
       cx_vec v(vreal,vcomplex);
       cx_vec u(ureal,ucomplex);

       cout <<"Vector v:" << "\n" << v <<endl;
       cout << endl;
       cout <<"Vector u:" << "\n" << u <<endl;
       cout << endl;
```

```
        cout <<"u . v :" << "\n" << dot(u,conj(v)) <<endl;
        cout << endl;
        cout <<"v . u :" << "\n" << dot(v,conj(u)) <<endl;
        cout << endl;

        cout <<"|| v || :" << "\n" << norm(v) <<endl;
        cout << endl;
        cout <<"|| u || :" << "\n" << norm(u) <<endl;

        cout <<"| 1+i | :" << "\n" << u[0] <<endl;
        cout <<"| 1+i | :" << "\n" << abs(u[0]) <<endl;
        cout <<"| 1 | :" << "\n" << abs(1) <<endl;
        cout <<"| -1 | :" << "\n" << abs(-1) <<endl;

        cout <<"| i | :" << "\n" << abs(u[1]) <<endl;
        cout <<"| i |^{2} :" << "\n" << abs(u[1])*abs(u[1]) <<endl;
        cout <<"| - i | :" << "\n" << abs(-u[1]) <<endl;

        cout <<"| 3 - i | :" << "\n" << abs(u[2]) <<endl;

        return 0;
    }
```

**C++ Code 181:** *main.cpp "Complex euclidean inner product and norm"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Vector v:
    (+1.000e+00,+1.000e+00)
    (+2.000e+00,+0.000e+00)
    (+0.000e+00,+4.000e+00)


Vector u:
    (+1.000e+00,+1.000e+00)
    (+0.000e+00,+1.000e+00)
    (+3.000e+00,-1.000e+00)


u . v :
(-2,-10)

v . u :
(-2,10)

|| v || :
4.69042

|| u || :
3.60555
```

**Figure 23.126:** *The computation to find the complex Euclidean inner product and norm of vectors* $\boldsymbol{u}$ *and* $\boldsymbol{v}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Complex Euclidean Inner Product with Armadillo/main.cpp**).*

## iii. C++ Plot and Computation: Geometric Interpretation of Complex Eigenvalues

We will see the geometric interpretation when a vector $\boldsymbol{x}$ is multiplied by matrix with real entries

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

with eigenvalues $\lambda = a \pm bi$. If $a$ and $b$ are not both zero, then this matrix can be factored as

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

where $\theta$ is the angle from the positive $x$-axis to the ray that joins the origin to the point $(a, b)$.

Now we want to give an input of:

1. Vector $\boldsymbol{x}$
2. Matrix $C$ with the entries $a$ and $b$ are defined by us

The computation will produce:

1. $|\lambda|$ with
$$|\lambda| = \sqrt{a^2 + b^2}$$
2. $\theta$ with
$$\theta = \cos^{-1}\left(\frac{a}{|\lambda|}\right) = \sin^{-1}\left(\frac{b}{|\lambda|}\right)$$

then we can plot:

1. The vector $\boldsymbol{x}$:
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

2. The counterclockwise rotated vector $\boldsymbol{x}$ with angle of $\theta$ that is

$$R_\theta \boldsymbol{x} = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

3. The scaled counterclockwise rotated vector $\boldsymbol{x}$ with angle of $\theta$, that is

$$(|\lambda| I)(R_\theta)\boldsymbol{x} = \begin{bmatrix} |\lambda| & 0 \\ 0 & |\lambda| \end{bmatrix} \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We use **Armadillo** library to take vector $\boldsymbol{x}$ from textfile input, then to perform the computation fast and smooth, then for the plotting part we use **Gnuplot**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	X.load("vectorX.txt");
```

```
cout <<"Vector x:" << "\n" << X <<endl;

float a = 1.2;
float b = 1.3;
float l = sqrt(pow(a,2)+pow(b,2));
// theta is in radian
float theta = acos(a/l); // acos(a/l) = asin(b/l)

// Create matrix C
float matrixC[N][N] = {};
// Create standard matrix for counterclockwise rotation
float matrixRotation[N][N] = {};
matrixRotation[0][0] = cos(theta);
matrixRotation[0][1] = -sin(theta);
matrixRotation[1][0] = sin(theta);
matrixRotation[1][1] = cos(theta);
// Create matrix for scaling
float matrixScaling[N][N] = {};

for (int i = 0; i < N; ++i)
{
	for(int j = 0; j<N; ++j)
	{
		if (i == j)
		{
			matrixC[i][j] = a;
			matrixScaling[i][j] = l;
		}
		else
		{
			matrixC[i][j] = b;
			matrixScaling[i][j] = 0;
		}
	}
}
matrixC[0][1] = -b;

arma::mat ArmaScaling(N,N,fill::zeros);
arma::mat ArmaRotation(N,N,fill::zeros);
arma::mat ArmaC(N,N,fill::zeros);

for (int i = 0; i < N; ++i)
{
	for(int j = 0; j<N; ++j)
	{
		ArmaScaling[i+j*N] = matrixScaling[i][j] ;
		ArmaRotation[i+j*N] = matrixRotation[i][j] ;
		ArmaC[i+j*N] = matrixC[i][j] ;
```

```
        }
    }

    cout <<"Matrix C:" << "\n" << ArmaC <<endl;
    cout <<"Standard Matrix for scaling:" << "\n" << ArmaScaling <<endl;
    cout <<"Standard Matrix for counterclockwise rotation:" << "\n" <<
        ArmaRotation <<endl;

    arma::mat Xrotated = ArmaRotation*X ;
    arma::mat Xscaled = ArmaScaling*Xrotated ;
    arma::mat CX = ArmaC*X ;
    cout <<"C*x:" << "\n" << CX <<endl;
    cout <<"Vector x rotated with angle of " << theta*RADTODEG << " :"
        << "\n" << Xrotated <<endl;
    cout <<"Rotated vector x is being scaled by |l|:" << "\n" << Xscaled
         <<endl;

    // We use a separate container for each column, like so:
    std::vector<double> pts_C_x;
    std::vector<double> pts_C_y;
    std::vector<double> pts_C_dx;
    std::vector<double> pts_C_dy;
    std::vector<double> pts_D_x;
    std::vector<double> pts_D_y;
    std::vector<double> pts_D_dx;
    std::vector<double> pts_D_dy;
    std::vector<double> pts_E_x;
    std::vector<double> pts_E_y;
    std::vector<double> pts_E_dx;
    std::vector<double> pts_E_dy;
    std::vector<double> pts_F_x;
    std::vector<double> pts_F_y;
    std::vector<double> pts_F_dx;
    std::vector<double> pts_F_dy;
    float o = 0;

    // Create a vector x
    pts_C_x .push_back(o);
    pts_C_y .push_back(o);
    pts_C_dx.push_back(X[0]);
    pts_C_dy.push_back(X[1]);
    // The rotated vector x
    pts_D_x .push_back(o);
    pts_D_y .push_back(o);
    pts_D_dx.push_back(Xrotated[0]);
    pts_D_dy.push_back(Xrotated[1]);
    // The scaled vector x after rotated
    pts_E_x .push_back(o);
```

```
            pts_E_y .push_back(o);
            pts_E_dx.push_back(Xscaled[0]);
            pts_E_dy.push_back(Xscaled[1]);
            // The scaled vector x after rotated with C
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_dx.push_back(CX[0]);
            pts_F_dy.push_back(CX[1]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set xrange [0:2.5]\nset yrange [0:2.5]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "plot '-' with vectors title 'x','-' with vectors title '
                Rotated x' lc 'blue', '-' with vectors title 'Scaled rotated x'
                 lc 'green'\n";
            //gp << "plot '-' with vectors title 'x','-' with vectors title '
                Rotated x' lc 'blue', '-' with vectors title 'Scaled rotated x'
                 lc 'green', '-' with vectors title 'Cx' lc 'black'\n";
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
            //gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy))
                ;
        }
```

**C++ Code 182:** *main.cpp "Geometric interpretation of complex eigenvalues"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- To plot

$$Cx$$

we uncomment and comment some lines to become like this

```
            gp << "set xrange [0:2.5]\nset yrange [0:2.5]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
            // '-' means read from stdin. The send1d() function sends data
                 to gnuplot's stdin.
            // gp << "plot '-' with vectors title 'x','-' with vectors
                title 'Rotated x' lc 'blue', '-' with vectors title '
                Scaled rotated x' lc 'green'\n";
```

```
            gp << "plot '—' with vectors title 'x', '—' with vectors title
                 'Cx' lc 'black'\n";
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx,
                pts_C_dy));
            //gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx,
                pts_D_dy));
            //gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx,
                pts_E_dy));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx,
                pts_F_dy));
```

```
Vector x:
   1.0000
   0.5000

Matrix C:
   1.2000  -1.3000
   1.3000   1.2000

Standard Matrix for scaling:
   1.7692        0
        0   1.7692

Standard Matrix for counterclockwise rotation:
   0.6783  -0.7348
   0.7348   0.6783

C*x:
   0.5500
   1.9000

Vector x rotated  with angle of 47.2906 :
   0.3109
   1.0739

Rotated vector x is being scaled by |λ|:
   0.5500
   1.9000
```

**Figure 23.127:** *The computation to find the transformation after a vector* $\boldsymbol{x}$ *is multiplied by a matrix C that has complex eigenvalues* $\lambda = a \pm bi$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Geometric Interpretation of Complex Eigenvalues/-main.cpp**).*

**Figure 23.128:** *The plot of vector* $\boldsymbol{x}$ *that is rotated with angle of* $\theta$ *then scaled with value of* $|\lambda|$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Geometric Interpretation of Complex Eigenvalues/main.cpp**).*

There is a special case where we let $a = 0$ in matrix $C$, then we will have complex eigenvalues of $\lambda = \pm bi$. This will create the symptom of

$$C^T = -C$$

If you remember on subspaces section we encounter a problem where we have to find the set of all matrices $n \times \ n$ such that $A^T = -A$. Surprisingly in this section we can apply such property and see what will happens for this complex eigenvalues when $C^T = -C$. Turns out, the rotation will be multiplication of $\frac{\pi}{2} + n\pi$, since the angle that will make the value of cosine to be zero ($a = 0$) is $\frac{\pi}{2} + n\pi$ with $n = 0, 1, 2, \ldots$.

**Figure 23.129:** *The plot of vector* $\boldsymbol{x}$ *that is rotated with angle of* $\theta = \frac{\pi}{2}$ *then scaled with value of* $|\lambda|$ *with* $a = 0$ *and* $b = 2.3$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Geometric Interpretation of Complex Eigenvalues/main.cpp**).*

## iv. C++ Computation: Matrix Factorization Using Complex Eigenvalues

Suppose we have a matrix $A$ of size $2 \times 2$ with complex eigenvalues.

$$\begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix}$$

and we want to factorize it so we will obtain this form:

$$A = PCP^{-1}$$

remember the theorem that stated:

If $\boldsymbol{x}$ is an eigenvector of $A$ corresponding to $\lambda = a - bi$, then the matrix

$$P = [\mathrm{Re}(\boldsymbol{x}) \quad \mathrm{Im}(\boldsymbol{x})]$$

is invertible and

$$A = P \begin{bmatrix} a & -b \\ b & a \end{bmatrix} P^{-1}$$

The eigenvalues of $A$ are $\lambda = i$ and $\lambda = -i$.

We choose to use $\lambda = -i$ and we will obtain $a = 0$ and $b = 1$ so

$$C = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Now to find $\mathrm{Re}(\boldsymbol{x})$ and $\mathrm{Im}(\boldsymbol{x})$ that corresponding to $\lambda = -i$, you construct matrix

$$(\lambda I - A)\boldsymbol{x} = \mathbf{0}$$

Then we perform Gauss-Jordan elimination and will obtain by parameterization the general solution:

$$x_1 = \left(-\frac{2}{5} - \frac{i}{5}\right) t$$
$$x_2 = t$$

thus

$$\mathrm{Re}(\boldsymbol{x}) = \begin{bmatrix} -\frac{2}{5} \\ 1 \end{bmatrix}, \quad \mathrm{Im}(\boldsymbol{x}) = \begin{bmatrix} -\frac{1}{5} \\ 0 \end{bmatrix}$$

and we will obtain the matrix $P$ corresponding to $\lambda = -i$ as

$$P = [\mathrm{Re}(\boldsymbol{x}) \quad \mathrm{Im}(\boldsymbol{x})] = \begin{bmatrix} -\frac{2}{5} & -\frac{1}{5} \\ 1 & 0 \end{bmatrix}$$

we can easily compute the inverse for $P$

$$P^{-1} = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}$$

All the matrices needed to perform matrix factorization have been computed. We can check now and verify that

$$A = PCP^{-1}$$

$$\begin{bmatrix} -2 & -1 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} -\frac{2}{5} & -\frac{1}{5} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}$$

The code is using **Armadillo** and there are some important things needed to be known:

1. First, we will need to know the eigenvalues of matrix $A$ then choose one of the eigenvalues and use the form of
$$\lambda = a - bi$$
so we can determine the values for $a$ and $b$ to construct matrix $C$.
For example, for $\lambda = i$, we will have $a = 0$ and $b = -1$. For $\lambda = -i$, we will have $a = 0$ and $b = 1$.

2. The next step is constructing matrix $P$ that corresponds to the eigenvalue that we have chosen. Afterwards, computing the $P^{-1}$ won't be too hard for matrix with dimension of 2.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

#define N 2 // Define the number of dimension

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);

       float a = 0;
       float b = 1;
       float matrixC[N][N] = {};
       matrixC[0][0] = a;
       matrixC[0][1] = -b;
       matrixC[1][0] = b;
       matrixC[1][1] = a;

       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix
```

```
cout << "Eigenvalues:" << endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

cout << endl;
cout <<"Matrix l_1I - A:" << "\n" << I*eigval[0] - A <<endl;
cout <<"Matrix l_2I - A:" << "\n" << I*eigval[1] - A <<endl;
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
cout << endl;

cout << "Re(x):" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

cout << "Im(x):" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << imag(eigvec.col(i-1)) <<
           endl;
}

arma::mat ArmaC(N,N,fill::zeros);

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaC[i+j*N] = matrixC[i][j] ;
       }
}

cout <<"For l = -i" <<endl;
cout << endl;
cout <<"Matrix C:" << "\n" << ArmaC <<endl;

// For l = -i, the second eigenvalue
arma::mat P(N,N,fill::zeros);
P.col(0) = real(eigvec.col(1)) ; // make Re(x) the first column for
```

```
                matrix P
            P.col(1) = imag(eigvec.col(1)) ; // make Im(x) the second column for
                 matrix P
            cout <<"Matrix P:" << "\n" << P <<endl;

            arma::mat P_inv(N,N,fill::zeros);
            P_inv = inv(P);
            cout <<"Matrix P^{-1}:" << "\n" << P_inv <<endl;

            cout <<"Matrix P * C * P^{-1}:" << "\n" << P*ArmaC*P_inv <<endl;

            return 0;
        }
```

**C++ Code 183:** *main.cpp "Matrix factorization using complex eigenvalues"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- To construct matrix $P$, we will take the real part of eigenvector $\boldsymbol{x}$ corresponds to $\lambda = -i$ with function **real(eigvec.col(1))** as the first column for matrix P.

  Then, we will take the imaginary part of eigenvector $\boldsymbol{x}$ corresponds to $\lambda = -i$ with function **imag(eigvec.col(1))** as the second column for matrix P.
  The number $(1)$ on the functions is to pointing to the second eigenvalue of matrix $A$, which is $\lambda = -i$.

```
            arma::mat P(N,N,fill::zeros);
            P.col(0) = real(eigvec.col(1)) ; // make Re(x) the first column
                 for matrix P
            P.col(1) = imag(eigvec.col(1)) ; // make Im(x) the second
                column for matrix P
            cout <<"Matrix P:" << "\n" << P <<endl;
```

```
Matrix A:
  -2.0000  -1.0000
   5.0000   2.0000

Eigenvalues:
λ_1 = -6.93889e-17 + 1i
λ_2 = -6.93889e-17 + -1i

Matrix λ_1I - A:
    (+2.000e+00,+1.000e+00)    (+1.000e+00,+0.000e+00)
    (-5.000e+00,+0.000e+00)    (-2.000e+00,+1.000e+00)

Matrix λ_2I - A:
    (+2.000e+00,-1.000e+00)    (+1.000e+00,-0.000e+00)
    (-5.000e+00,-0.000e+00)    (-2.000e+00,-1.000e+00)


Re(x):
x[λ_1] :
   -0.3651
   0.9129

x[λ_2] :
   -0.3651
   0.9129

Im(x):
x[λ_1] :
    0.1826
         0

x[λ_2] :
   -0.1826
         0

For λ = -i

Matrix C:
         0  -1.0000
   1.0000         0

Matrix P:
  -0.3651  -0.1826
   0.9129         0

Matrix P^{-1}:
         0   1.0954
  -5.4772  -2.1909

Matrix P * C * P^{-1}:
  -2.0000  -1.0000
   5.0000   2.0000
```

**Figure 23.130:** *The computation for matrix factorization using complex eigenvalues and to prove that $A = PCP^{-1}$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix Factorization Using Complex Eigenvalues/main.cpp**).*

## v. C++ Plot and Computation: Power Sequences of a Matrix Transformation on a Vector

We will see how a successive applciations of a matrix transformation affect a specific vector. For example, if $A$ is the standard matrix for an operator on $\mathbb{R}^n$ and $\boldsymbol{x}_0$ is some fixed vector in $\mathbb{R}^n$, then one might be interested in the behavior of the power sequence

$$\boldsymbol{x}_0, \quad A\boldsymbol{x}_0, \quad A^2\boldsymbol{x}_0, \ldots, \quad A^k\boldsymbol{x}_0, \ldots$$

For example, if

$$A = \begin{bmatrix} \frac{1}{2} & \frac{3}{4} \\ -\frac{3}{5} & \frac{11}{10} \end{bmatrix}, \quad \boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

the first four terms in the power sequence are

$$\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad A\boldsymbol{x}_0 = \begin{bmatrix} 1.25 \\ 0.5 \end{bmatrix}, \quad A^2\boldsymbol{x}_0 = \begin{bmatrix} 1.0 \\ -0.2 \end{bmatrix}, \quad A^3\boldsymbol{x}_0 = \begin{bmatrix} 0.35 \\ -0.82 \end{bmatrix}$$

If we plot the first 15 terms as ordered pairs $(x, y)$, then the points move along the elliptical path.

To understand why the points move along an elliptical path, we will need to examine the eigenvalues and eigenvectors of $A$.

The eigenvalues of $A$ are

$$\lambda = \frac{4}{5} \pm \frac{3}{5}i$$

and the corresponding eigenvectors are

$$\lambda_1 = \frac{4}{5} - \frac{3}{5}i$$

$$\boldsymbol{x}_1 = \begin{bmatrix} 0.7454 \\ 0.2981 - 0.5963i \end{bmatrix}$$

now for the second eigenvalue

$$\lambda_2 = \frac{4}{5} + \frac{3}{5}i$$

$$\boldsymbol{x}_1 = \begin{bmatrix} 0.7454 \\ 0.2981 + 0.5963i \end{bmatrix}$$

If we take the eigenvalue $\lambda = \frac{4}{5} - \frac{3}{5}i$ we will have

$$\begin{aligned} a &= \frac{4}{5} \\ b &= \frac{3}{5} \\ |\lambda| &= \sqrt{a^2 + b^2} = 1 \end{aligned}$$

then we obtain the factorization

$$\begin{aligned} A &= PCP^{-1} \\ &= PSR_\theta P^{-1} \end{aligned}$$

$$\begin{bmatrix} \frac{1}{2} & \frac{3}{4} \\ -\frac{3}{5} & \frac{11}{10} \end{bmatrix} = \begin{bmatrix} 0.7454 & 0 \\ 0.2981 & -0.5963 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{4}{5} & -\frac{3}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix} \begin{bmatrix} 1.3416 & 0 \\ 0.6708 & -1.6771 \end{bmatrix}$$

where $R_\theta$ is a rotation matrix about the origin through the angle $\theta$ whose tangent is

$$\begin{aligned} \tan\ \theta &= \frac{\sin\ \theta}{\cos\ \theta} \\ &= \frac{3/5}{4/5} \\ &= \frac{3}{4} \\ \theta &= \tan^{-1}\left(\frac{3}{4}\right) = 36.9^0 \end{aligned}$$

The matrix $P$ is the transition matrix from the basis

$$B = \{\mathrm{Re}(\boldsymbol{x}),\ \mathrm{Im}(\boldsymbol{x}) = \{(0.7454, 0.2981), (0, -0.5963)\}$$

to the standard basis, and $P^{-1}$ is the transition matrix from the standard basis to the basis $B$.

**Figure 23.131:** *Matrix $P$ is the transition matrix from the old basis $B = \{Re(\boldsymbol{x}),\ Im(\boldsymbol{x}) = \{(0.7454, 0.2981), (0, -0.5963)\}$ (above) to a new basis, the standard basis $B' = \{(1,0), (0,1)\}$.*

Next, observe that if $n$ is a positive integer, then

$$A^n \boldsymbol{x}_0 = (PSR_\theta P^{-1})^n \boldsymbol{x}_0 = PS^n R_\theta^n P^{-1} \boldsymbol{x}_0$$

so the product $A^n \boldsymbol{x}_0$ can be computed by first mapping $\boldsymbol{x}_0$ into the point $P^{-1}\boldsymbol{x}_0$ in $B$-coordinates, then multiplying by $R_\theta^n$ to rotate this point about the origin through the angle $n\theta$, then multiplying by $S^n$ to scale the rotated vector, and then multiplying $S^n R_\theta^n P^{-1}\boldsymbol{x}_0$ by $P$ to map the resulting point back to standard coordinates.

We can now see what is happening geometrically:
In $B$-coordinates each successive multiplication by $A$ causes the point $P^{-1}\boldsymbol{x}_0$ to advance through an angle $\theta$, thereby tracing a circular orbit about the origin. The scaling ($S^n$) is not making the vector larger or smaller, since $|\lambda| = 1$, the matrix $I = S$, thus the radius of rotation is the same. However, the basis $B$ is skewed (not orthogonal), so when the points on the circular orbit are transformed back to standard coordinates, the effect is to distort the circular orbit into elliptical orbit traced by $A^n \boldsymbol{x}_0$.

The C++ code using **Armadillo** library for the computation and the plotting is using **Gnuplot**.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

#include "gnuplot−iostream.h"

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

using namespace std;
using namespace arma;

double division(double x, double y)
{
       return x/y;
}

int main() {
       Gnuplot gp;
       int N = 2;
       vec x, y, ystdbasis;
       x.load("vectorx.txt");

       float theta = atan(division(3,4))*RADTODEG ;
       //cout <<"Tan(theta) :" << "\n" << tan(theta*DEGTORAD) <<endl;

       arma::mat xstdbasis(N,1,fill::zeros);
       xstdbasis[0] = 1;
```

```
xstdbasis[1] = division(1,2);
arma::mat A(N,N,fill::zeros);
A[0] = division(1,2);
A[1] = division(-3,5);
A[2] = division(3,4);
A[3] = division(11,10);

cout <<"Matrix A :" << "\n" << A <<endl;
cout <<"Vector x :" << "\n" << x <<endl;
cout <<"Vector xstdbasis :" << "\n" << xstdbasis <<endl;
cout <<"Ax:" << "\n" << A*x <<endl;

// Compute eigenvalues
cx_vec eigval;
cx_mat eigvec;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
    square matrix

cout << "Eigenvalues:" << endl;

for (int i = 1; i <= N; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}
cout << endl;
cout <<"Eigenvectors:" << "\n" << eigvec <<endl;

arma::mat B(N,N,fill::zeros);
B[0] = A[0];
B[1] = A[1];
B[2] = A[2];
B[3] = A[3];

int n = 15;
// n = 1 makes B = A^2
// n = 2 makes B = A^3
// n = 3 makes B = A^4
y = A*x;
arma::mat PSA;
// we can also declare a matrix PSA as a matrix with no column / 0
    column and n rows, since we will insert the column later on
//arma::mat PSA(n,0,fill::zeros);
arma::vec Ax1(n,1);
arma::vec Ax2(n,1);
Ax1[0] = y[0];
Ax2[0] = y[1];
```

```
for (int i = 1; i <= n; i++)
{
        B*=A;
        // cout <<"A^{"<< i+1 << "}:" << "\n" << B <<endl;
        // cout <<"A^{"<< i+1 << "}*x:" << "\n" << B*x <<endl;
        y = B*x;
        Ax1[i] = y[0];
        Ax2[i] = y[1];
}

PSA.insert_cols(0, Ax1);
PSA.insert_cols(1, Ax2);

cout << endl;
cout << "Power Sequences of A" << "\n" << PSA<< endl;
// save matrix as a text file
PSA.save("PSAMatrix.txt", raw_ascii);

// Compute P S R_theta and P^{-1}
// We choose l = 0.8 - 0.6i, thus a = 0.8, b = 0.6
float a = 0.8;
float b = 0.6;
float matrixC[N][N] = {};
matrixC[0][0] = a;
matrixC[0][1] = -b;
matrixC[1][0] = b;
matrixC[1][1] = a;

arma::mat ArmaC(N,N,fill::zeros);


for (int i = 0; i < N; ++i)
{
        for(int j = 0; j<N; ++j)
        {
                ArmaC[i+j*N] = matrixC[i][j] ;
        }
}

// To compute and plot the points that move along a circular path
arma::mat BN(N,N,fill::zeros);
BN[0] = ArmaC[0];
BN[1] = ArmaC[1];
BN[2] = ArmaC[2];
BN[3] = ArmaC[3];

ystdbasis = ArmaC*xstdbasis;
arma::mat PSStandardBasis;
```

```
arma::vec Axnewbasis1(n,1);
arma::vec Axnewbasis2(n,1);
Axnewbasis1[0] = ystdbasis[0];
Axnewbasis2[0] = ystdbasis[1];
for (int i = 1; i <= n; i++)
{
       BN*=ArmaC;
       ystdbasis = BN*xstdbasis;
       Axnewbasis1[i] = ystdbasis[0];
       Axnewbasis2[i] = ystdbasis[1];
}

PSStandardBasis.insert_cols(0, Axnewbasis1);
PSStandardBasis.insert_cols(1, Axnewbasis2);

cout << endl;
cout << "Power Sequences for standard basis (as new basis)" << "\n"
    << PSStandardBasis<< endl;
// save matrix as a text file
PSStandardBasis.save("PSStandardBasis.txt", raw_ascii);

cout <<"For l = 0.8 - 0.6 i, Matrix C:" << "\n" << ArmaC <<endl;

arma::mat P(N,N,fill::zeros);
P.col(0) = real(eigvec.col(1)) ; // make Re(x) the first column for
    matrix P
P.col(1) = imag(eigvec.col(1)) ; // make Im(x) the second column for
     matrix P
cout <<"Matrix P:" << "\n" << P <<endl;

arma::mat P_inv(N,N,fill::zeros);
P_inv = inv(P);
cout <<"Matrix P^{-1}:" << "\n" << P_inv <<endl;

cout <<"Matrix P * C * P^{-1}:" << "\n" << P*ArmaC*P_inv <<endl;

// Don't forget to put "\n" at the end of each line!
gp << "set key inside bottom right\n";
gp << "set xlabel 'x'\n";
gp << "set ylabel 'y'\n";
gp << "set title 'Power Sequences of Matrix with Complex Eigenvalues
    '\n";
gp << "set style line 1 lc rgb 'blue' pt 7\n";
gp << "set style line 2 lc rgb 'green' pt 7\n";

// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
```

```
            gp << "plot 'PSAMatrix.txt' using 1:2 title 'A^{k} x' with points ls
                 1, 'PSStandardBasis.txt' using 1:2 title 'R_{t} x_{std}' with
                points ls 2, \n";

            return 0;
    }
```

**C++ Code 184:** *main.cpp "Power sequences of a matrix transformation on a vector"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Before **int main()**, we create a function **double (division)** with two inputs of double type to do division operation

$$\frac{x}{y}$$

```
double division(double x, double y)
{
       return x/y;
}
```

Inside the **int main()**, we use the function **double (division)** to input the entries of matrix $A$ as fraction.

```
arma::mat A(N,N,fill::zeros);
A[0] = division(1,2);
A[1] = division(-3,5);
A[2] = division(3,4);
A[3] = division(11,10);
```

- We create a matrix $B$ with entries the same as matrix $A$. We choose the number of the terms to be computed which is $n = 15$ terms.

We then compute the first term

$$A\boldsymbol{x}_0$$

and save it as vector $\boldsymbol{y}$. Then we declare a matrix name **PSA**

```
arma::mat B(N,N,fill::zeros);
B[0] = A[0];
B[1] = A[1];
B[2] = A[2];
B[3] = A[3];
```

```
int n = 15;

y = A*x;
arma::mat PSA;
```

We create two new vectors with **Armadillo** of size $n \times 1$, the first entries for this vectors are the from vector $\mathbf{y}$ / $Ax_0$

```
arma::vec Ax1(n,1);
arma::vec Ax2(n,1);
Ax1[0] = y[0];
Ax2[0] = y[1];
```

We do a **for** looping to compute the second term till the $n$-th term for $A\boldsymbol{x}_0, A^2\boldsymbol{x}_0, \ldots, A^n\boldsymbol{x}_0$. Here is the procedure
At $i = 1$

$$B = A * A$$
$$y = B * \boldsymbol{x}$$
$$Ax[i = 1] = y[0]$$
$$Ax[i = 1] = y[1]$$

At $i = 2$

$$B = B * A = A * A * A$$
$$y = B * \boldsymbol{x}$$
$$Ax[i = 2] = y[0]$$
$$Ax[i = 2] = y[1]$$

At $i = 3$

$$B = B * A = A * A * A * A$$
$$y = B * \boldsymbol{x}$$
$$Ax[i = 3] = y[0]$$
$$Ax[i = 3] = y[1]$$

Then after the looping finish, we will insert the vector $Ax1$ as the first column for matrix **PSA** and vector $Ax2$ as the second column for matrix **PSA**.

Then we save matrix **PSA** as a textfile.

```
for (int i = 1; i <= n; i++)
{
        float d;

        B*=A;
        y = B*x;
        Ax1[i] = y[0];
        Ax2[i] = y[1];
}

PSA.insert_cols(0, Ax1);
```

```
PSA.insert_cols(1, Ax2);

cout << endl;
cout << "Power Sequences of A" << "\n" << PSA<< endl;
// save matrix as a text file
PSA.save("PSAMatrix.txt", raw_ascii);
```

The last part is computing the factorization

$$A = PCP^{-1}$$

we choose one of the eigenvalue $\lambda = \frac{4}{5} - \frac{3}{5}i$ and the corresponding eigenvector for that $\lambda$ will be used to contruct matrix $P$ and $P^{-1}$.

Finally, with the saved textfile of matrix **PSA** we will be able to plot the first 15 terms of $A^k x_0$ with $k = 1, 2, \ldots, 15$.

```
arma::mat P(N,N,fill::zeros);
P.col(0) = real(eigvec.col(1)) ; // make Re(x) the first column
     for matrix P
P.col(1) = imag(eigvec.col(1)) ; // make Im(x) the second
    column for matrix P
cout <<"Matrix P:" << "\n" << P <<endl;

arma::mat P_inv(N,N,fill::zeros);
P_inv = inv(P);
cout <<"Matrix P^{-1}:" << "\n" << P_inv <<endl;

cout <<"Matrix P * C * P^{-1}:" << "\n" << P*ArmaC*P_inv <<
    endl;

// Don't forget to put "\n" at the end of each line!
gp << "set key inside bottom right\n";
gp << "set xlabel 'x'\n";
gp << "set ylabel 'y'\n";
gp << "set title 'Power Sequences of Matrix with Complex
    Eigenvalues'\n";
gp << "set style line 1 lc rgb 'blue' pt 7\n";
gp << "set style line 2 lc rgb 'blue' pt 5\n";

// '-' means read from stdin. The send1d() function sends data
     to gnuplot's stdin.
gp << "plot 'PSAMatrix.txt' using 1:2 title 'A^{k} x' with
    points ls 1\n";
```

- To plot the circular path, we first define a vector

$$x_{std} = \begin{bmatrix} 1 \\ \frac{1}{2} \end{bmatrix}$$

This vector will be rotated counterclockwise with angle of $\theta = 36.9^0$ with the standard basis $B' = \{(1,0),(0,1)\}$. Thus the transformation should be like this

$$R_\theta \boldsymbol{x}_{std} = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} 1 \\ \frac{1}{2} \end{bmatrix}$$

```
arma::mat xstdbasis(N,1,fill::zeros);
xstdbasis[0] = 1;
xstdbasis[1] = division(1,2);

...
...

arma::mat BN(N,N,fill::zeros);
BN[0] = ArmaC[0];
BN[1] = ArmaC[1];
BN[2] = ArmaC[2];
BN[3] = ArmaC[3];

ystdbasis = ArmaC*xstdbasis;
arma::mat PSStandardBasis;

arma::vec Axnewbasis1(n,1);
arma::vec Axnewbasis2(n,1);
Axnewbasis1[0] = ystdbasis[0];
Axnewbasis2[0] = ystdbasis[1];
for (int i = 1; i <= n; i++)
{
       BN*=ArmaC;
       ystdbasis = BN*xstdbasis;
       Axnewbasis1[i] = ystdbasis[0];
       Axnewbasis2[i] = ystdbasis[1];
}

PSStandardBasis.insert_cols(0, Axnewbasis1);
PSStandardBasis.insert_cols(1, Axnewbasis2);

cout << endl;
cout << "Power Sequences for standard basis (as new basis)" <<
    "\n" << PSStandardBasis<< endl;
// save matrix as a text file
PSStandardBasis.save("PSStandardBasis.txt", raw_ascii);
```

**Figure 23.132:** *The plot of ordered pairs $(x, y)$ from power sequences $A^n \boldsymbol{x}_0 = (PSR_\theta P^{-1})^n \boldsymbol{x}_0$ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Power Sequences for Matrix with Complex Eigenvalues/main.cpp**).*

```
Matrix A :
   0.5000   0.7500
  -0.6000   1.1000

Vector x :
   1.0000
   1.0000

Ax:
   1.2500
   0.5000

Eigenvalues:
λ_1 = 0.8 + 0.6i
λ_2 = 0.8 + -0.6i

Eigenvectors:
    (+7.454e-01,+0.000e+00)    (+7.454e-01,-0.000e+00)
    (+2.981e-01,+5.963e-01)    (+2.981e-01,-5.963e-01)


Power Sequences of A
   1.2500   0.5000
   1.0000  -0.2000
   0.3500  -0.8200
  -0.4400  -1.1120
  -1.0540  -0.9592
  -1.2464  -0.4227
  -0.9402   0.2828
  -0.2580   0.8753
   0.5275   1.1176
   1.1019   0.9129
   1.2356   0.3430
   0.8751  -0.3641
   0.1645  -0.9255
  -0.6119  -1.1168
  -1.1435  -0.8613

For λ = 0.8 - 0.6 i, Matrix C:
   0.8000  -0.6000
   0.6000   0.8000

Matrix P:
   0.7454        0
   0.2981  -0.5963

Matrix P^{-1}:
   1.3416        0
   0.6708  -1.6771

Matrix P * C * P^{-1}:
   0.5000   0.7500
  -0.6000   1.1000
```

**Figure 23.133:** *The computation for first* 15 *terms of* $A^n \boldsymbol{x}_0 = (PSR_\theta P^{-1})^n \boldsymbol{x}_0$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Power Sequences for Matrix with Complex Eigenvalues/main.cpp**).*

**Figure 23.134:** *The plot of ordered pairs $(x, y)$ from power sequences $A^n \boldsymbol{x}_0$ and power sequences $R_\theta \boldsymbol{x}_{std}$, with $\boldsymbol{x}_{std} = (1, 0.5)$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Power Sequences for Matrix with Complex Eigenvalues/main.cpp**).*

## VIII. Solving Systems of Differential Equations with Linear Algebra

[DF*] Differential equations are equations involving functions and their derivatives. Linear algebra, eigenvalues and eigenvectors can be applied to solving systems of differential equations.

[DF*] A differential equation is an equation involving unknown functions and their derivatives. The order of a differential equation is the order of the highest derivative it contains. The simplest differential equations are the first-order equations of the form

$$y' = ay \tag{23.135}$$

where $y = f(x)$ is an unknown differentiable function to be determined, the basic definition of derivative of a function $y = f(x)$ with respect to $x$ is

$$y' = \frac{dy}{dx}$$

where $a$ is a constant. As with most differential equations, this equation has infinitely many solutions; they are the functions of the form

$$y = ce^{ax} \tag{23.136}$$

where $c$ is an arbitrary constant, we call this function $y = ce^{ax}$ as the general solution of $y' = ay$. Below is the step to obtain this general solution

$$\begin{aligned} y' &= ay \\ \frac{dy}{dx} &= ay \\ \frac{dy}{y} &= adx \\ \int \frac{dy}{y} &= \int adx \\ \ln|y| &= ax + c \\ e^{\ln|y|} &= e^{ax+c} \\ y &= ce^{ax} \end{aligned}$$

Thus every function of this form is a solution of $y' = ay$, if we substitute $y$ then

$$y' = cae^{ax} = ay$$

[DF*] Often a physical problem that leads to a differential equation imposes some conditions that enable us to isolate one particular solution from the general solution.

For example, if we have a diferential equation

$$y' = 5y$$

the general solution is

$$y = ce^{5x}$$

now, there is another condition, that is called initial condition that can be represented with this equation

$$y(0) = 6$$

it means $y = 6$ when $x = 0$, then we substitute the initial condition into $y = ce^{5x}$ to obtain the constant $c$

$$\begin{aligned} y &= ce^{5x} \\ y(0) &= ce^{5(0)} \\ 6 &= ce^{0} \\ c &= 6 \end{aligned}$$

so with this initial condition we are able to obtain a single line, or a single solution for certain condition, instead of infinitely many solutions, that is

$$y = 6e^{5x}$$

the equation above is the only solution for the differential equation $y' = 5y$ with initial condition $y(0) = 6$.

The problem of solving a differential equation subject to an initial condition is called an initial-value problem.

[DF*] We will be concerned with solving systems of differential equations of the form

$$\begin{aligned} y_1' &= a_{11}y_1 + a_{12}y_2 + \cdots + a_{1n}y_n \\ y_2' &= a_{21}y_1 + a_{22}y_2 + \cdots + a_{2n}y_n \\ &\vdots \\ y_n' &= a_{n1}y_1 + a_{n2}y_2 + \cdots + a_{nn}y_n \end{aligned} \tag{23.137}$$

where $y_1 = f_1(x),\ y_2 = f_2(x), \ldots, y_n = f_n(x)$ are functions to be determined, and the $a_{ij}$ are constants. In matrix notation, we can write it as

$$\begin{bmatrix} y_1' \\ y_2' \\ \vdots \\ y_n' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

or, more briefly as

$$\boldsymbol{y}' = A\boldsymbol{y} \tag{23.138}$$

where the notation $\boldsymbol{y}'$ denotes the vector obtained by differentiating each component of $y$.

[DF*] The basic idea for solving a system

$$\boldsymbol{y}' = A\boldsymbol{y}$$

whose coefficient matrix is not diagonal is to introduce a new unknown vector $\boldsymbol{u}$ that is related to the unknown vector $\boldsymbol{y}$ by an equation of the form $\boldsymbol{y} = P\boldsymbol{u}$ in which $P$ is an invertible matrix that diagonalizes $A$. Of course, such a matrix may or may not exist, but if it does then we can rewrite the equation $\boldsymbol{y}' = A\boldsymbol{y}$ as

$$P\boldsymbol{u}' = A(P\boldsymbol{u})$$

[DF*] If $A$ a square matrix of size $n \times n$ is diagonalizable and

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

is a solution of the system $\boldsymbol{y}' = A\boldsymbol{y}$, then there will exists a matrix $P$ such that

$$P = [\boldsymbol{p}_1 \ \boldsymbol{p}_2 \ \cdots \ \boldsymbol{p}_n \ ]$$

where $\boldsymbol{p}_i$ is an eigenvector corresponding to $\lambda_i$. Matrix $P$ diagonalizes $A$, and

$$D = P^{-1}AP = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \lambda_n \end{bmatrix}$$

Then we make the substitution

$$\boldsymbol{y} = P\boldsymbol{u}$$
$$\boldsymbol{y}' = P\boldsymbol{u}'$$

that will yield the diagonal system

$$\boldsymbol{u}' = D\boldsymbol{u}$$

or

$$\begin{aligned} u_1' &= \lambda_1 u_1 \\ u_2' &= \lambda_2 u_2 \\ &= \vdots \\ u_n' &= \lambda_n u_n \end{aligned}$$

the solution of this system is

$$\begin{aligned} u_1 &= c_1 e^{\lambda_1 x} \\ u_2 &= c_2 e^{\lambda_2 x} \\ &= \vdots \\ u_n &= c_n e^{\lambda_n x} \end{aligned}$$

the equation $\boldsymbol{y} = P\boldsymbol{u}$ will be

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [\boldsymbol{p}_1 \ \boldsymbol{p}_2 \ \cdots \ \boldsymbol{p}_n \ ] \begin{bmatrix} c_1 e^{\lambda_1 x} \\ c_2 e^{\lambda_2 x} \\ \vdots \\ c_n e^{\lambda_n x} \end{bmatrix} = \boldsymbol{p}_1 c_1 e^{\lambda_1 x} + \boldsymbol{p}_2 c_2 e^{\lambda_2 x} + \cdots + \boldsymbol{p}_n c_n e^{\lambda_n x}$$

so the general solution of the system is a linear combination of $e^{\lambda_1 x}, e^{\lambda_2 x}, \dots, e^{\lambda_n x}$ where $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of $A$.

A little note here, The vector column for matrix $P$ that we sometimes wrote $\boldsymbol{p}_i$ is another notation for $\boldsymbol{x}_i$ the eigenvector of $A$ corresponding to $\lambda_i$.

$$P = [\boldsymbol{p}_1 \ \boldsymbol{p}_2 \ \dots \ \boldsymbol{p}_n \ = [\boldsymbol{x}_1 \ \boldsymbol{x}_2 \ \dots \ \boldsymbol{x}_n$$

**Theorem 23.101: Diagonalizable Matrix $A$ and the General Solution**

If the coefficient matrix $A$ of the system $\boldsymbol{y}' = A\boldsymbol{y}$ is diagonalizable, then the general solution of the system can be expressed as

$$\boldsymbol{y} = c_1 e^{\lambda_1 x}\boldsymbol{x}_1 + c_2 e^{\lambda_2 x}\boldsymbol{x}_2 + \cdots + c_n e^{\lambda_n x}\boldsymbol{x}_n \tag{23.139}$$

where $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $A$, and $\boldsymbol{x}_i$ is an eigenvector of $A$ corresponding to $\lambda_i$.

[DF*] **Example:**
Consider the system of differential equation $\boldsymbol{y}' = A\boldsymbol{y}$, where $A$ is a $2 \times 2$ matrix. For what values of $a_{11}, a_{12}, a_{21}, a_{22}$ do the component solutions $y_1(t), y_2(t)$ tend to zero as $t \to \infty$? In particular, what must be true about the determinant and the trace of $A$ for this to happen?
**Solution:**
From the theorem we know that

$$\boldsymbol{y} = c_1 e^{\lambda_1 t}\boldsymbol{t}_1 + c_2 e^{\lambda_2 t}\boldsymbol{t}_2 \tag{23.140}$$

where $\lambda_1, \lambda_2$ are the eigenvalues of $A$, and $\boldsymbol{t}_i$ is an eigenvector of $A$ corresponding to $\lambda_i$. Now if we take a look at the eigenvector components it will be like this

$$\boldsymbol{t}_1 = \begin{bmatrix} t_{11} \\ t_{12} \end{bmatrix}, \quad \boldsymbol{t}_2 = \begin{bmatrix} t_{21} \\ t_{22} \end{bmatrix}$$

thus

$$\begin{aligned} y_1(t) &= c_1 e^{\lambda_1 t} t_{11} + c_2 e^{\lambda_2 t} t_{21} \\ y_2(t) &= c_1 e^{\lambda_1 t} t_{12} + c_2 e^{\lambda_2 t} t_{22} \end{aligned}$$

when $t \to \infty$, $y_1(t)$ and $y_2(t)$ will tend to zero if and only if

$$\lambda_1 < 0 \quad \text{and} \quad \lambda_2 < 0$$

Now, we will check the condition for the eigenvalues $\lambda_1$ and $\lambda_2$ with the formula

$$\det(\lambda I - A) = 0$$

$$\begin{vmatrix} \lambda - a_{11} & -a_{12} \\ -a_{21} & \lambda - a_{22} \end{vmatrix} = \lambda^2 - (a_{11} + a_{22})\lambda + (a_{11}a_{22} - a_{12}a_{21}) = 0$$

The eigenvalues formula for the characteristic equation above is

$$\lambda_{1,2} = \frac{(a_{11} + a_{22}) \pm \sqrt{(a_{11} + a_{22})^2 - 4(1)(a_{11}a_{22} - a_{12}a_{21})}}{2(1)}$$

we know that

$$\begin{aligned} \operatorname{tr}(A) &= a_{11} + a_{22} \\ \det(A) &= a_{11}a_{22} - a_{12}a_{21} \end{aligned}$$

we can substitute into the eigenvalues formula

$$\lambda_{1,2} = \frac{\text{tr}(A) \pm \sqrt{(\text{tr}(A))^2 - 4(1)(\det(A))}}{2(1)}$$
$$\lambda_{1,2} = \frac{\text{tr}(A) \pm \sqrt{(\text{tr}(A))^2 - 4(\det(A))}}{2}$$

in order to obtain negative eigenvalues then the conditions need to be fulfilled are

$$\begin{aligned}
(\text{tr}(A))^2 - 4(\det(A)) &\geq 0 \\
\text{tr}(A) &< 0 \\
\text{tr}(A) + \sqrt{(\text{tr}(A))^2 - 4(\det(A))} &< 0 \\
\text{tr}(A) - \sqrt{(\text{tr}(A))^2 - 4(\det(A))} &< 0
\end{aligned}$$

if the 4 conditions above are met, then as $t \to \infty$, $y_1(t)$ and $y_2(t)$ will go to zero.

## i. C++ Computation: Solution of a Simple First-Order Linear System of Differential Equations with Initial Conditions

(a) Write the following system in matrix form:

$$\begin{aligned} y_1' &= 3y_1 \\ y_2' &= -2y_2 \\ y_3' &= 5y_3 \end{aligned}$$

(b) Solve the system

(c) Find a solution of the system that satisfies the initial conditions $y_1(0) = 1, y_2(0) = 4$, and $y_3(0) = -2$.

**Solution:**

(a) Write the following system in matrix form:

$$\begin{bmatrix} y_1' \\ y_2' \\ y_3' \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

or

$$\boldsymbol{y}' = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 5 \end{bmatrix} \boldsymbol{y}$$

(b) Because each equation involves only one unknown function, we can solve the equations individually. It follows that these solutions are

$$\begin{aligned} y_1 &= c_1 e^{3x} \\ y_2 &= c_2 e^{-2x} \\ y_3 &= c_3 e^{5x} \end{aligned}$$

or, in matrix notation,

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_1 e^{3x} \\ c_2 e^{-2x} \\ c_3 e^{5x} \end{bmatrix}$$

(c) From the given initial conditions, we obtain

$$\begin{aligned} 1 = y_1(0) &= c_1 e^{3(0)} = c_1 \\ 4 = y_2(0) &= c_2 e^{-2(0)} = c_2 \\ -2 = y_3(0) &= c_3 e^{5(0)} = c_3 \end{aligned}$$

so the solution satisfying these conditions are

$$\begin{aligned} y_1 &= e^{3x} \\ y_2 &= 4e^{-2x} \\ y_3 &= -2e^{5x} \end{aligned}$$

or, in matrix notation,

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} e^{3x} \\ 4e^{-2x} \\ -2e^{5x} \end{bmatrix}$$

For the C++ code, we use the library **GiNaC** that is able to perform symbolic and numerical computation at once. Computing the solution of a linear system of differential equations with initial conditions can be done easily, especially for a very simple example like this one.

```
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 5; // define maximum decimal digits
       symbol x("x") ;
       symbol y1("y1"), y2("y2"), y3("y3") ;
       symbol c1("c1"), c2("c2"), c3("c3") ;

       matrix A = { { 3, 0, 0 }, { 0, -2, 0}, { 0, 0, 5 } };
       matrix y = { { y1}, { y2}, { y3} };
       cout << "\n A = " << A << endl;
       cout << "\n y = " << y << endl;

       matrix result = A.mul(y);

       cout << endl;

       cout << "y' = Ay = " << result <<endl;
       cout << endl;
       cout << "y1' = " << result[0] <<endl;
       cout << "y2' = " << result[1] <<endl;
       cout << "y3' = " << result[2] <<endl;

       ex a1 = result[0] / y1 ;
       ex a2 = result[1] / y2 ;
       ex a3 = result[2] / y3 ;
       ex fx1 = c1*exp(a1*x) ;
       ex fx2 = c2*exp(a2*x) ;
       ex fx3 = c3*exp(a3*x) ;
       cout << endl;

       cout << "y1 = " << fx1 <<endl;
       cout << "y2 = " << fx2 <<endl;
       cout << "y3 = " << fx3 <<endl;
       cout << endl;
```

```
        cout << "Solving the differential equations with initial conditions
            " << endl;
        cout << "y1(0) = 1, y2(0) = 4, y3(0) = -2 " << endl;
        cout << endl;

        // Define the initial conditions y1(0) = ic1, y2(0) = ic2, y3(0) =
            ic3
        ex ic1 = 1 ;
        ex ic2 = 4 ;
        ex ic3 = -2;

        ex c1_ans = lsolve(fx1.subs(x==0) == 1, c1) ;
        ex c2_ans = lsolve(fx2.subs(x==0) == 4, c2) ;
        ex c3_ans = lsolve(fx3.subs(x==0) == -2, c3) ;

        cout << "c1 = " << c1_ans<<endl;
        cout << "c2 = " << c2_ans <<endl;
        cout << "c3 = " << c3_ans <<endl;
        cout << endl;

        ex fx1_sol = c1_ans*exp(a1*x) ;
        ex fx2_sol = c2_ans*exp(a2*x) ;
        ex fx3_sol = c3_ans*exp(a3*x) ;

        cout << "The general solution satisfying the initial conditions are"
             << endl;
        cout << endl;

        cout << "y1 = " << fx1_sol <<endl;
        cout << "y2 = " << fx2_sol <<endl;
        cout << "y3 = " << fx3_sol <<endl;
        cout << endl;

        matrix y_sol = { { fx1_sol}, { fx2_sol}, { fx3_sol} };
        cout << "The general solution in matrix notation:" << endl;
        cout << "\n y = " << y_sol << endl;

        return 0;
    }
```

**C++ Code 185:** *main.cpp "Solution of a simple first order linear system of differential equations with initial conditions"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**

**./main**

```
 A = [[3,0,0],[0,-2,0],[0,0,5]]

 y = [[y1],[y2],[y3]]

y' = Ay = [[3*y1],[-2*y2],[5*y3]]

y1' = 3*y1
y2' = -2*y2
y3' = 5*y3

y1 = exp(3*x)*c1
y2 = c2*exp(-2*x)
y3 = c3*exp(5*x)

Solving the differential equations with initial conditions
y1(0) = 1, y2(0) = 4, y3(0) = -2

c1 = 1
c2 = 4
c3 = -2

The general solution satisfying the initial conditions are

y1 = exp(3*x)
y2 = 4*exp(-2*x)
y3 = -2*exp(5*x)

The general solution in matrix notation:

 y = [[exp(3*x)],[4*exp(-2*x)],[-2*exp(5*x)]]
```

**Figure 23.135:** *The computation to find the solution of a first-order linear system of differential equation with initial conditions (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of a First Order Linear System of Differential Equations with Initial Conditions/main.cpp**).*

## ii. C++ Computation: Solution of a First-Order Linear System of Differential Equations with Initial Conditions

If the previous section has a very simple form of a system of differential equations, this section involves a bit complex form of a first-order linear system of differential equations.

(a) Solve the system

$$\begin{aligned} y_1' &= y_1 + y_2 \\ y_2' &= 4y_1 - 2y_2 \end{aligned}$$

(b) Find the solution that satisfies the initial conditions $y_1(0) = 1, y_2(0) = 6$.

**Solution:**

(a) The coefficient matrix for the system is

$$\begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

Matrix $A$ will be diagonalized by any matrix $P$ whose columns are linearly independent eigenvectors of $A$. Since

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - 1 & -1 \\ -4 & \lambda + 2 \end{vmatrix} = \lambda^2 + \lambda - 6 = (\lambda + 3)(\lambda - 2)$$

the eigenvalues of $A$ are

$$\begin{aligned} \lambda_1 &= 2 \\ \lambda_2 &= -3 \end{aligned}$$

By definition,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of

$$\begin{bmatrix} \lambda - 1 & -1 \\ -4 & \lambda + 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

If $\lambda = 2$, this system becomes

$$\begin{bmatrix} 1 & -1 \\ -4 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solving this system yields

$$\begin{aligned} x_1 - x_2 &= 0 \\ x_1 &= x_2 \\ x_2 &= t \end{aligned}$$

so

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix} = t \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Thus,

$$\boldsymbol{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 2$.

Similarly, you can show that

$$\boldsymbol{p}_2 = \begin{bmatrix} -\frac{1}{4} \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = -3$. Thus,

$$P = [\boldsymbol{p}_1 \ \boldsymbol{p}_2] = \begin{bmatrix} 1 & -\frac{1}{4} \\ 1 & 1 \end{bmatrix}$$

diagonalizes $A$, and

$$D = P^{-1}AP = \begin{bmatrix} 2 & 0 \\ 0 & -3 \end{bmatrix}$$

The substitution

$$\begin{aligned} \boldsymbol{y} &= P\boldsymbol{u} \\ \boldsymbol{y}' &= P\boldsymbol{u}' \end{aligned}$$

yields the "diagonal system"

$$\boldsymbol{u}' = D\boldsymbol{u} = \begin{bmatrix} 2 & 0 \\ 0 & -3 \end{bmatrix} \boldsymbol{u}$$

or

$$\begin{aligned} u_1' &= 2u_1 \\ u_2' &= -3u_2 \end{aligned}$$

The solution of this system is

$$\begin{aligned} u_1 &= c_1 e^{2x} \\ u_2 &= c_2 e^{-3x} \end{aligned}$$

or

$$\boldsymbol{u} = \begin{bmatrix} c_1 e^{2x} \\ c_2 e^{-3x} \end{bmatrix}$$

now we go back to the equation $\boldsymbol{y} = P\boldsymbol{u}$ and substitute $\boldsymbol{u}$ that we just compute above

$$\boldsymbol{y} = P\boldsymbol{u} = \begin{bmatrix} 1 & -\frac{1}{4} \\ 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 e^{2x} \\ c_2 e^{-3x} \end{bmatrix} = \begin{bmatrix} c_1 e^{2x} - \frac{1}{4} c_2 e^{-3x} \\ c_1 e^{2x} + c_2 e^{-3x} \end{bmatrix}$$

or

$$\begin{aligned} y_1 &= c_1 e^{2x} - \frac{1}{4} c_2 e^{-3x} \\ y_2 &= c_1 e^{2x} + c_2 e^{-3x} \end{aligned}$$

(b) If we substitute the initial conditions $y_1(0) = 1, y_2(0) = 6$ into

$$y_1 = c_1 e^{2x} - \frac{1}{4} c_2 e^{-3x}$$
$$y_2 = c_1 e^{2x} + c_2 e^{-3x}$$

we will obtain

$$c_1 - \frac{1}{4} c_2 = 1$$
$$c_1 + c_2 = 6$$

Solving this system, we obtain

$$c_1 = 2$$
$$c_2 = 4$$

so the solution satisfying the initial condition is

$$y_1 = 2e^{2x} - \frac{1}{4} 4e^{-3x}$$
$$y_2 = 2e^{2x} + 4e^{-3x}$$

In this section, the C++ code will be using two libraries, first is **Armadillo** for the efficiency in computing eigenvalues and eigenvectors in numerical terms, then we use **GiNaC** to do symbolic matrix computation till we obtain the general solution for the general first order linear system of differential equations.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <ginac/ginac.h>

using namespace std;
using namespace arma;
using namespace GiNaC;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix
```

```
cout << "Eigenvalues:" << endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;

cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
cout << endl;

cout << "Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

mat P(2,2,fill::eye);
mat PI(2,2,fill::eye);
P = real(eigvec);
PI = inv(P);

mat D(2,2,fill::eye);
D = PI*A*P;
for (int i = 0; i < n; i++)
{
       for (int j = 0; j < n; j++)
       {
              if (i !=j)
              {
                     D(i,j) = 0;
              }
       }
}

cout <<"Matrix P:" << "\n" << P <<endl;
cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;

// GiNaC computation starts here for symbolic computation
Digits = 5; // define maximum decimal digits
```

```
symbol x("x") ;
symbol u1("u1"), y2("u2");
symbol c1("c1"), c2("c2") ;

ex fx1 = c1*exp(D(0,0)*x) ;
ex fx2 = c2*exp(D(1,1)*x) ;
cout << endl;

cout << "u1 = " << fx1 <<endl;
cout << "u2 = " << fx2 <<endl;
cout << endl;

matrix P_ginac = { { P(0,0), P(0,1) }, { P(1,0), P(1,1)} };
matrix u = { { fx1}, { fx2} };

cout << "\n P = " << P_ginac << endl;

matrix Pu = P_ginac.mul(u);

cout << "\n y1 = " << Pu(0,0) << endl;
cout << "\n y2 = " << Pu(1,0) << endl;

ex fx1_final = Pu(0,0) ;
ex fx2_final = Pu(1,0) ;

// Define the initial conditions y1(0) = ic1, y2(0) = ic2, y3(0) =
    ic3
ex ic1 = 1 ;
ex ic2 = 6 ;

// List the equations and the variables needed to be solved
lst equations = {fx1_final.subs(x==0) == ic1, fx2_final.subs(x==0)
    == ic2} ;
lst variables = {c1, c2};

cout << endl;
cout << "Solving the differential equations with initial conditions
    " << endl;
cout << "y1(0) = 1, y2(0) = 6 " << endl;
cout << endl;

//cout << "Solutions for c1 and c2 = " << lsolve(equations,
    variables) <<endl;

ex c1_ans = lsolve(equations, variables)[0].rhs() ;
ex c2_ans = lsolve(equations, variables)[1].rhs() ;

cout << "c1 = " << c1_ans <<endl;
```

```
        cout << "c2 = " << c2_ans <<endl;

        cout << endl;
        cout << "The general solution satisfying the initial conditions are"
             << endl;
        cout << endl;

        ex y1_sol = fx1_final.subs(lst{c1==c1_ans, c2 ==c2_ans});
        ex y2_sol = fx2_final.subs(lst{c1==c1_ans, c2 ==c2_ans});

        cout << "y1 = " << y1_sol <<endl;
        cout << "y2 = " << y2_sol <<endl;

        return 0;
    }
```

**C++ Code 186:** *main.cpp "Solution of a general first order linear system of differential equations with initial conditions"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lgln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The first half of the code is using **Armadillo** to compute the eigenvalues, eigenvectors, matrix $P$, and the diagonal matrix $D$.

```
        mat A;
        A.load("matrixA.txt");
        cx_mat eigvec;
        cx_vec eigval;

        mat I(2,2,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense
            general square matrix

        cout << "Eigenvalues:" << endl;

        ...
        ...

        mat P(2,2,fill::eye);
        mat PI(2,2,fill::eye);
        P = real(eigvec);
```

```
PI = inv(P);

mat D(2,2,fill::eye);
D = PI*A*P;
for (int i = 0; i < n; i++)
{
        for (int j = 0; j < n; j++)
        {
                if (i !=j)
                {
                        D(i,j) = 0;
                }
        }
}

cout <<"Matrix P:" << "\n" << P <<endl;
cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;
```

- The second half of the code is using **GiNac** to do symbolic computation to find the solution in symbolic term.

  What we need to compute with **GiNac** for this problem:

  1. Vector $\boldsymbol{u}'$, as the result of multiplication of $D\boldsymbol{u}$. $u_i'$ will only contain the term $u_i$ since $D$ is a diagonal matrix, so we can easily determine the solution for the differential equation of
  $$u_i' = au_i$$
  $$u_i = ce^{ax}$$
  with $c_i$ as arbitrary constant, $a$ is a constant, and $i = 1, 2$.
  2. Vector $\boldsymbol{u}$ as the solution for the system $\boldsymbol{u}' = D\boldsymbol{u}$.
  3. $c_1$ and $c_2$ with initial conditions given.
  4. $y_1$ and $y_2$ after we know the values of $c_1$ and $c_2$.

```
Digits = 5; // define maximum decimal digits
symbol x("x") ;
symbol u1("u1"), y2("u2");
symbol c1("c1"), c2("c2") ;

ex fx1 = c1*exp(D(0,0)*x) ;
ex fx2 = c2*exp(D(1,1)*x) ;
cout << endl;

...
...

// Define the initial conditions y1(0) = ic1, y2(0) = ic2, y3
    (0) = ic3
ex ic1 = 1 ;
```

```
ex ic2 = 6 ;

// List the equations and the variables needed to be solved
lst equations = {fx1_final.subs(x==0) == ic1, fx2_final.subs(x
    ==0) == ic2} ;
lst variables = {c1, c2};

cout << endl;
cout << "Solving the differential equations with initial
    conditions " << endl;
cout << "y1(0) = 1, y2(0) = 6 " << endl;
cout << endl;

//cout << "Solutions for c1 and c2 = " << lsolve(equations,
    variables) <<endl;

ex c1_ans = lsolve(equations, variables)[0].rhs() ;
ex c2_ans = lsolve(equations, variables)[1].rhs() ;

cout << "c1 = " << c1_ans <<endl;
cout << "c2 = " << c2_ans <<endl;

cout << endl;
cout << "The general solution satisfying the initial conditions
     are" << endl;
cout << endl;

ex y1_sol = fx1_final.subs(lst{c1==c1_ans, c2 ==c2_ans});
ex y2_sol = fx2_final.subs(lst{c1==c1_ans, c2 ==c2_ans});

cout << "y1 = " << y1_sol <<endl;
cout << "y2 = " << y2_sol <<endl;
```

```
Matrix A:
   1.0000   1.0000
   4.0000  -2.0000

Eigenvalues:
λ_1 = 2 + 0i
λ_2 = -3 + 0i

Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Eigenvectors:
x[λ_1] :
    0.7071
   0.7071

x[λ_2] :
   -0.2425
   0.9701

Matrix P:
   0.7071  -0.2425
   0.7071   0.9701

Matrix P^{-1}AP:
   2.0000        0
        0  -3.0000


u1 = exp((2.0)*x)*c1
u2 = c2*exp(-(3.0)*x)


 P = [[0.70710677,-0.24253562],[0.70710677,0.9701425]]

 y1 = (0.70710677)*exp((2.0)*x)*c1-(0.24253562)*c2*exp(-(3.0)*x)

 y2 = (0.70710677)*exp((2.0)*x)*c1+(0.9701425)*c2*exp(-(3.0)*x)

Solving the differential equations with initial conditions
y1(0) = 1, y2(0) = 6

c1 = 2.828427
c2 = 4.1231055

The general solution satisfying the initial conditions are

y1 = -(0.99999994)*exp(-(3.0)*x)+(1.9999999)*exp((2.0)*x)
y2 = (3.9999998)*exp(-(3.0)*x)+(1.9999999)*exp((2.0)*x)
```

**Figure 23.136:** *The computation to find the solution of a general first-order linear system of differential equation with initial conditions (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of a General First Order Linear System of Differential Equations with Initial Conditions/main.cpp**).*

## iii. C++ Computation: Solution of a Second-Order Linear Differential Equation

Suppose we have a second order differential equation

$$y'' - 7y' + 6y = 0$$

it is possible to solve a single higher-order linear differential equation with constant coefficients by expressing it as a system of first-order differential equations.

First, we substitute

$$\begin{aligned} y_1 &= y \\ y_2 &= y' \end{aligned}$$

So, the equation $y'' - 7y' + 6y = 0$ can be written as

$$y_2' - 7y_2 + 6y_1 = 0$$

we can write the system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -6y_1 + 7y_2 \end{aligned}$$

The coefficient matrix for the system is

$$\begin{bmatrix} 0 & 1 \\ -6 & 7 \end{bmatrix}$$

Matrix $A$ will be diagonalized by any matrix $P$ whose columns are linearly independent eigenvectors of $A$. Since

$$\det(\lambda I - A) = \begin{vmatrix} \lambda & -1 \\ 6 & \lambda - 7 \end{vmatrix} = \lambda^2 - 7\lambda + 6 = (\lambda - 1)(\lambda - 6)$$

the eigenvalues of $A$ are

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 6 \end{aligned}$$

By definition,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of

$$\begin{bmatrix} \lambda & -1 \\ 6 & \lambda - 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

If $\lambda = 1$, this system becomes

$$\begin{bmatrix} 1 & -1 \\ 6 & -6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Solving this system yields

$$\begin{aligned} x_1 - x_2 &= 0 \\ x_1 &= x_2 \\ x_2 &= t \end{aligned}$$

so

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix} = t \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Thus,

$$\boldsymbol{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 1$.

Similarly, you can show that

$$\boldsymbol{p}_2 = \begin{bmatrix} \frac{1}{6} \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 6$. Thus,

$$P = [\boldsymbol{p}_1 \ \boldsymbol{p}_2] = \begin{bmatrix} 1 & \frac{1}{6} \\ 1 & 1 \end{bmatrix}$$

diagonalizes $A$, and

$$D = P^{-1}AP = \begin{bmatrix} 1 & 0 \\ 0 & 6 \end{bmatrix}$$

The substitution

$$\begin{aligned} \boldsymbol{y} &= P\boldsymbol{u} \\ \boldsymbol{y}' &= P\boldsymbol{u}' \end{aligned}$$

yields the "diagonal system"

$$\boldsymbol{u}' = D\boldsymbol{u} = \begin{bmatrix} 1 & 0 \\ 0 & 6 \end{bmatrix} \boldsymbol{u}$$

or

$$\begin{aligned} u_1' &= u_1 \\ u_2' &= 6u_2 \end{aligned}$$

The solution of this system is

$$\begin{aligned} u_1 &= c_1 e^x \\ u_2 &= c_2 e^{6x} \end{aligned}$$

or

$$\boldsymbol{u} = \begin{bmatrix} c_1 e^x \\ c_2 e^{6x} \end{bmatrix}$$

now we go back to the equation $\boldsymbol{y} = P\boldsymbol{u}$ and substitute $\boldsymbol{u}$ that we just compute above

$$\boldsymbol{y} = P\boldsymbol{u} = \begin{bmatrix} 1 & \frac{1}{6} \\ 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 e^x \\ c_2 e^{6x} \end{bmatrix} = \begin{bmatrix} c_1 e^x + \frac{1}{6} c_2 e^{6x} \\ c_1 e^x + c_2 e^{6x} \end{bmatrix}$$

or

$$y_1 = c_1 e^x + \frac{1}{6} c_2 e^{6x}$$
$$y_2 = c_1 e^x + c_2 e^{6x}$$

Now if we substitute this to the original second-order differential equation $y'' - 7y' + 6y = 0$ that can be written as

$$y_2' - 7y_2 + 6y_1 = 0$$

now we substitute the general solution above, thus we will obtain

$$y_1' = y_2 = c_1 e^x + c_2 e^{6x}$$
$$y_2' = -6y_1 + 7y_2 = -6\left(c_1 e^x + \frac{1}{6} c_2 e^{6x}\right) + 7(c_1 e^x + c_2 e^{6x})$$

so

$$y_2' - 7y_2 + 6y_1 = 0$$
$$-6\left(c_1 e^x + \frac{1}{6} c_2 e^{6x}\right) + 7(c_1 e^x + c_2 e^{6x}) - 7(c_1 e^x + c_2 e^{6x}) + 6\left(c_1 e^x + \frac{1}{6} c_2 e^{6x}\right) = 0$$

If there is initial conditions we can obtain the constants for $c_1$ and $c_2$ to get the single solution. But without initial conditions, we just stop till this step.

The C++ code is pretty much the same as the previous section we use **Armadillo** to do the numerical computation for matrices and then we use **GiNaC** to do the symbolic computation.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <ginac/ginac.h>

using namespace std;
using namespace arma;
using namespace GiNaC;

// The second order differential equaiton is: y'' — 7y' + 6y = 0
// Substitutions: y1 = y, y2 = y'
// The system becomes
// y1' = y2
// y2' = —6y1 + 7 y2
//
// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
```

```
cx_vec eigval;

mat I(2,2,fill::eye);
cout <<"Matrix A:" << "\n" << A <<endl;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
    square matrix

cout << "Eigenvalues:" << endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;

cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
cout << endl;

cout << "Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

mat P(2,2,fill::eye);
mat PI(2,2,fill::eye);
P = real(eigvec);
PI = inv(P);

mat D(2,2,fill::eye);
D = PI*A*P;
for (int i = 0; i < n; i++)
{
       for (int j = 0; j < n; j++)
       {
              if (i !=j)
              {
                     D(i,j) = 0;
              }
       }
```

```
        }

        cout <<"Matrix P:" << "\n" << P <<endl;
        cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;

        // GiNaC computation starts here for symbolic computation
        Digits = 5; // define maximum decimal digits
        symbol x("x") ;
        symbol u1("u1"), y2("u2");
        symbol c1("c1"), c2("c2") ;

        ex fx1 = c1*exp(D(0,0)*x) ;
        ex fx2 = c2*exp(D(1,1)*x) ;
        cout << endl;

        cout << "u1 = " << fx1 <<endl;
        cout << "u2 = " << fx2 <<endl;
        cout << endl;

        matrix P_ginac = { { P(0,0), P(0,1) }, { P(1,0), P(1,1)} };
        matrix u = { { fx1}, { fx2} };

        cout << "\n P = " << P_ginac << endl;

        matrix Pu = P_ginac.mul(u);

        cout << "\n y1 = " << Pu(0,0) << endl;
        cout << "\n y2 = " << Pu(1,0) << endl;

        ex fx1_final = Pu(0,0) ;
        ex fx2_final = Pu(1,0) ;

        cout << endl;
        cout << "The second order differential equation is :" << endl;
        cout << "y'' - 7' + 6y = 0 " <<endl;
        cout << "y2' - 7y2 + 6y1 = 0 " <<endl;

        cout << "The general solution for the second order differential
            equation is :" << endl;
        cout << (-6*fx1_final + 7*fx2_final ) << -7*fx2_final << + 6*
            fx1_final<< "=" ;
        cout << (-6*fx1_final + 7*fx2_final ) -7*fx2_final + 6*fx1_final
            <<endl;

        return 0;
    }
```

**C++ Code 187:** *main.cpp "Solution of a second order linear differential equation"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lgln**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A:
        0   1.0000
  -6.0000   7.0000

Eigenvalues:
λ_1 = 1 + 0i
λ_2 = 6 + 0i

Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Eigenvectors:
x[λ_1] :
   -0.7071
  -0.7071

x[λ_2] :
   -0.1644
  -0.9864

Matrix P:
  -0.7071  -0.1644
  -0.7071  -0.9864

Matrix P^{-1}AP:
   1.0000        0
        0   6.0000


u1 = exp(x)*c1
u2 = exp((6.0)*x)*c2


 P = [[-0.70710677,-0.16439898],[-0.70710677,-0.9863939]]

 y1 = -(0.70710677)*exp(x)*c1-(0.16439898)*exp((6.0)*x)*c2

 y2 = -(0.70710677)*exp(x)*c1-(0.9863939)*exp((6.0)*x)*c2

The second order differential equation is :
y'' - 7' + 6y = 0
y2' - 7y2 + 6y1 = 0
The general solution for the second order differential equation is :
-(0.70710707)*exp(x)*c1-(5.9183636)*exp((6.0)*x)*c2(4.9497476)*exp(x)*c1+(6.9047
575)*exp((6.0)*x)*c2-(4.2426405)*exp(x)*c1-(0.9863939)*exp((6.0)*x)*c2=0
```

**Figure 23.137:** *The computation to find the solution of a second-order differential equation with substitution to make it become system of first-order differential equations (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of a Second Order Linear Differential Equation with Substitution to Become System of First Order/main.cpp**).*

## iv. C++ Computation: Solution of a Third-Order Linear Differential Equation

Solve

$$y''' - 6y'' + 11y' - 6y = 0$$

**Solution:**

First, we substitute

$$\begin{aligned} y_1 &= y \\ y_2 &= y' \\ y_3 &= y'' \end{aligned}$$

So, the equation $y''' - 6y'' + 11y' - 6y = 0$ can be written as

$$y_3' - 6y_3 + 11y_2 - 6y_1 = 0$$

we can write the system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= 6y_1 - 11y_2 + 6y_3 \end{aligned}$$

The coefficient matrix for the system is

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 6 & -11 & 6 \end{bmatrix}$$

Matrix $A$ will be diagonalized by any matrix $P$ whose columns are linearly independent eigenvectors of $A$. Since

$$\det(\lambda I - A) = \begin{vmatrix} \lambda & -1 & 0 \\ 0 & \lambda & -1 \\ -6 & 11 & \lambda - 6 \end{vmatrix} = (\lambda - 1)(\lambda - 2)(\lambda - 3)$$

the eigenvalues of $A$ are

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 2 \\ \lambda_3 &= 3 \end{aligned}$$

By definition,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

is an eigenvector of $A$ corresponding to $\lambda$ if and only if $\boldsymbol{x}$ is a nontrivial solution of

$$\begin{bmatrix} \lambda & -1 & 0 \\ 0 & \lambda & -1 \\ -6 & 11 & \lambda - 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If $\lambda = 1$, this system becomes

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -6 & 11 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this system yields

$$\begin{aligned} x_1 - x_2 &= 0 \\ x_2 - x_3 &= 0 \\ -6x_1 + 11x_2 - 5x_3 &= 0 \\ x_3 &= t \end{aligned}$$

so

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} t \\ t \\ t \end{bmatrix} = t \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Thus,

$$\boldsymbol{p}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 1$.

Similarly, you can show that

$$\boldsymbol{p}_2 = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{2} \\ 1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 2$, and

$$\boldsymbol{p}_3 = \begin{bmatrix} -\frac{1}{9} \\ -\frac{1}{3} \\ -1 \end{bmatrix}$$

is a basis for the eigenspace corresponding to $\lambda = 3$. Thus,

$$P = [\boldsymbol{p}_1 \ \boldsymbol{p}_2] = \begin{bmatrix} 1 & \frac{1}{4} & -\frac{1}{9} \\ 1 & \frac{1}{2} & -\frac{1}{3} \\ 1 & 1 & -1 \end{bmatrix}$$

diagonalizes $A$, and

$$D = P^{-1}AP = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

The substitution

$$\begin{aligned} \boldsymbol{y} &= P\boldsymbol{u} \\ \boldsymbol{y}' &= P\boldsymbol{u}' \end{aligned}$$

yields the "diagonal system"

$$\boldsymbol{u}' = D\boldsymbol{u} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \boldsymbol{u}$$

or

$$\begin{aligned} u_1' &= u_1 \\ u_2' &= 2u_2 \\ u_3' &= 3u_3 \end{aligned}$$

The solution of this system is

$$\begin{aligned} u_1 &= c_1 e^x \\ u_2 &= c_2 e^{2x} \\ u_3 &= c_3 e^{3x} \end{aligned}$$

or

$$\boldsymbol{u} = \begin{bmatrix} c_1 e^x \\ c_2 e^{2x} \\ c_3 e^{3x} \end{bmatrix}$$

now we go back to the equation $\boldsymbol{y} = P\boldsymbol{u}$ and substitute $\boldsymbol{u}$ that we just compute above

$$\boldsymbol{y} = P\boldsymbol{u} = \begin{bmatrix} 1 & \frac{1}{4} & -\frac{1}{9} \\ 1 & \frac{1}{2} & -\frac{1}{3} \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} c_1 e^x \\ c_2 e^{2x} \\ c_3 e^{3x} \end{bmatrix} = \begin{bmatrix} c_1 e^x + \frac{1}{4} c_2 e^{2x} - \frac{1}{9} c_3 e^{3x} \\ c_1 e^x + \frac{1}{2} c_2 e^{2x} - \frac{1}{3} c_3 e^{3x} \\ c_1 e^x + c_2 e^{2x} - c_3 e^{3x} \end{bmatrix}$$

or

$$\begin{aligned} y_1 &= c_1 e^x + \frac{1}{4} c_2 e^{2x} - \frac{1}{9} c_3 e^{3x} \\ y_2 &= c_1 e^x + \frac{1}{2} c_2 e^{2x} - \frac{1}{3} c_3 e^{3x} \\ y_3 &= c_1 e^x + c_2 e^{2x} - c_3 e^{3x} \end{aligned}$$

Now if we substitute this to the original second-order differential equation $y''' - 6y'' + 11y' - 6y = 0$ that can be written as

$$y_3' - 6y_3 + 11y_2 - 6y_1 = 0$$

now we substitute the general solution above, thus we will obtain

$$\begin{aligned} y_1' &= y_2 = c_1 e^x + \frac{1}{2} c_2 e^{2x} - \frac{1}{3} c_3 e^{3x} \\ y_2' &= y_3 = c_1 e^x + c_2 e^{2x} - c_3 e^{3x} \\ y_3' &= 6y_1 - 11y_2 + 6y_3 = 6\left(c_1 e^x + \frac{1}{4} c_2 e^{2x} - \frac{1}{9} c_3 e^{3x}\right) - 11\left(c_1 e^x + \frac{1}{2} c_2 e^{2x} - \frac{1}{3} c_3 e^{3x}\right) - 6(c_1 e^x + c_2 e^{2x} - c_3 e^{3x}) \end{aligned}$$

so

$$y_3' - 6y_3 + 11y_2 - 6y_1 = 0$$

you can substitute all the terms of $y_3' - 6y_3 + 11y_2 - 6y_1$ into the terms with $c_1, c_2, c_3$ and the equation will hold, meaning that we have found the general solution. If there is initial conditions we can obtain the constants for $c_1$ and $c_2$ to get the single solution. But without initial conditions, we just stop till this step.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <ginac/ginac.h>

using namespace std;
using namespace arma;
using namespace GiNaC;

// The third order differential equation is: y''' — 6y'' + 11y' — 6y = 0
// Substitutions: y1 = y, y2 = y', y3 = y''
// The system becomes
// y1' = y2
// y2' = y3
// y3' = 6y1 — 11y2 + 6y3
// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(3,3,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix

       cout << "Eigenvalues:" << endl;

       int n = 3;
       for (int i = 1; i <= n; i++)
       {
              cout << "l_" << i << " = " <<real(eigval[i—1]) << " + " <<
                  imag(eigval[i—1]) << "i" << endl;
       }

       mat A1 = I*real(eigval[0]) — A;
       mat A2 = I*real(eigval[1]) — A;
       mat A3 = I*real(eigval[2]) — A;
```

```
cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
cout << "Eigenspaces for l_3: " << n - arma::rank(A3) << endl;
cout << endl;

cout << "Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

mat P(3,3,fill::eye);
mat PI(3,3,fill::eye);
P = real(eigvec);
PI = inv(P);

mat D(3,3,fill::eye);
D = PI*A*P;
for (int i = 0; i < n; i++)
{
       for (int j = 0; j < n; j++)
       {
              if (i !=j)
              {
                     D(i,j) = 0;
              }
       }
}

cout <<"Matrix P:" << "\n" << P <<endl;
cout <<"Matrix P^{-1}AP:" << "\n" << D <<endl;

// GiNaC computation starts here for symbolic computation
Digits = 5; // define maximum decimal digits
symbol x("x") ;
symbol u1("u1"), u2("u2"), u3("u3");
symbol c1("c1"), c2("c2"), c3("c3") ;

ex fx1 = c1*exp(D(0,0)*x) ;
ex fx2 = c2*exp(D(1,1)*x) ;
ex fx3 = c3*exp(D(2,2)*x) ;
cout << endl;

cout << "u1 = " << fx1 <<endl;
cout << "u2 = " << fx2 <<endl;
cout << "u3 = " << fx3 <<endl;
```

```
        cout << endl;

        matrix P_ginac = { { P(0,0), P(0,1), P(0,2) }, { P(1,0), P(1,1), P
            (1,2)}, { P(2,0), P(2,1), P(2,2)} };
        matrix u = { { fx1}, { fx2}, {fx3} };

        cout << "\n P = " << P_ginac << endl;

        matrix Pu = P_ginac.mul(u);

        cout << "\n y1 = " << Pu(0,0) << endl;
        cout << "\n y2 = " << Pu(1,0) << endl;
        cout << "\n y2 = " << Pu(2,0) << endl;

        ex fx1_final = Pu(0,0) ;
        ex fx2_final = Pu(1,0) ;
        ex fx3_final = Pu(2,0) ;

        cout << endl;
        cout << "The third order differential equation is :" << endl;
        cout << "y''' - 6y'' + 11y' - 6y = 0" <<endl;
        cout << "y3' - 6y3 + 11y2 - 6y1 = 0 " <<endl;

        cout << "The general solution for the second order differential
            equation is :" << endl;
        cout << (6*fx1_final - 11*fx2_final + 6*fx3_final ) << -6*
            fx3_final << + 11*fx2_final << -6*fx1_final << "=" ;
        cout << (6*fx1_final - 11*fx2_final + 6*fx3_final ) -6*fx3_final +
             11*fx2_final -6*fx1_final <<endl;

        return 0;
    }
```

**C++ Code 188:** *main.cpp "Solution of a third order linear differential equation"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A:
         0    1.0000         0
         0         0    1.0000
    6.0000  -11.0000    6.0000

Eigenvalues:
λ_1 = 1 + 0i
λ_2 = 2 + 0i
λ_3 = 3 + 0i

Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Eigenvectors:
x[λ_1] :
    0.5774
   0.5774
   0.5774

x[λ_2] :
    0.2182
   0.4364
   0.8729

x[λ_3] :
   -0.1048
  -0.3145
  -0.9435

Matrix P:
   0.5774   0.2182  -0.1048
   0.5774   0.4364  -0.3145
   0.5774   0.8729  -0.9435

Matrix P^{-1}AP:
   1.0000        0        0
        0   2.0000        0
        0        0   3.0000
```

```
xterm

u1 = c1*exp(x)
u2 = c2*exp((2.0)*x)
u3 = exp((3.0)*x)*c3


 P = [[0.57735026,0.2182179,-0.104828484],[0.57735026,0.4364358,-0.31448546],[0.
57735026,0.8728716,-0.94345635]]

 y1 = (0.57735026)*c1*exp(x)+(0.2182179)*c2*exp((2.0)*x)-(0.104828484)*exp((3.0)
*x)*c3

 y2 = (0.57735026)*c1*exp(x)+(0.4364358)*c2*exp((2.0)*x)-(0.31448546)*exp((3.0)*
x)*c3

 y2 = (0.57735026)*c1*exp(x)+(0.8728716)*c2*exp((2.0)*x)-(0.94345635)*exp((3.0)*
x)*c3

The third order differential equation is :
y''' - 6y'' + 11y' - 6y = 0
y3' - 6y3 + 11y2 - 6y1 = 0
The general solution for the second order differential equation is :
(0.57735014)*c1*exp(x)+(1.745743)*c2*exp((2.0)*x)-(2.8303688)*exp((3.0)*x)*c3-(3
.4641016)*c1*exp(x)-(5.2372293)*c2*exp((2.0)*x)+(5.660738)*exp((3.0)*x)*c3(6.350
853)*c1*exp(x)+(4.8007936)*c2*exp((2.0)*x)-(3.45934)*exp((3.0)*x)*c3-(3.4641016)
*c1*exp(x)-(1.3093073)*c2*exp((2.0)*x)+(0.6289709)*exp((3.0)*x)*c3=(5.9604645E-8
)*exp((3.0)*x)*c3
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Compute Solution of a Third Order Linear
 Differential Equation with Substitution to Become System of First Order ]#
```

**Figure 23.138:** *The computation to find the solution of a third-order differential equation with substitution to make it become system of first-order differential equations (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of a Third Order Linear Differential Equation with Substitution to Become System of First Order/main.cpp**).*

# IX. Inner Product Spaces

[DF*] We will use the most important properties of the dot product on $\mathbb{R}^n$ as axioms, which, if satisfied by the vectors in a vector space $V$, will enable us to extend the notions of length, distance, angle, and perpendicularity to general vector spaces.

**Definition 23.50: Inner Product**

An inner product on a real vector space $V$ is a function that associates a real number $< \boldsymbol{u}, \boldsymbol{v} >$ with each pair of vectors in $V$ in such a way that the following axioms are satisfied for all vectors $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ in $V$ and all scalars $k$.

1. $< \boldsymbol{u}, \boldsymbol{v} >=< \boldsymbol{v}, \boldsymbol{u} >$ [Symmetry axiom]
2. $< \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} >=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} >$ [Additivity axiom]
3. $< k\boldsymbol{u}, \boldsymbol{v} >= k < \boldsymbol{u}, \boldsymbol{v} >$ [Homogeneity axiom]
4. $< \boldsymbol{v}, \boldsymbol{v} \geq 0$ and $< \boldsymbol{v}, \boldsymbol{v} >= 0$ if and only if $\boldsymbol{v} = \mathbf{0}$ [Positivity axiom]

A real vector space with an inner product is called a real inner product space. This definition applies only to real vector spaces (vector spaces with real scalars).

[DF*] The inner product space axioms will be satisfied automatically if we define the inner product of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ to be

$$< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{u} \cdot \boldsymbol{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

This inner product is commonly called the Euclidean inner product (or the standard inner product) on $\mathbb{R}^n$ to distinguish it from other possible inner products that might be defined on $\mathbb{R}^n$. We call $\mathbb{R}^n$ with the Euclidean inner product Euclidean $n$-space.

[DF*] Inner products can be used to define notions of norm and distance in a general inner product space just as we did with dot products in $\mathbb{R}^n$.

**Definition 23.51: Norm and Distance**

If $V$ is a real inner product space, then the norm (or length) of a vector $\boldsymbol{v}$ in $V$ is denoted by $||\boldsymbol{v}||$ and is defined by

$$||\boldsymbol{v}|| = \sqrt{< \boldsymbol{v}, \boldsymbol{v} >} = \sqrt{\boldsymbol{v} \cdot \boldsymbol{u}}$$

and the distance between two vectors is denoted by $d(\boldsymbol{u}, \boldsymbol{v})$ and is defined by

$$d(\boldsymbol{u}, \boldsymbol{v}) = ||\boldsymbol{u} - \boldsymbol{v}|| = \sqrt{< \boldsymbol{u} - \boldsymbol{v}, \boldsymbol{u} - \boldsymbol{v} >} = \sqrt{(\boldsymbol{u} - \boldsymbol{v}) \cdot (\boldsymbol{u} - \boldsymbol{v})}$$

A vector of norm 1 is called a unit vector.

**Theorem 23.102: Properties of Real Inner Product Spaces**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in a real inner product space $V$, and if $k$ is a scalar, then:

(a) $||\boldsymbol{v}|| \geq 0$ with equality if and only if $\boldsymbol{v} = \mathbf{0}$.

(b) $||k\boldsymbol{v}|| = |k|||\boldsymbol{v}||$.

(c) $d(\boldsymbol{u}, \boldsymbol{v}) = d(\boldsymbol{v}, \boldsymbol{u})$.

(d) $d(\boldsymbol{u}, \boldsymbol{v}) \geq 0$ with equality if and only if $\boldsymbol{u} = \boldsymbol{v}$.

[DF*] There are various applications when we modify Euclidean inner product by weighting each term differently. If

$$w_1, w_2, \ldots, w_n$$

are positive real numbers, which we will call weights, and if $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ are vectors in $\mathbb{R}^n$, then it can be shown that the formula

$$< \boldsymbol{u}, \boldsymbol{v} >= w_1 u_1 v_1 + w_2 u_2 v_2 + \cdots + w_n u_n v_n \tag{23.141}$$

defines an inner product on $\mathbb{R}^n$ that we call the weighted Euclidean inner product with weights $w_1, w_2, \ldots, w_n$.

[DF*] **Example:**
Let $\boldsymbol{u} = (u_1, u_2)$ and $\boldsymbol{v} = (v_1, v_2)$ be vectors in $\mathbb{R}^2$. Verify that the weighted Euclidean inner product

$$< \boldsymbol{u}, \boldsymbol{v} >= 3u_1 v_1 + 2u_2 v_2$$

satisfies the four inner products axioms.
**Solution:**

**Axiom 1:**
Interchanging $\boldsymbol{u}$ and $\boldsymbol{v}$ does not change the sum on the right side, so

$$\begin{aligned} < \boldsymbol{u}, \boldsymbol{v} > &= 3u_1 v_1 + 2u_2 v_2 \\ &= 3v_1 u_1 + 2v_2 u_2 \\ &=< \boldsymbol{v}, \boldsymbol{u} > \end{aligned}$$

**Axiom 2:**
If $\boldsymbol{w} = (w_1, w_2)$, then

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &= 3(u_1 + v_1)w_1 + 2(u_2 + v_2)w_2 \\ &= 3(u_1 w_1 + v_1 w_1) + 2(u_2 w_2 + v_2 w_2) \\ &= (3u_1 w_1 + 2u_2 w_2) + (3v_1 w_1 + 2v_2 w_2) \\ &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} > \end{aligned}$$

**Axiom 3:**

$$\begin{aligned} < k\boldsymbol{u}, \boldsymbol{v} > &= 3(ku_1)v_1 + 2(ku_2)v_2 \\ &= k(3u_1 v_1 + 2u_2 v_2) \\ &= k < \boldsymbol{u}, \boldsymbol{v} > \end{aligned}$$

**Axiom 4:**

$$\begin{aligned} < \boldsymbol{v}, \boldsymbol{v} > &= 3(v_1, v_1) + 2(v_2 v_2) \\ &= 3v_1^2 + 2v_2^2 \geq 0 \end{aligned}$$

with equality if and only if $v_1 = v_2 = 0$, that is, if $\boldsymbol{v} = \mathbf{0}$.

[DF*] Suppose that some physical experiment has $n$ possible numerical outcomes

$$x_1, x_2, \ldots, x_n$$

and that a series of $m$ repetitions of the experiment yields these values with various frequencies. Specifically, suppose that $x_1$ occurs $f_1$ times, $x_2$ occurs $f_2$ times, and so forth. Since there are a total of $m$ repetitions of the experiment, it follows that

$$f_1 + f_2 + \cdots + f_n = m$$

Thus, the arithmetic average of the observed numerical values (denoted by $\overline{x}$) is

$$\overline{x} = \frac{f_1 x_1 + f_2 x_2 + \cdots + f_n x_n}{f_1 + f_2 + \cdots + f_n} = \frac{1}{m}(f_1 x_1 + f_2 x_2 + \cdots + f_n x_n) \tag{23.142}$$

if we let

$$\begin{aligned} \boldsymbol{f} &= (f_1, f_2, \ldots, f_n) \\ \boldsymbol{x} &= (x_1, x_2, \ldots, x_n) \\ w_1 = w_2 = \cdots = w_n &= 1/m \end{aligned}$$

then we can express $\overline{x}$ as the weighted Euclidean inner product

$$\overline{x} =< \boldsymbol{f}, \boldsymbol{x} >= w_1 f_1 x_1 + w_2 f_2 x_2 + \cdots + w_n f_n x_n$$

[DF*] Norm and distance depend on the inner product being used. If the inner product is changed, then the norms and distances between vectors also change.

[DF*] **Example:**
Let $\boldsymbol{u} = (1, 0)$ and $\boldsymbol{v} = (0, 1)$ be vectors in $\mathbb{R}^2$, with the Euclidean inner product we have

$$||\boldsymbol{u}|| = \sqrt{1^2 + 0^2} = 1$$

and

$$d(\boldsymbol{u}, \boldsymbol{v}) = ||\boldsymbol{u} - \boldsymbol{v}|| = ||(1, -1)|| = \sqrt{1^2 + (-1)^2} = \sqrt{2}$$

but if we change to the weighted Euclidean inner product

$$< \boldsymbol{u}, \boldsymbol{v} >= 3u_1 v_1 + 2u_2 v_2$$

$$||\boldsymbol{u}|| =< \boldsymbol{u}, \boldsymbol{u} >^{1/2}= \sqrt{3(1)(1) + 2(0)(0)} = \sqrt{3}$$

and

$$\begin{aligned} d(\boldsymbol{u}, \boldsymbol{v}) &= ||\boldsymbol{u} - \boldsymbol{v}|| \\ &=< \boldsymbol{u} - \boldsymbol{v}, \boldsymbol{u} - \boldsymbol{v} >^{1/2} \\ &=< (1, -1), (1, -1) >^{1/2} \\ &= \sqrt{3(1)(1) + 2(-1)(-1)} \\ &= \sqrt{5} \end{aligned}$$

[DF*] If $V$ is an inner product space, then the set of points in $V$ that satisfy

$$||\boldsymbol{u}|| = 1$$

is called the unit sphere or sometimes the unit circle in $V$.

**Case 1:**
Let the Euclidean inner product be

$$< \boldsymbol{u}, \boldsymbol{v} >= u_1 v_1 + u_2 v_2$$

If $\boldsymbol{u} = (x, y)$, then

$$||\boldsymbol{u}|| =< \boldsymbol{u}, \boldsymbol{u} >^{1/2} = \sqrt{x^2 + y^2}$$

so the equation of the unit circle is

$$\sqrt{x^2 + y^2} = 1$$
$$x^2 + y^2 = 1$$

the graph of this equation is a circle with radius of 1 centered at the origin.

**Figure 23.139:** *The circle in an xy-coordinate system in* $\mathbb{R}^2$ *resulted from Euclidean inner product* $< \boldsymbol{u}, \boldsymbol{v} >= u_1 v_1 + u_2 v_2$.

**Case 2:**
Let the weighted Euclidean inner product be

$$< \boldsymbol{u}, \boldsymbol{v} >= \frac{1}{9} u_1 v_1 + \frac{1}{4} u_2 v_2$$

If $\boldsymbol{u} = (x, y)$, then

$$||\boldsymbol{u}|| =< \boldsymbol{u}, \boldsymbol{u} >^{1/2} = \sqrt{\frac{1}{9} x^2 + \frac{1}{4} y^2}$$

so the equation of the unit circle is

$$\sqrt{\frac{1}{9}x^2 + \frac{1}{4}y^2} = 1$$
$$\frac{x^2}{9} + \frac{y^2}{4} = 1$$

the graph of this equation is an ellipse centered at the origin. What we see geometrically

**Figure 23.140:** *The ellipse in an xy-coordinate system in* $\mathbb{R}^2$ *resulted from weighted Euclidean inner product* $< \boldsymbol{u}, \boldsymbol{v} >= \frac{1}{9}u_1v_1 + \frac{1}{4}u_2v_2$.

here is two different shapes, the first is a circle, the second is an ellipse. Since both share the same norm of $||\boldsymbol{u}|| = 1$, the Euclidean inner product will produces the equation for a circle, while the weighted Euclidean inner product will produces the equation for an ellipse.

[DF*] The Euclidean inner product and the weighted Euclidean inner products are special cases of a general class of inner products on $\mathbb{R}^n$ called matrix inner products.

To define this class of inner products, let $\boldsymbol{u}$ and $\boldsymbol{v}$ be vectors in $\mathbb{R}^n$ that are expressed in column form, and let $A$ be an invertible $n \times n$ matrix. It can be shown that if $\boldsymbol{u} \cdot \boldsymbol{v}$ is the Euclidean inner product on $\mathbb{R}^n$, then the formula

$$< \boldsymbol{u}, \boldsymbol{v} >= A\boldsymbol{u} \cdot A\boldsymbol{v} \tag{23.143}$$

also defines an inner product; it is called the inner product on $\mathbb{R}^n$ generated by $A$.

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are in column form, then $\boldsymbol{u} \cdot \boldsymbol{v}$ can be written as $\boldsymbol{v}^T\boldsymbol{u}$ from which it follows that

$$< \boldsymbol{u}, \boldsymbol{v} >= (A\boldsymbol{v})^T A\boldsymbol{u}$$

or, equivalently as

$$< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{v}^T A^T A \boldsymbol{u} \tag{23.144}$$

[DF*] The standard Euclidean inner product on $\mathbb{R}^n$ is generated by the $n \times n$ identity matrix, since setting $A = I$ yields

$$< \boldsymbol{u}, \boldsymbol{v} >= I\boldsymbol{u} \cdot I\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{v}$$

and the weighted Euclidean inner product

$$< \boldsymbol{u}, \boldsymbol{v} >= w_1 u_1 v_1 + w_2 u_2 v_2 + \cdots + w_n u_n v_n$$

is generated by the matrix

$$A = \begin{bmatrix} \sqrt{w_1} & 0 & 0 & \dots & 0 \\ 0 & \sqrt{w_2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & \sqrt{w_n} \end{bmatrix}$$

matrix $A$ is a diagonal matrix with entries of $\sqrt{w_i}$ at the row-$i$th, column-$i$th, and zeros elsewhere.

We can observe that $A^T A$ will become $n \times n$ diagonal matrix whose diagonal entries are the weights $w_1, w_2, \dots, w_n$.

[DF*] The weighted Euclidean inner product $< \boldsymbol{u}, \boldsymbol{v} >= 3u_1 v_1 + 2u_2 v_2$ is the inner product on $\mathbb{R}^2$ generated by matrix

$$A = \begin{bmatrix} \sqrt{3} & 0 \\ 0 & \sqrt{2} \end{bmatrix}$$

[DF*] If $U$ and $V$ are $n \times n$ matrices, then the formula

$$< U, V >= \operatorname{tr}\left(U^T V\right) \tag{23.145}$$

defines an inner product on the vector space $M_{nn}$. This can be proved by confirming that the four inner product space axioms are satisfied.

For the $2 \times 2$ matrices we will have

$$U = \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \end{bmatrix}, \quad \text{and} \quad V = \begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix}$$

This yields

$$< U, V >= \operatorname{tr}\left(U^T V\right) = u_1 v_1 + u_2 v_2 + u_3 v_3 + u_4 v_4$$

which is, just the dot product of the corresponding entries in the two matrices. The norm of a matrix $U$ relative tot his inner product is

$$||U|| =< U, U >^{1/2} = \sqrt{u_1^2 + u_2^2 + u_3^2 + u_4^2}$$

[DF*] If

$$\boldsymbol{p} = a_0 + a_1 x + \cdots + a_n x^n$$
$$\boldsymbol{q} = b_0 + b_1 x + \cdots + b_n x^n$$

are polynomials in $P_n$, then the following formula defines an inner product on $P_n$ that we will call the standard inner product on this space:

$$< \boldsymbol{p}, \boldsymbol{q} >= a_0 b_0 + a_1 b_1 + \cdots + a_n b_n \tag{23.146}$$

The norm of a polynomial $\boldsymbol{p}$ relative to this inner product is

$$||\boldsymbol{p}|| = \sqrt{< \boldsymbol{p}, \boldsymbol{p} >} = \sqrt{a_0^2 + a_1^2 + \cdots + a_n^2}$$

[DF*] If

$$\begin{aligned} \boldsymbol{p} &= a_0 + a_1 x + \cdots + a_n x^n \\ \boldsymbol{q} &= b_0 + b_1 x + \cdots + b_n x^n \end{aligned}$$

are polynomials in $P_n$, and if $x_0, x_1, \ldots, x_n$ are distinct real numbers (sample points), then the formula

$$< \boldsymbol{p}, \boldsymbol{q} >= p(x_0)q(x_0) + p(x_1)q(x_1) + \cdots + p(x_n)q(x_n) \tag{23.147}$$

defines an inner product on $P_n$ called the evaluation inner product at $x_0, x_1, \ldots, x_n$. Algebraically, this can be viewed as the dot product in $\mathbb{R}^n$ of the $n$-tuples

$$(p(x_0), p(x_1), \ldots, p(x_n)) \quad \text{and} \quad (q(x_0), q(x_1), \ldots, q(x_n))$$

and hence the first three inner product axioms follow from properties of the dot product. The fourth inner product axiom follows from the fact that

$$< \boldsymbol{p}, \boldsymbol{p} >= [p(x_0)]^2 + [p(x_1)]^2 + \cdots + [p(x_n)]^2 \geq 0$$

with equality holding if and only if

$$p(x_0) = p(x_1) = \cdots = p(x_n) = 0$$

But a nonzero polynomial of degree $n$ or less can have at most $n$ distinct roots, so it must be that $\boldsymbol{p} = \boldsymbol{0}$, which proves that the fourth inner product axiom holds.

The norm of a polynomial $\boldsymbol{p}$ relative to the evaluation inner product is

$$||\boldsymbol{p}|| =< \boldsymbol{p}, \boldsymbol{p} >^{1/2}= \sqrt{[p(x_0)]^2 + [p(x_1)]^2 + \cdots + [p(x_n)]^2} \tag{23.148}$$

[DF*] Let $\boldsymbol{f} = f(x)$ and $\boldsymbol{g} = g(x)$ be two functions in $C[a, b]$ and define

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_a^b f(x)g(x)\, dx \tag{23.149}$$

this formula defines an inner product on $C[a, b]$ by verifying the four inner product axioms for functions $\boldsymbol{f} = f(x)$, $\boldsymbol{g} = g(x)$, and $\boldsymbol{h} = h(x)$ in $C[a, b]$:

1. Axiom 1:

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_a^b f(x)g(x)\, dx = \int_a^b g(x)f(x)\, dx =< \boldsymbol{g}, \boldsymbol{f} >$$

which proves that Axiom 1 holds.

2. Axiom 2:

$$\begin{aligned} < \boldsymbol{f}+\boldsymbol{g}, \boldsymbol{h} > &= \int_a^b (f(x)+g(x))\, h(x)\, dx \\ &= \int_a^b f(x)h(x)\, dx + \int_a^b g(x)h(x)\, dx \\ &=< \boldsymbol{f}, \boldsymbol{h} > + < \boldsymbol{g}, \boldsymbol{h} > \end{aligned}$$

which proves that Axiom 2 holds.

3. Axiom 3:

$$\begin{aligned} < k\boldsymbol{f}, \boldsymbol{g} > &= \int_a^b kf(x)g(x)\, dx \\ &= k\int_a^b f(x)g(x)\, dx \\ &= k < \boldsymbol{f}, \boldsymbol{g} > \end{aligned}$$

which proves that Axiom 3 holds.

4. Axiom 4:
If $\boldsymbol{f} = f(x)$ is any function in $C[a,b]$, then

$$< \boldsymbol{f}, \boldsymbol{f} > = \int_a^b f^2(x)\, dx \geq 0$$

since $f^2(x) \geq 0$ for all $x$ in the interval $[a,b]$. Moreover because $f$ is continuous on $[a,b]$, the equality holds if and only if the function $f$ is identically zero on $[a,b]$, that is, if and only if $\boldsymbol{f} = \mathbf{0}$; and this proves that Axiom 4 holds.

[DF*] If $C[a,b]$ has the inner product that was defined, then the norm of a function $\boldsymbol{f} = f(x)$ relative to this inner product is

$$||\boldsymbol{f}|| =< \boldsymbol{f}, \boldsymbol{f} >^{1/2} = \sqrt{\int_a^b f^2(x)\, dx} \tag{23.150}$$

and the unit sphere in this space consists of all functions $\boldsymbol{f}$ in $C[a,b]$ that satisfy the equation

$$\int_a^b f^2(x)\, dx = 1$$

Note that the vector space $P_n$ is a subspace of $C[a,b]$ because polynomials are continuous functions.

[DF*] Recall from calculus that the arc length of a curve $y = f(x)$ over an interval $[a,b]$ is given by the formula

$$L = \int_a^b \sqrt{1+[f'(x)]^2}\, dx$$

the arc length is not the same with $||\boldsymbol{f}||$, which is the length of $\boldsymbol{f}$ when $\boldsymbol{f}$ is viewed as a vector in $C[a,b]$.

**Theorem 23.103: Algebraic Properties of Inner Products**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in a real inner product space $V$, and if $k$ is a scalar, then:

(a) $< \mathbf{0}, \boldsymbol{v} >=< \boldsymbol{v}, \mathbf{0} >= 0$

(b) $< \boldsymbol{u}, \boldsymbol{v} + \boldsymbol{w} >=< \boldsymbol{u}, \boldsymbol{v} > + < \boldsymbol{u}, \boldsymbol{w} >$

(c) $< \boldsymbol{p}, \boldsymbol{v} - \boldsymbol{w} >=< \boldsymbol{u}, \boldsymbol{v} > - < \boldsymbol{u}, \boldsymbol{w} >$

(d) $< \boldsymbol{u} - \boldsymbol{v}, \boldsymbol{w} >=< \boldsymbol{u}, \boldsymbol{w} > - < \boldsymbol{v}, \boldsymbol{w} >$

(e) $k < \boldsymbol{u}, \boldsymbol{v} >=< \boldsymbol{u}, k\boldsymbol{v} >$

[DF*] **Example:**
Let $\boldsymbol{u} = (u_1, u_2, u_3)$ and $\boldsymbol{v} = (v_1, v_2, v_3)$. Determine which of the following are inner products on $\mathbb{R}^3$.

(a) $< \boldsymbol{u}, \boldsymbol{v} >= u_1v_1 + u_3v_3$

(b) $< \boldsymbol{u}, \boldsymbol{v} >= u_1^2v_1^2 + u_2^2v_2^2 + u_3^2v_3^2$

(c) $< \boldsymbol{u}, \boldsymbol{v} >= 2u_1v_1 + u_2v_2 + 4u_3v_3$

(d) $< \boldsymbol{u}, \boldsymbol{v} >= u_1v_1 - u_2v_2 + u_3v_3$

**Solution:**

(a) Axiom 1:

$$\begin{aligned} < \boldsymbol{u}, \boldsymbol{v} > &= u_1v_1 + u_3v_3 \\ &= v_1u_1 + v_3u_3 \\ &=< \boldsymbol{v}, \boldsymbol{u} > \end{aligned}$$

which proves that axiom 1 holds.

Axiom 2: If $\boldsymbol{w} = (w_1, w_2, w_3)$, then

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &= (u_1 + v_1)w_1 + (u_3 + v_3)w_3 \\ &= u_1w_1 + v_1w_1 + u_3w_3 + v_3w_3 \\ &= (u_1w_1 + u_3w_3) + (v_1w_1 + v_3w_3) \\ &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} > \end{aligned}$$

which proves that axiom 2 holds.

Axiom 3: If $k$ is any scalar, then

$$\begin{aligned} < k\boldsymbol{u}, \boldsymbol{v} > &= (ku_1)v_1 + (ku_3)v_3 \\ &= k(u_1v_1 + u_3v_3) \\ &= k < \boldsymbol{u}, \boldsymbol{v} > \end{aligned}$$

which proves that axiom 3 holds.

Axiom 4:

$$< \boldsymbol{v}, \boldsymbol{v} > = v_1 v_1 + v_3 v_3 = v_1^2 + v_3^2 \geq 0$$

with equality $< \boldsymbol{v}, \boldsymbol{v} >= 0$ if and only if $\boldsymbol{v} = \mathbf{0}$. All the axioms hold, thus $< \boldsymbol{u}, \boldsymbol{v} >= u_1 v_1 + u_3 v_3$ is an inner product.

(b) Axiom 1:

$$\begin{aligned} < \boldsymbol{u}, \boldsymbol{v} > &= u_1^2 v_1^2 + u_2^2 v_2^2 + u_3^2 v_3^2 \\ &= v_1^2 u_1^2 + v_2^2 u_2^2 + v_3^2 u_3^2 \\ &=< \boldsymbol{v}, \boldsymbol{u} > \end{aligned}$$

which proves that axiom 1 holds.

Axiom 2: If $\boldsymbol{w} = (w_1, w_2, w_3)$, then

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &= (u_1 + v_1)^2 w_1^2 + (u_2 + v_2)^2 w_2^2 + (u_3 + v_3)^2 w_3^2 \\ &= u_1^2 w_1^2 + u_2^2 w_2^2 + u_3^2 w_3^2 + v_1^2 w_1^2 + v_2^2 w_2^2 + v_3^2 w_3^2 \\ &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} > \end{aligned}$$

which proves that axiom 2 holds.

Axiom 3: If $k$ is any scalar, then

$$\begin{aligned} < k\boldsymbol{u}, \boldsymbol{v} > &= (ku_1)^2 v_1^2 + (ku_2)^2 v_2^2 + (ku_3)^2 v_3^2 \\ &= k^2 \left( u_1^2 v_1^2 + u_2^2 v_2^2 + u_3^2 v_3^2 \right) \neq k < \boldsymbol{u}, \boldsymbol{v} > \end{aligned}$$

Axiom 3 is not satisfied for $< \boldsymbol{u}, \boldsymbol{v} >= u_1^2 v_1^2 + u_2^2 v_2^2 + u_3^2 v_3^2$, thus it is not an inner products on $\mathbb{R}^3$.

(c) Axiom 1:

$$\begin{aligned} < \boldsymbol{u}, \boldsymbol{v} > &= 2u_1 v_1 + u_2 v_2 + 4u_3 v_3 \\ &= 2v_1 u_1 + v_2 u_2 + 4v_3 u_3 \\ &=< \boldsymbol{v}, \boldsymbol{u} > \end{aligned}$$

which proves that axiom 1 holds.

Axiom 2: If $\boldsymbol{w} = (w_1, w_2, w_3)$, then

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &= 2(u_1 + v_1) w_1 + (u_2 + v_2) w_2 + 4(u_3 + v_3) w_3 \\ &= 2u_1 w_1 + u_2 w_2 + 4u_3 w_3 + 2v_1 w_1 + v_2 w_2 + 4v_3 w_3 \\ &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} > \end{aligned}$$

which proves axiom 2 holds.

Axiom 3: If $k$ is any scalar, then

$$\begin{aligned} < k\boldsymbol{u}, \boldsymbol{v} > &= 2(ku_1) v_1 + (ku_2) v_2 + 4(ku_3) v_3 \\ &= k \left( 2u_1 v_1 + u_2 v_2 + 4u_3 v_3 \right) \\ &= k < \boldsymbol{u}, \boldsymbol{v} > \end{aligned}$$

which proves axiom 3 holds.

Axiom 4:

$$\begin{aligned} < \boldsymbol{v}, \boldsymbol{v} > &= 2v_1v_1 + v_2v_2 + 4v_3v_3 \\ &= 2v_1^2 + v_2^2 + 4v_3^2 \end{aligned}$$

$< \boldsymbol{v}, \boldsymbol{v} >= 0$ if and only if $\boldsymbol{v} = \mathbf{0}$. All the axioms hold, thus $< \boldsymbol{u}, \boldsymbol{v} >= 2u_1v_1 + u_2v_2 + 4u_3v_3$ is an inner product on $\mathbb{R}^3$.

(d) Axiom 1:

$$\begin{aligned} < \boldsymbol{u}, \boldsymbol{v} > &= u_1v_1 - u_2v_2 + u_3v_3 \\ &= v_1u_1 - v_2u_2 + v_3u_3 \\ &=< \boldsymbol{v}, \boldsymbol{u} > \end{aligned}$$

which proves that axiom 1 holds.

Axiom 2: If $\boldsymbol{w} = (w_1, w_2, w_3)$, then

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &= (u_1 + v_1)w_1 - (u_2 + v_2)w_2 + (u_3 + v_3)w_3 \\ &= (u_1w_1 - u_2w_2 + u_3w_3) + (v_1w_1 - v_2w_2 + v_3w_3) \\ &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} > \end{aligned}$$

which proves that axiom 3 holds.

Axiom 3: If $k$ is any scalar, then

$$\begin{aligned} < k\boldsymbol{u}, \boldsymbol{v} > &= (ku_1)v_1 - (ku_2)v_2 + (ku_3)v_3 \\ &= k(u_1v_1 - u_2v_2 + u_3v_3) \\ &= k < \boldsymbol{u}, \boldsymbol{v} > \end{aligned}$$

Axiom 4:

$$\begin{aligned} < \boldsymbol{v}, \boldsymbol{v} > &= v_1v_1 - v_2v_2 + v_3v_3 \\ &= v_1^2 - v_2^2 + v_3^2 \end{aligned}$$

if $\boldsymbol{v} = \mathbf{0}$, then $< \boldsymbol{v}, \boldsymbol{v} > neq0$ therefore the axiom 4 is not satisfied and $< \boldsymbol{u}, \boldsymbol{v} >= u_1v_1 - u_2v_2 + u_3v_3$ is not an inner product on $\mathbb{R}^3$.

[DF*] **Example:**
Show that the following identity holds for vectors in any inner product space.

$$||\boldsymbol{u} + \boldsymbol{v}||^2 + ||\boldsymbol{u} - \boldsymbol{v}||^2 = 2||\boldsymbol{u}||^2 + 2||\boldsymbol{v}||^2$$

**Solution:**

$$\begin{aligned} ||\boldsymbol{u} + \boldsymbol{v}||^2 + ||\boldsymbol{u} - \boldsymbol{v}||^2 &=< \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{u} + \boldsymbol{v} > + < \boldsymbol{u} - \boldsymbol{v}, \boldsymbol{u} - \boldsymbol{v} > \\ &=< \boldsymbol{u}, \boldsymbol{u} + \boldsymbol{v} > + < \boldsymbol{v}, \boldsymbol{u} + \boldsymbol{v} > + < \boldsymbol{u}, \boldsymbol{u} - \boldsymbol{v} > - < \boldsymbol{v}, \boldsymbol{u} - \boldsymbol{v} > \\ &=< \boldsymbol{u}, \boldsymbol{u} > + < \boldsymbol{u}, \boldsymbol{v} > + < \boldsymbol{v}, \boldsymbol{u} > + < \boldsymbol{v}, \boldsymbol{v} > \\ &+ < \boldsymbol{u}, \boldsymbol{u} > - < \boldsymbol{u}, \boldsymbol{v} > - < \boldsymbol{v}, \boldsymbol{u} > + < \boldsymbol{v}, \boldsymbol{v} > \\ &= 2 < \boldsymbol{u}, \boldsymbol{u} > +2 < \boldsymbol{u}, \boldsymbol{v} > -2 < \boldsymbol{u}, \boldsymbol{v} > +2 < \boldsymbol{v}, \boldsymbol{v} > \\ &= 2 < \boldsymbol{u}, \boldsymbol{u} > +2 < \boldsymbol{v}, \boldsymbol{v} > \\ &= 2||\boldsymbol{u}||^2 + 2||\boldsymbol{v}||^2 \end{aligned}$$

[DF*] **Example:**
Show that the following identity holds for vectors in any inner product space.

$$< \boldsymbol{u}, \boldsymbol{v} >= \frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2$$

**Solution:**

$$\begin{aligned}
< \boldsymbol{u}, \boldsymbol{v} > &= \frac{1}{4}||\boldsymbol{u}+\boldsymbol{v}||^2 - \frac{1}{4}||\boldsymbol{u}-\boldsymbol{v}||^2 \\
&= \frac{1}{4} < \boldsymbol{u}+\boldsymbol{v}, \boldsymbol{u}+\boldsymbol{v} > - \frac{1}{4} < \boldsymbol{u}-\boldsymbol{v}, \boldsymbol{u}-\boldsymbol{v} > \\
&= \frac{1}{4}\left(< \boldsymbol{u}, \boldsymbol{u}+\boldsymbol{v} > + < \boldsymbol{v}, \boldsymbol{u}+\boldsymbol{v} >\right) - \frac{1}{4}\left(< \boldsymbol{u}, \boldsymbol{u}-\boldsymbol{v} > - < \boldsymbol{v}, \boldsymbol{u}-\boldsymbol{v} >\right) \\
&= \frac{1}{4} < \boldsymbol{u}, \boldsymbol{u} > + \frac{1}{4} < \boldsymbol{u}, \boldsymbol{v} > + \frac{1}{4} < \boldsymbol{v}, \boldsymbol{u} > + \frac{1}{4} < \boldsymbol{v}, \boldsymbol{v} > \\
&- \frac{1}{4} < \boldsymbol{u}, \boldsymbol{u} > + \frac{1}{4} < \boldsymbol{u}, \boldsymbol{v} > + \frac{1}{4} < \boldsymbol{v}, \boldsymbol{u} > - \frac{1}{4} < \boldsymbol{v}, \boldsymbol{v} > \\
&=< \boldsymbol{u}, \boldsymbol{v} >
\end{aligned}$$

[DF*] **Example:**
The definition of a complex inner product on a complex vector space $V$ is identical to Definition of real inner product except that scalars are allowed to be complex numbers, and Axiom 1 is replaced by

$$< \boldsymbol{u}, \boldsymbol{v} >= \overline{< \boldsymbol{v}, \boldsymbol{u} >}$$

The remaining axioms are unchanged. A complex vector space with a complex inner product is called a complex inner product space. Prove that if $V$ is a complex inner product space then $< \boldsymbol{u}, k\boldsymbol{v} >= \bar{k} < \boldsymbol{u}, \boldsymbol{v} >$.
**Solution:**
We will only need axiom 1 and 3 to prove it.

$$\begin{aligned}
< \boldsymbol{u}, k\boldsymbol{v} > &= \overline{< k\boldsymbol{v}, \boldsymbol{u} >} \quad \text{(Axiom 1)} \\
&= \bar{k}\overline{< \boldsymbol{v}, \boldsymbol{u} >} \quad \text{(Axiom 3)} \\
&= \bar{k} < \boldsymbol{u}, \boldsymbol{v} > \quad \text{(Axiom 1)}
\end{aligned}$$

[DF*] The angle $\theta$ between two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^n$ is

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||}\right) \tag{23.151}$$

it followed from the Cauchy-Schwarz inequality that

$$-1 \leq \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||} \leq 1 \tag{23.152}$$

as required for the inverse cosine to be defined.

**Theorem 23.104: Cauchy-Schwarz inequality**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in a real inner product space $V$, then

$$| < \boldsymbol{u}, \boldsymbol{v} > | \leq ||\boldsymbol{u}||||\boldsymbol{v}|| \tag{23.153}$$

[DF*] The following two alternative forms of the Cauchy-Schwarz inequality are useful to know:

$$< \boldsymbol{u}, \boldsymbol{v} >^2 \leq < \boldsymbol{u}, \boldsymbol{u} >< \boldsymbol{v}, \boldsymbol{v} > \tag{23.154}$$

$$< \boldsymbol{u}, \boldsymbol{v} >^2 \leq ||\boldsymbol{u}||^2 ||\boldsymbol{v}||^2 \tag{23.155}$$

[DF*] From the Cauchy-Schwarz inequality

$$-1 \leq \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||} \leq 1$$

we will have

$$-1 \leq \frac{< \boldsymbol{u}, \boldsymbol{v} >}{||\boldsymbol{u}||||\boldsymbol{v}||} \leq 1 \tag{23.156}$$

there is a unique angle $\theta$ in radian measure for which

$$\cos\ \theta = \frac{< \boldsymbol{u}, \boldsymbol{v} >}{||\boldsymbol{u}||||\boldsymbol{v}||}, \quad 0 \leq \theta \leq \pi \tag{23.157}$$

this enables us to define the angle $\theta$ between $\boldsymbol{u}$ and $\boldsymbol{v}$ to be

$$\theta = \cos^{-1}\left(\frac{< \boldsymbol{u}, \boldsymbol{v} >}{||\boldsymbol{u}||||\boldsymbol{v}||}\right) \tag{23.158}$$

Not to be confused with the angle between two vectors in Euclidean vector spaces that is definitely use Euclidean inner product $(\boldsymbol{u} \cdot \boldsymbol{v})$, the numerator here is the inner product which can be Euclidean inner product, weighted Euclidean inner product, or inner products generated by matrices.

**Theorem 23.105: Properties of Length and DIstance in General Inner Product Spaces**

If $\boldsymbol{u}, \boldsymbol{v}$, and $\boldsymbol{w}$ are vectors in a real inner product space $V$, and if $k$ is any scalar, then

(a) $||\boldsymbol{u} + \boldsymbol{v}|| \leq ||\boldsymbol{u}|| + ||\boldsymbol{v}||$ (Triangle inequality for vectors)

(b) $d(\boldsymbol{u}, \boldsymbol{v}) \leq d(\boldsymbol{u}, \boldsymbol{w}) + d(\boldsymbol{w}, \boldsymbol{v})$ (Triangle inequality for distances)

[DF*] There is only an occasional need to compute angles in vector spaces other than $\mathbb{R}^2$ and $\mathbb{R}^3$.

A problem of more interest in general vector spaces is ascertaining whether the angle between vectors is $\frac{\pi}{2}$. If $\boldsymbol{u}$ and $\boldsymbol{v}$ are nonzero vectors, then the angle between them is $\theta = \frac{\pi}{2}$ if and only if $< \boldsymbol{u}, \boldsymbol{v} >= 0$.s

**Definition 23.52: Orthogonality**

Two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in an inner product space are called orthogonal if $< \boldsymbol{u}, \boldsymbol{v} >= 0$.

[DF*] **Example:**
The vectors $\boldsymbol{u} = (1, 1)$ and $\boldsymbol{v} = (1, -1)$ are orthogonal with respect to the Euclidean inner product on $\mathbb{R}^2$, since

$$\boldsymbol{u} \cdot \boldsymbol{v} = (1)(1) + (1)(-1) = 0$$

However, they are not orthogonal with respect to the weighted Euclidean inner product $< \boldsymbol{u}, \boldsymbol{v} >= 3u_1v_1 + 5u_2v_2$, since

$$< \boldsymbol{u}, \boldsymbol{v} >= 3(1)(1) + 5(1)(-1) = -2 \neq 0$$

[DF*] **Example:**
If $M_{22}$ has the inner product of

$$< U, V >= \text{tr}(U^T V)$$

then the matrices

$$U = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}$$

are orthogonal since

$$< U, V >= u_1 v_1 + u_2 v_2 + u_3 v_3 + u_4 v_4 = 1(0) + 0(2) + 1(0) + 1(0)$$

[DF*] **Example:**
Let $P_2$ have the inner product

$$< \boldsymbol{p}, \boldsymbol{q} >= \int_{-1}^{1} p(x) q(x)\, dx$$

and let $\boldsymbol{p} = x$ and $\boldsymbol{q} = x^2$. Then

$$\begin{aligned} ||\boldsymbol{p}|| &=< \boldsymbol{p}, \boldsymbol{p} >^{1/2} = \left[ \int_{-1}^{1} xx\, dx \right] = \sqrt{\frac{2}{3}} \\ ||\boldsymbol{q}|| &=< \boldsymbol{q}, \boldsymbol{q} >^{1/2} = \left[ \int_{-1}^{1} x^2 x^2\, dx \right] = \sqrt{\frac{2}{5}} \\ < \boldsymbol{p}, \boldsymbol{q} > &= \int_{-1}^{1} xx^2\, dx = 0 \end{aligned}$$

Because $< \boldsymbol{p}, \boldsymbol{q} >= 0$, the vectors $\boldsymbol{p} = x$ and $\boldsymbol{q} = x^2$ are orthogonal relative to the given inner product.

**Theorem 23.106: Generalized Theorem of Pythagoras**

If $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal vectors in an inner product space, then

$$||\boldsymbol{u} + \boldsymbol{v}||^2 = ||\boldsymbol{u}|| + ||\boldsymbol{v}||^2$$

[DF*] **Example:**
Let $P_2$ have the inner product

$$< \boldsymbol{p}, \boldsymbol{q} >= \int_{-1}^{1} p(x) q(x)\, dx$$

and let $\boldsymbol{p} = x$ and $\boldsymbol{q} = x^2$. Then

$$\begin{aligned} ||\boldsymbol{p} + \boldsymbol{q}||^2 &=< \boldsymbol{p} + \boldsymbol{q}, \boldsymbol{p} + \boldsymbol{q} > \\ &= \int_{-1}^{1} (x + x^2)(x + x^2)\, dx \\ &= \int_{-1}^{1} x^2 + 2x^3 + x^4\, dx \\ &= \frac{16}{15} \\ &= ||\boldsymbol{p}||^2 + ||\boldsymbol{q}||^2 \end{aligned}$$

We have computed $||\boldsymbol{p}||$ and $||\boldsymbol{q}||$, so we can check whether theorem of Pythagoras applied in this inner product of $P_2$

$$\begin{aligned} ||\boldsymbol{p}||^2 + ||\boldsymbol{q}||^2 &= \left(\sqrt{\frac{2}{3}}\right)^2 + \left(\sqrt{\frac{2}{5}}\right)^2 \\ &= \frac{16}{15} \\ &= ||\boldsymbol{p} + \boldsymbol{q}||^2 \end{aligned}$$

**Definition 23.53: Orthogonal Complements**

If $W$ is a subspace of an inner product space $V$, then the set of all vectors in $V$ that are orthogonal to every vector in $W$ is called the orthogonal complement of $W$ and is denoted by the symbol $W^{\perp}$.

**Theorem 23.107: Orthogonality in Subspace of Inner Product Space**

If $W$ is a subspace of an inner product space $V$, then:

(a) $W^{\perp}$ is a subspace of $V$.

(b) $W \cap W^{\perp} = \{\mathbf{0}\}$

[DF*] The set $W^{\perp}$ contains at least the zero vector (zero vector is perpendicular toward all vectors in $\mathbb{R}^n$), since

$$< \mathbf{0}, \boldsymbol{w} >= 0$$

for every vector $\boldsymbol{w}$ in $W$. Thus, we need to show that $W^{\perp}$ is closed under addition and scalar multiplication. To do this, suppose that $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $W^{\perp}$, so that for every vector $\boldsymbol{w}$ in $W$ we have $< \boldsymbol{u}, \boldsymbol{w} >= 0$ and $< \boldsymbol{v}, \boldsymbol{w} >= 0$. It follows from the additivity and homogeneity axioms of inner products that

$$\begin{aligned} < \boldsymbol{u} + \boldsymbol{v}, \boldsymbol{w} > &=< \boldsymbol{u}, \boldsymbol{w} > + < \boldsymbol{v}, \boldsymbol{w} >= 0 + 0 = 0 \\ < k\boldsymbol{u}, \boldsymbol{w} > &= k < \boldsymbol{u}, \boldsymbol{w} >= k(0) = 0 \end{aligned}$$

since $\mathbf{0}$ is in $W$ and $W^{\perp}$, thus it proves that $\boldsymbol{u} + \boldsymbol{v}$ and $k\boldsymbol{u}$ are in $W^{\perp}$.

If there exists a vector $\boldsymbol{z}$ in both $W$ and $W^{\perp}$, then $\boldsymbol{z}$ is orthogonal to itself; that is,

$$< \boldsymbol{z}, \boldsymbol{z} >= 0$$

It follows from the positivity axiom for inner products that

$$\boldsymbol{z} = \mathbf{0}$$

**Theorem 23.108: Orthogonal Complement of $W^{\perp}$**

If $W$ is a subspace of a finite-dimensional inner product space $V$, then the orthogonal complement of $W^{\perp}$ is $W$; that is,

$$(W^{\perp})^{\perp} = W$$

[DF*] **Example:**
Let $V$ be an inner product space. Show that if $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal unit vectors in $V$, then $||\boldsymbol{u}-\boldsymbol{v}|| = \sqrt{2}$.
**Solution:**
Orthogonal unit vector means that

$$\begin{aligned} ||\boldsymbol{u}|| &= 1 \\ ||\boldsymbol{v}|| &= 1 \\ < \boldsymbol{u}, \boldsymbol{v} > &= 0 \end{aligned}$$

$$\begin{aligned} ||\boldsymbol{u}-\boldsymbol{v}||^2 &=< \boldsymbol{u}-\boldsymbol{v}, \boldsymbol{u}-\boldsymbol{v} > \\ &=< \boldsymbol{u}, \boldsymbol{u}-\boldsymbol{v} > - < \boldsymbol{v}, \boldsymbol{u}-\boldsymbol{v} > \\ &=< \boldsymbol{u}, \boldsymbol{u} > - < \boldsymbol{u}, \boldsymbol{v} > - < \boldsymbol{v}, \boldsymbol{u} > + < \boldsymbol{v}, \boldsymbol{v} > \\ &= ||\boldsymbol{u}||^2 + ||\boldsymbol{v}||^2 \\ &= 2 \end{aligned}$$

Thus $||\boldsymbol{u}-\boldsymbol{v}|| = \sqrt{2}$.

[DF*] **Example:**
Let $V$ be an inner product space. Show that if $\boldsymbol{w}$ is orthogonal to each of the vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, then it is orthogonal to $k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2$ for all scalars $k_1$ and $k_2$. Interpret this result geometrically in the case where $V$ is $\mathbb{R}^3$ with the Euclidean inner product.
**Solution:**
We know that

$$\begin{aligned} < \boldsymbol{w}, \boldsymbol{u}_1 > &= 0 \\ < \boldsymbol{w}, \boldsymbol{u}_2 > &= 0 \end{aligned}$$

thus

$$\begin{aligned} < \boldsymbol{w}, k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 > &=< k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2, \boldsymbol{w} > \\ &=< k_1\boldsymbol{u}_1, \boldsymbol{w} >< k_2\boldsymbol{u}_2, \boldsymbol{w} > \\ &= k_1 < \boldsymbol{u}_1, \boldsymbol{w} > + k_2 < \boldsymbol{u}_2, \boldsymbol{w} > \\ &= 0 + 0 = 0 \end{aligned}$$

it is proven that $\boldsymbol{w}$ is orthogonal to $k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2$ for all scalars $k_1$ and $k_2$ in an inner product space $V$.

[DF*] **Example:**
Let $V$ be an inner product space. Show that if $\boldsymbol{w}$ is orthogonal to each of the vectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r$, then it is orthogonal to every vector in $\text{span}\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$.
**Solution:**

$$\text{span}\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\} = k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 + \cdots + k_r\boldsymbol{u}_r$$

where $k_1, k_2, \ldots, k_r$ are arbitrary scalars. Let

$$v \in \text{span}\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$$

then

$$\begin{aligned} < \boldsymbol{w}, \boldsymbol{v} > &=< \boldsymbol{w}, k_1\boldsymbol{u}_1 + k_2\boldsymbol{u}_2 + \cdots + k_r\boldsymbol{u}_r > \\ &= k_1 < \boldsymbol{w}, \boldsymbol{u}_1 > + k_2 < \boldsymbol{w}, \boldsymbol{u}_2 > + \cdots + k_r < \boldsymbol{w}, \boldsymbol{u}_r > \\ &= 0 + 0 + \cdots + 0 \\ &= 0 \end{aligned}$$

Thus if $\boldsymbol{w}$ is orthogonal to each vector $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r$, then it must be orthogonal to every vector in span$\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$.

[DF*] **Example:**
Let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ be a basis for an inner product space $V$. Show that the zero vector is the only vector in $V$ that is orthogonal to all of the basis vectors.
**Solution:**

Suppose that $\boldsymbol{w}$ is orthogonal to every basis vector in $V$

$$\begin{gathered} \boldsymbol{w} \perp \boldsymbol{v}_1 \\ \boldsymbol{w} \perp \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{w} \perp \boldsymbol{v}_r \end{gathered}$$

If $\boldsymbol{w}$ is in $V$ then it will be a linear combination of the basis vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$, from the axiom 4 of the inner product definition, we know that

$$< \boldsymbol{v}, \boldsymbol{v} >= 0 \iff \boldsymbol{v} = \mathbf{0}$$

If there exists a vector $\boldsymbol{w} \neq \mathbf{0}$ such that

$$< \boldsymbol{w}, \boldsymbol{v} > = 0$$

we will expand the left hand side

$$< k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r, \boldsymbol{v} > = k_1 < \boldsymbol{v}_1, \boldsymbol{v} > + k_2 < \boldsymbol{v}_2, \boldsymbol{v} > + \cdots + k_r < \boldsymbol{v}_r, \boldsymbol{v} >$$

with $k_1, k_2, \ldots, k_r$ arbitrary scalars. It is a contradiction to make the equation above to be equal to 0 if $\boldsymbol{w} \neq \mathbf{0}$, so from theorem we know that there are only two scenarios:

$$\boldsymbol{w} \notin V$$

meaning that $\boldsymbol{w}$ is not a linear combination of the basis vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$, or

$$\boldsymbol{w} = \mathbf{0}$$

therefore $\mathbf{0}$ is the only vector in $V$ that is orthogonal to all of the basis vectors in $V$.

[DF*] **Example:**
Let $\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k\}$ be a basis for a subspace $W$ of $V$. Show that $W^\perp$ consists of all vectors in $V$ that are orthogonal to every basis vector.
**Solution:**
Suppose that $\boldsymbol{v}$ is orthogonal to every basis vector. Then $\boldsymbol{v}$ is orthogonal to the span of the set of basis vectors, which is all of $W$, hence $\boldsymbol{v}$ must be in $W^\perp$. If $\boldsymbol{v}$ is not orthogonal to every basis vector, then $\boldsymbol{v}$ clearly cannot be in $W^\perp$. Thus $W^\perp$ consists of all vectors orthogonal to every basis vector.

[DF*] **Example:**
Prove: If $\boldsymbol{u}$ and $\boldsymbol{v}$ are $n \times 1$ matrices and $A$ is an $n \times n$ matrix, then

$$(\boldsymbol{v}^T A^T A\boldsymbol{u})^2 \leq (\boldsymbol{u}^T A^T A\boldsymbol{u})(\boldsymbol{v}^T A^T A\boldsymbol{v})$$

**Solution:**
We define the inner product as

$$< \boldsymbol{u}, \boldsymbol{v} >= (\boldsymbol{v}^T A^T A\boldsymbol{u})$$

then use the Cauchy-Schwarz inequality

$$\begin{aligned}
| < \boldsymbol{u}, \boldsymbol{v} > | &\leq ||\boldsymbol{u}||||\boldsymbol{v}|| \\
| < \boldsymbol{u}, \boldsymbol{v} > | &\leq < \boldsymbol{u}, \boldsymbol{u} >^{1/2} < \boldsymbol{v}, \boldsymbol{v} >^{1/2} \\
< \boldsymbol{u}, \boldsymbol{v} >^2 &\leq < \boldsymbol{u}, \boldsymbol{u} >< \boldsymbol{v}, \boldsymbol{v} > \\
(\boldsymbol{v}^T A^T A\boldsymbol{u})^2 &\leq (\boldsymbol{u}^T A^T A\boldsymbol{u})(\boldsymbol{v}^T A^T A\boldsymbol{v})
\end{aligned}$$

[DF*] **Example:**
Use the Cauchy-Schwarz inequality to prove that for all real values of $a, b$, and $\theta$,

$$(a \cos\ \theta + b \sin\ \theta)^2 \leq a^2 + b^2$$

**Solution:**
From Cauchy-Schwarz inequality we know that

$$\begin{aligned}
-1 \leq \cos\ \theta \leq 1 \\
-1 \leq \sin\ \theta \leq 1
\end{aligned}$$

then we can multiply the inequality of $\cos\ \theta$ with $a$ and the inequality of $\sin\ \theta$ with $b$

$$\begin{aligned}
-a \leq a \cos\ \theta \leq a \\
-b \leq b \sin\ \theta \leq b
\end{aligned}$$

adding them then give a power of 2 it will become

$$\begin{aligned}
-(a+b) \leq (a \cos\ \theta + b \sin\ \theta) \leq a + b \\
-(a+b)^2 \leq (a \cos\ \theta + b \sin\ \theta)^2 \leq (a+b)^2
\end{aligned}$$

[DF*] **Example:**
Show that equality holds in the Cauchy-Schwarz inequality if and only if $\boldsymbol{u}$ and $\boldsymbol{v}$ are linearly dependent.
**Solution:**
If $\boldsymbol{u}$ and $\boldsymbol{v}$ are linearly dependent then we can define them with

$$\boldsymbol{u} = k\boldsymbol{v}$$

with $k$ as scalar. The Cauchy-Schwarz inequality states

$$-1 \leq \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||} \leq 1$$

Now, we will do the substitution

$$\begin{aligned}\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||} &= \frac{k\boldsymbol{v}\cdot\boldsymbol{v}}{||\boldsymbol{k}\boldsymbol{v}||||\boldsymbol{v}||}\\ &= \frac{k\boldsymbol{v}\cdot\boldsymbol{v}}{|k|||\boldsymbol{v}||||\boldsymbol{v}||}\\ &= \frac{\boldsymbol{v}\cdot\boldsymbol{v}}{||\boldsymbol{v}||^2}\\ &= \frac{<\boldsymbol{v},\boldsymbol{v}>}{||\boldsymbol{v}||^2}\\ &= \frac{||\boldsymbol{v}||^2}{||\boldsymbol{v}||^2} = 1\end{aligned}$$

it is proven then that the equality holds in Cauchy-Schwarz inequality if $\boldsymbol{u}$ and $\boldsymbol{v}$ are linearly dependent.

[DF*] **Example:**
Use vector methods to prove that a triangle that is inscribed in a circle so that it has a diameter for a side must be a right triangle.
**Solution:**
We will express the vectors $\vec{AB}$ and $\vec{BC}$ in terms of $\boldsymbol{u}$ and $\boldsymbol{v}$.

**Figure 23.141:** *The illustration for a triangle inscribed in a unit circle.*

$$\begin{aligned}\vec{AB} &= \boldsymbol{v}+\boldsymbol{u}\\ \vec{BC} &= \boldsymbol{v}-\boldsymbol{u}\end{aligned}$$

Use the Euclidean inner product on $\mathbb{R}^2$ and note that $||\boldsymbol{v}|| = ||\boldsymbol{u}||$, since it is the radius of the circle.

$$\begin{aligned}<\vec{AB},\vec{BC}> &= <\boldsymbol{v}+\boldsymbol{u},\boldsymbol{v}-\boldsymbol{u}>\\ &= <\boldsymbol{v},\boldsymbol{v}> - <\boldsymbol{v},\boldsymbol{u}> + <\boldsymbol{u},\boldsymbol{v}> - <\boldsymbol{u},\boldsymbol{u}>\\ &= ||\boldsymbol{v}||^2 - ||\boldsymbol{u}||^2\\ &= 0\end{aligned}$$

Thus $\vec{AB}$ and $\vec{BC}$ are orthogonal, so the angle at $B$ is a right angle.

[DF*] **Example:**
The vectors $\boldsymbol{u} = (1, \sqrt{3})$ and $\boldsymbol{v} = (-1, \sqrt{3})$ have norm 2 and an angle of $60^0$ between them relative to the Euclidean inner product. Find a weighted Euclidean inner product with respect to which $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal unit vectors.

**Figure 23.142:** *The illustration for two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ with norm 2 in a circle of radius 2.*

**Solution:**
We will compute the Euclidean inner product

$$\begin{aligned} < \boldsymbol{u} \cdot \boldsymbol{v} > &= (1, \sqrt{3}) \cdot (-s1, \sqrt{3}) \\ &= 1(-1) + (\sqrt{3})(\sqrt{3}) \\ &= 2 \end{aligned}$$

The Euclidean inner product can also be obtained from the formula

$$\begin{aligned} \cos\ \theta &= \frac{< \boldsymbol{u}, \boldsymbol{v} >}{||\boldsymbol{u}||||\boldsymbol{v}||} \\ \frac{1}{2} &= \frac{< \boldsymbol{u}, \boldsymbol{v} >}{2(2)} \\ < \boldsymbol{u} \cdot \boldsymbol{v} > &= 2 \end{aligned}$$

The weighted Euclidean inner product formula needs vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ as unit vector, thus

$$\begin{aligned} \boldsymbol{u} &= \frac{\boldsymbol{u}}{||\boldsymbol{u}||} = \frac{(1, \sqrt{3})}{2} = \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right) \\ \boldsymbol{v} &= \frac{\boldsymbol{v}}{||\boldsymbol{v}||} = \frac{(-1, \sqrt{3})}{2} = \left(\frac{-1}{2}, \frac{\sqrt{3}}{2}\right) \end{aligned}$$

then we will have

$$
\begin{aligned}
< \boldsymbol{u}, \boldsymbol{v} > &= w_1 u_1 v_1 + w_2 u_2 v_2 \\
0 &= w_1 \left(\frac{1}{2}\right)\left(\frac{-1}{2}\right) + w_2 \left(\frac{\sqrt{3}}{2}\right)\left(\frac{\sqrt{3}}{2}\right) \\
\frac{w_1}{4} &= \frac{3w_2}{4} \\
w_1 &= 3w_2
\end{aligned}
$$

so for any scalar $w_2 \in \mathbb{R}$, we set the other weight $w_1 = 3w_2$. It means that the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthogonal with respect to the weighted Euclidean inner product

$$< \boldsymbol{u}, \boldsymbol{v} >= 3u_1 v_1 + u_2 v_2$$

The weight is not depending on the norm of each vector $\boldsymbol{u}$ and $\boldsymbol{v}$, if we didn't do the normalization of vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ we will still end up with the same weights for the weighted Euclidean inner product formula.

[DF*] **Example:**

Let $f(x)$ and $g(x)$ be continuous functions on $[0, 1]$. Prove

(a)

$$\left[\int_0^1 f(x)g(x)\ dx\right]^2 \leq \left[\int_0^1 f^2(x)\ dx\right]\left[\int_0^1 g^2(x)\ dx\right]$$

(b)

$$\left[\int_0^1 [f(x)+g(x)]^2\ dx\right]^{1/2} \leq \left[\int_0^1 f^2(x)\ dx\right]^{1/2}\left[\int_0^1 g^2(x)\ dx\right]^{1/2}$$

**Solution:**

(a) First we will define the inner product

$$
\begin{aligned}
< \boldsymbol{f}, \boldsymbol{g} > &= \int_0^1 f(x)g(x)\ dx \\
< \boldsymbol{f}, \boldsymbol{g} > &= \int_0^1 f(x)g(x)\ dx \\
< \boldsymbol{f}, \boldsymbol{f} > &= \int_0^1 f^2(x)\ dx \\
< \boldsymbol{g}, \boldsymbol{g} > &= \int_0^1 g^2(x)\ dx
\end{aligned}
$$

then from Cauchy-Schwarz inequality we have

$$
\begin{aligned}
| < \boldsymbol{f}, \boldsymbol{g} > | &\leq ||\boldsymbol{f}||\ ||\boldsymbol{g}|| \\
| < \boldsymbol{f}, \boldsymbol{g} > | &\leq < \boldsymbol{f}, \boldsymbol{f} >^{1/2} < \boldsymbol{g}, \boldsymbol{g} >^{1/2} \\
< \boldsymbol{f}, \boldsymbol{g} >^2 &\leq < \boldsymbol{f}, \boldsymbol{f} > < \boldsymbol{g}, \boldsymbol{g} > \\
\left[\int_0^1 f(x)g(x)\ dx\right]^2 &\leq \left[\int_0^1 f^2(x)\ dx\right]\left[\int_0^1 g^2(x)\ dx\right]
\end{aligned}
$$

with $< \boldsymbol{f}, \boldsymbol{f} >$ and $< \boldsymbol{g}, \boldsymbol{g} >$ are nonnegative.

(b) First we will define the inner product

$$
\begin{aligned}
< f, g > &= \int_0^1 f(x)g(x)\ dx \\
< f, f > &= \int_0^1 f^2(x)\ dx \\
< g, g > &= \int_0^1 g^2(x)\ dx
\end{aligned}
$$

From theorem, if $f$ and $g$ are vectors in a real inner product space, then we can use the triangle inequality for vectors

$$
\begin{aligned}
||f+g||^2 &=< f+g, f+g > \\
&=< f, f > +2 < f, g > + < g, g > \\
&\leq< f, f > +2| < f, g|+ < g, g > \\
&\leq< f, f > +2||f||\ ||g||+ < g, g > \\
&\leq ||f||^2 + 2||f||\ ||g|| + ||g||^2 \\
&\leq (||f|| + ||g||)^2
\end{aligned}
$$

we use Cauchy-Schwarz inequality at the process above where $| < f, g > | \leq || < f||\ || < g||$, then taking square roots gives

$$||f+g|| \leq ||f|| + ||g||$$

substitutes the vectors

$$
\begin{aligned}
||f+g|| &\leq ||f|| + ||g|| \\
< f+g, f+g >^{1/2} &\leq< f, f >^{1/2} + < g, g >^{1/2} \\
\left[\int_0^1 [f(x)+g(x)]^2\ dx\right]^{1/2} &\leq \left[\int_0^1 f^2(x)\ dx\right]^{1/2} \left[\int_0^1 g^2(x)\ dx\right]^{1/2}
\end{aligned}
$$

[DF*] **Example:**
Let $C[0, \pi]$ have the inner product

$$< f, g >= \int_0^\pi f(x)g(x)\ dx$$

and let $f_n = \cos\ nx$ $(n = 0, 1, 2, \ldots)$. Show that if $k \neq l$, then $f_k$ and $f_l$ are orthogonal vectors.
**Solution:**
We will use the generalized theorem of pythagoras, if $f_k$ and $f_l$ are orthogonal vectors in an inner product space, then

$$
\begin{aligned}
||f_k + f_l||^2 &= ||f_k||^2 + ||f_l||^2 \\
< f_k + f_l, f_k + f_l > &=< f_k, f_k > + < f_l, f_l > \\
\int_0^\pi (\cos\ kx + \cos\ lx)^2\ dx &= \int_0^\pi \cos^2\ kx\ dx + \int_0^\pi \cos^2\ lx\ dx \\
\int_0^\pi (\cos^2 kx + 2\cos\ kx \cos\ lx + \cos^2 lx)\ dx &= \int_0^\pi \cos^2\ kx\ dx + \int_0^\pi \cos^2\ lx\ dx
\end{aligned}
$$

From the trigonometric sum rule we know that

$$\begin{aligned}\int_0^\pi 2\cos\ kx\cos\ lx\ dx &= \int_0^\pi \cos\ (k+l)x + \cos\ (k-l)x\ dx \\ &= \frac{1}{k+l}\sin(k+l)x\Big|_0^\pi + \frac{1}{k-l}\sin(k-l)x\Big|_0^\pi \\ &= 0\end{aligned}$$

because sin $n\pi x$ with $n = 0, 1, 2, \ldots$ have values of zero. Thus it completes the proof

$$\begin{aligned}\int_0^\pi (\cos^2 kx + 2\cos\ kx\cos\ lx + \cos^2 lx)\ dx &= \int_0^\pi \cos^2\ kx\ dx + \int_0^\pi \cos^2\ lx\ dx \\ \int_0^\pi (\cos^2 kx + \cos^2 lx)\ dx &= \int_0^\pi \cos^2\ kx\ dx + \int_0^\pi \cos^2\ lx\ dx\end{aligned}$$

it means that vectors $\boldsymbol{f}_k$ and $\boldsymbol{f}_l$ are orthogonal and satisfies the generalized theorem of pythagoras where $||\boldsymbol{f}_k + \boldsymbol{f}_l||^2 = ||\boldsymbol{f}_k||^2 + ||\boldsymbol{f}_l||^2$.

[DF*] **Example:**

(a) Let $W$ be the line $y = x$ in an $xy$-coordinate system in $\mathbb{R}^2$. Describe the subspace $W^\perp$.

(b) Let $W$ be the $y$-axis in an $xyz$-coordinate system in $\mathbb{R}^3$. Describe the subspace $W^\perp$.

(c) Let $W$ be the line $yz$-plane in an $xyz$-coordinate system in $\mathbb{R}^3$. Describe the subspace $W^\perp$.

**Solution:**

(a) $W^\perp$ is the line through the origin which is perpendicular to $y = x$. Thus $W^\perp$ is the line $y = -x$.

(b) $W^\perp$ is the plane through the origin which is perpendicular to the $x$-axis. Always remember if $W$ is a line then $W^\perp$ must be a plane. Thus $W^\perp$ is the $xz$-plane.

(c) $W^\perp$ is the line through the origin which is perpendicular to the $yz$-plane. Thus $W^\perp$ is the $x$-axis.

## i. C++ Computation: Evaluation Inner Product on Polynomials

Let $P_2$ have the evaluation inner product at the points

$$x_0 = -2, \quad x_1 = 0, \quad \text{and} \quad x_2 = 2$$

Compute $< \boldsymbol{p}, \boldsymbol{q} >$, $||\boldsymbol{p}||$ and $||\boldsymbol{q}||$ for the polynomials

$$\begin{aligned} \boldsymbol{p} &= p(x) = x^2 \\ \boldsymbol{q} &= q(x) = 1 + x \end{aligned}$$

**Solution:**
It follows that

$$\begin{aligned} < \boldsymbol{p}, \boldsymbol{q} > &= p(-2)q(-2) + p(0)q(0) + p(2)q(2) \\ &= 8 \end{aligned}$$

then for the norm

$$\begin{aligned} ||\boldsymbol{p}|| &= \sqrt{[p(x_0)]^2 + [p(x_1)]^2 + [p(x_2)]^2} \\ &= \sqrt{[p(-2)]^2 + [p(0)]^2 + [p(2)]^2} \\ &= 5.65685 \end{aligned}$$

$$\begin{aligned} ||\boldsymbol{q}|| &= \sqrt{[q(x_0)]^2 + [q(x_1)]^2 + [q(x_2)]^2} \\ &= \sqrt{[q(-2)]^2 + [q(0)]^2 + [q(2)]^2} \\ &= 3.31662 \end{aligned}$$

The C++ code is only using **GiNaC**.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

// Driver code
int main(int argc, char** argv)
{
       // GiNaC computation starts here for symbolic computation
       Digits = 5; // define maximum decimal digits
       symbol x("x") ;

       // Define points for the evaluation inner product
       float x0 = -2;
       float x1 = 0;
       float x2 = 2;
```

```
        // Define the polynomials
        ex p = pow(x,2) ; // polynomial p
        ex q = 1 + x ; // polynomial q
        cout << endl;

        cout << "p = " << p <<endl;
        cout << "q = " << q <<endl;
        cout << endl;

        ex inner_product = p.subs(x==x0)*q.subs(x==x0) + p.subs(x==x1)*q.
            subs(x==x1) + p.subs(x==x2)*q.subs(x==x2) ;
        ex norm_p = sqrt(pow(p.subs(x==x0),2) + pow(p.subs(x==x1),2) + pow(p
            .subs(x==x2),2) ) ;
        ex norm_q = sqrt(pow(q.subs(x==x0),2) + pow(q.subs(x==x1),2) + pow(q
            .subs(x==x2),2) ) ;

        cout << "<p,q> = " << inner_product <<endl;
        cout << "||p|| = " << norm_p <<endl;
        cout << "||q|| = " << norm_q <<endl;

        return 0;
    }
```

**C++ Code 189:** *main.cpp "Evaluation inner product on polynomials"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
q = 1+x

<p,q> = 8.0
||p|| = 5.656854
||q|| = 3.3166249
root [ ~/latex/DianFreya Mat
mbolicC++/ch23-Numerical Lin
```

**Figure 23.143:** *The computation for the evaluation inner product on polynomials (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Evaluation Inner Product on Polynomials with GiNaC/main.cpp**).*

## ii. C++ Computation: Inner Product or Definite Integral for Polynomials

Let the vector space $P_2$ have the inner product

$$< \boldsymbol{p}, \boldsymbol{q} >= \int_{-1}^{1} p(x)q(x)\ dx$$

(a) Find $||\boldsymbol{p}||$ for $\boldsymbol{p} = 1$, $\boldsymbol{p} = x$, and $\boldsymbol{p} = x^2$.

(b) Find $d(\boldsymbol{p}, \boldsymbol{q})$ if $\boldsymbol{p} = 1$ and $\boldsymbol{q} = x$.

**Solution:**

(a) First we will compute $||\boldsymbol{p}||$

$$\begin{aligned} ||\boldsymbol{p}|| &=< \boldsymbol{p}, \boldsymbol{p} >^{1/2} \\ &= \sqrt{\int_{-1}^{1} p(x)p(x)\ dx} \\ &= \sqrt{\int_{-1}^{1} p^2(x)\ dx} \end{aligned}$$

for $\boldsymbol{p} = 1$

$$||\boldsymbol{p}|| = \sqrt{2}$$

for $\boldsymbol{p} = x$

$$||\boldsymbol{p}|| = \sqrt{\frac{2}{3}}$$

for $\boldsymbol{p} = x^2$

$$||\boldsymbol{p}|| = \sqrt{\frac{2}{5}}$$

(b) To compute $d(\boldsymbol{p}, \boldsymbol{q})$ with $\boldsymbol{p} = 1$ and $\boldsymbol{q} = x$ we will use the norm and distance formula

$$\begin{aligned} d(\boldsymbol{p}, \boldsymbol{q}) &= ||\boldsymbol{p} - \boldsymbol{q}|| \\ &=< \boldsymbol{p} - \boldsymbol{q}, \boldsymbol{p} - \boldsymbol{q} >^{1/2} \\ &=< 1 - x, 1 - x >^{1/2} \\ &= \sqrt{\int_{-1}^{1} (1 - x)^2} \\ &= \sqrt{\frac{8}{3}} \end{aligned}$$

The C++ code is using **SymbolicC++** to do the symbolic integration and definite integral afterwards.

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
        Symbolic x("x"), p, p1, p2, p3, q, p1_int, p2_int, p3_int, pq_int;
        p1 = 1;
        p2 = x;
        p3 = x^(2);
        p = 1;
        q = x;

        cout << "p1 = " << p1 << endl;
        cout << "p2 = " << p2 << endl;
        cout << "p3 = " << p3 << endl;
        cout << "p = " << p << endl;
        cout << "q = " << q << endl;
        cout << endl;

        p1_int = integrate(p1^(2),x);
        p2_int = integrate(p2^(2),x);
        p3_int = integrate(p3^(2),x);
        pq_int = integrate((p-q)^(2),x);

        cout << "integral of p1 = " << p1_int << endl;
        cout << "integral of p1 from -1 to 1 = " << p1_int[x==1] - p1_int[x
            ==-1] << endl;
        cout << "|| p1 || = " << sqrt(p1_int[x==1] - p1_int[x==-1]) << endl
            ;
        cout << endl;
        cout << "integral of p2 = " << p2_int << endl;
        cout << "integral of p2 from -1 to 1 = " << p2_int[x==1] - p2_int[x
            ==-1]<< endl;
        cout << "|| p2 || = " << sqrt(p2_int[x==1] - p2_int[x==-1]) << endl
            ;
        cout << endl;
        cout << "integral of p3 = " << p3_int << endl;
        cout << "integral of p3 from -1 to 1 = " << p3_int[x==1] - p3_int[x
            ==-1]<< endl;
        cout << "|| p3 || = " << sqrt(p3_int[x==1] - p3_int[x==-1]) << endl
            ;
        cout << endl;
        cout << "d(p,q) = || p-q || = " << sqrt(pq_int[x==1] - pq_int[x
            ==-1]) << endl;

        return 0;
```

```
}
```

**C++ Code 190:** *main.cpp "Inner product or definite integral for polynomials"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

```
p1 = 1
p2 = x
p3 = x^(2)
p = 1
q = x

integral of p1 = x
integral of p1 from -1 to 1 = 2
|| p1 || = (2)^(1/2)

integral of p2 = 1/3*x^(3)
integral of p2  from -1 to 1 = 2/3
|| p2 || = (2/3)^(1/2)

integral of p3 = 1/5*x^(5)
integral of p3  from -1 to 1 = 2/5
|| p3 || = (2/5)^(1/2)

d(p,q) = || p-q || = (8/3)^(1/2)
```

**Figure 23.144:** *The computation for the inner product on polynomials in vector space $P_2$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Inner Product or Definite Integral on Polynomials with SymbolicC++/main.cpp**).*

## iii. C++ Computation: Inner Product or Definite Integral for Trigonometry and Transcendental Functions

In each part, use the inner product

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_0^1 f(x)g(x)\ dx$$

on $C[0,1]$ to compute $< \boldsymbol{f}, \boldsymbol{g} >$.

(a) $\boldsymbol{f} = \cos\ 2\pi x$, $\boldsymbol{g} = \sin\ 2\pi x$.

(b) $\boldsymbol{f} = x$, $\boldsymbol{g} = e^x$.

(c) $\boldsymbol{f} = \tan\ \frac{\pi}{4}x$, $\boldsymbol{g} = 1$.

**Solution:**

(a) We will use substitution, let $u = \sin\ 2\pi x$ and $du = 2\pi\ \cos\ 2\pi x\ dx$, then

$$\begin{aligned}
< \boldsymbol{f}, \boldsymbol{g} > &= \int_0^1 f(x)g(x)\ dx \\
&= \int_0^1 \sin\ 2\pi x\ \cos\ 2\pi x\ dx \\
&= \frac{1}{2\pi}\int_0^1 \sin\ 2\pi x\ (2\pi\ \cos\ 2\pi x\ dx) \\
&= \frac{1}{2\pi}\int_0^1 u\ du \\
&= \frac{1}{2\pi}\Big|\frac{u^2}{2}\Big|_0^1 + C \\
&= \frac{1}{2\pi}\Big|\frac{(\sin\ 2\pi x)^2}{2}\Big|_0^1 + C \\
&= C \\
&\approx 0
\end{aligned}$$

The computation for $\frac{1}{2\pi}\Big|\frac{(\sin\ 2\pi x)^2}{2}\Big|_0^1$ will result in 0 since $\sin 2\pi n$ is 0 with $n = 0, 1, 2, \ldots$. We will asume that $C$ is a negligible constant, or small enough that we can say it is near zero.

(b) We will use integration by parts, let $u = x$, $v = e^x, du = dx$, and $dv = e^x dx$.

$$\begin{aligned}
< \boldsymbol{f}, \boldsymbol{g} > &= \int_0^1 f(x)g(x)\ dx \\
&= \int_0^1 xe^x\ dx \\
&= uv - \int v\ du \\
&= xe^x - \int e^u\ du \\
&= xe^x - e^u \\
&= xe^x - e^x\Big|_0^1 \\
&= 1
\end{aligned}$$

(c) We will use substitution, let $u = \cos\ \frac{\pi}{4}x$ and $du = -\frac{\pi}{4}\ \sin\frac{\pi}{4}x\ dx$, then

$$
\begin{aligned}
< \boldsymbol{f}, \boldsymbol{g} > &= \int_0^1 f(x)g(x)\ dx \\
&= \int_0^1 \tan\ \frac{\pi}{4}x\ dx \\
&= \int_0^1 \frac{\sin\ \frac{\pi}{4}x}{\cos\ \frac{\pi}{4}x}\ dx \\
&= -\frac{4}{\pi}\int_0^1 \frac{1}{\cos\ \frac{\pi}{4}x}(-\frac{\pi}{4}\sin\ \frac{\pi}{4}x)\ dx \\
&= -\frac{4}{\pi}\int_0^1 \frac{1}{u}\ du \\
&= -\frac{4}{\pi}(-\ln|u|) + C \\
&= -\frac{4}{\pi}|-\ln\cos\ \frac{\pi}{4}x|_0^1 + C \\
&= -\frac{4}{\pi}\left(-\ln\frac{\sqrt{2}}{2}\right) - (-\ln(1)) + C
\end{aligned}
$$

The functions involved in this section are not easy to be integrated since they are transcendental function times a polynomial, or combination of trigonometric functions. If we observe, **GiNaC** can only be used to do polynomial symbolic integration, while **SymbolicC++** has its' own limit (we will show how it cannot tackle this problem). For the available solution now, what can be used is not C++, the famous library **SymPy** that can be used from **JULIA** or **Python** can easily compute all the symbolic integration fo the functions here, so we can try to call **SymPy** from C++.

**C++ Code 191:** *main.cpp "Inner product or definite integral for trigonometry and transcendental functions"*

To compile it, type:
**g++ -o main main.cpp -lcsympy**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Cambridge

- Browni

## iv. C++ Computation: Basis for an Orthogonal Complement

Let $W$ be the subspace of $\mathbb{R}^6$ spanned by the vectors

$$\begin{aligned}
\boldsymbol{w}_1 &= (1, 3, -2, 0, 2, 0) \\
\boldsymbol{w}_2 &= (2, 6, -5, -2, 4, -3) \\
\boldsymbol{w}_3 &= (0, 0, 5, 10, 0, 15) \\
\boldsymbol{w}_4 &= (2, 6, 0, 8, 4, 18)
\end{aligned}$$

Find a basis for the orthogonal complement of $W$.

**Solution:**

The space $W$ is the same as the row space of the matrix

$$A = \begin{bmatrix} 1 & 3 & -2 & 0 & 2 & 0 \\ 2 & 6 & -5 & -2 & 4 & -3 \\ 0 & 0 & 5 & 10 & 0 & 15 \\ 2 & 6 & 0 & 8 & 4 & 18 \end{bmatrix}$$

Since the row space and null space of $A$ are orthogonal complements, our problem reduces to finding a basis for the null space of the matrix $A$.

The reduced row echelon form of matrix $A$ is (verify)

$$A_{RREF} = \begin{bmatrix} 1 & 3 & 0 & 4 & 2 & 6 \\ 0 & 0 & 1 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

the number of leading 1's are three, so

$$\begin{aligned}
\text{rank}(A) &= 3 \\
\dim(A) &= 6 \\
\text{nullity}(A) &= 3
\end{aligned}$$

Remember again to find the nullity of $A$, we must find the dimension of the solution space of the linear system $A\boldsymbol{x} = \mathbf{0}$. The matrix $A$ above will have this corresponding system of equations

$$\begin{aligned}
x_1 + 3x_2 + 4x_4 + 2x_5 + 6x_6 &= 0 \\
x_3 + 2x_4 + 3x_6 &= 0 \\
x_6 &= 0
\end{aligned}$$

the last variable is already assigned as 0, thus we obtain the general solution

$$\begin{aligned}
x_1 &= -3x_2 - 4x_4 - 2x_5 - 6x_6 = -3r - 4s - 2t \\
x_2 &= r \\
x_3 &= -2x_4 - 3x_6 = -2s \\
x_4 &= s \\
x_5 &= t \\
x_6 &= 0
\end{aligned}$$

or in column vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = r \begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -4 \\ 0 \\ -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

these 3 vectors form a basis for the solution space for the null space of matrix $A$

$$\boldsymbol{v}_1 = \begin{bmatrix} -3 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} -4 \\ 0 \\ -2 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{v}_3 = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

You can check whether $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$ are orthogonal to $\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3$, and $\boldsymbol{w}_4$ by checking the dot products.

$$\begin{aligned}
\boldsymbol{v}_1 \cdot \boldsymbol{w}_1 &= 0 \\
\boldsymbol{v}_1 \cdot \boldsymbol{w}_2 &= 0 \\
\boldsymbol{v}_1 \cdot \boldsymbol{w}_3 &= 0 \\
\boldsymbol{v}_1 \cdot \boldsymbol{w}_4 &= 0 \\
\boldsymbol{v}_2 \cdot \boldsymbol{w}_1 &= 0 \\
\boldsymbol{v}_2 \cdot \boldsymbol{w}_2 &= 0 \\
\boldsymbol{v}_2 \cdot \boldsymbol{w}_3 &= 0 \\
\boldsymbol{v}_2 \cdot \boldsymbol{w}_4 &= 0 \\
\boldsymbol{v}_3 \cdot \boldsymbol{w}_1 &= 0 \\
\boldsymbol{v}_3 \cdot \boldsymbol{w}_2 &= 0 \\
\boldsymbol{v}_3 \cdot \boldsymbol{w}_3 &= 0 \\
\boldsymbol{v}_3 \cdot \boldsymbol{w}_4 &= 0
\end{aligned}$$

The first C++ code is using **SymbolicC++**, if we want the basis like above, we can just apply Gauss-Jordan elimination on $A$ with the first code. The second C++ code is using **Armadillo** that is able to compute orthonormal basis for the null space of $A$. Here are the brief explanation of codes used:

1. Gauss-Jordan Elimination with **SymbolicC++** to determine the Reduced Row Echelon Form for matrix $A$, after that we can determine the orthogonal complement for vectors $\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3$, and $\boldsymbol{w}_4$ easily by hand / manual computation. We choose to use **SymbolicC++** so if the entries are in symbolic term for another case, this code can be reused again.

2. Computing the orthonormal basis for null space for matrix $A$ with **Armadillo** and compare them with the orthogonal basis that we obtain from Reduced Row Echelon Form for matrix $A$ (manual computation).

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 4 // number of rows
#define C 6 // number of columns

// Driver program
int main()
{
       Symbolic a("a");
       Symbolic b("b");
       Symbolic c("c");

       // Construct a symbolic matrix of size 4 X 6
       Matrix<Symbolic> B_mat(4,6);
       B_mat[0][0] = 1;  B_mat[0][1] = 3; B_mat[0][2] = -2; B_mat[0][3] =
           0; B_mat[0][4] = 2; B_mat[0][5] = 0;
       B_mat[1][0] = 2;  B_mat[1][1] = 6; B_mat[1][2] = -5; B_mat[1][3] =
           -2; B_mat[1][4] = 4; B_mat[1][5] = -3;
       B_mat[2][0] = 0;  B_mat[2][1] = 0; B_mat[2][2] = 5; B_mat[2][3] =
           10; B_mat[2][4] = 0; B_mat[2][5] = 15;
       B_mat[3][0] = 2;  B_mat[3][1] = 6; B_mat[3][2] = 0; B_mat[3][3] = 8;
            B_mat[3][4] = 4; B_mat[3][5] = 18;

       cout << "The matrix A:\n" << B_mat <<endl;
       cout << endl;

       // Perform Gaussian Elimination / Forward elimination
       for (int k=0; k<R; k++)
       {
              int i_max = k;
              Symbolic v_max = B_mat[i_max][k];

              if (i_max != k)

              for (int m=0; m<C; k++)
              {
                     Symbolic temp = B_mat[k][m];
                     B_mat[k][m] = B_mat[i_max][m];
                     B_mat[i_max][m] = temp;
              }

              for (int i=k+1; i<R; i++)
```

```
        {
                // factor f to set current row kth element to 0 and
                    subsequently remaining kth column to 0
                Symbolic f = B_mat[i][k]/B_mat[k][k];

                // subtract fth multiple of corresponding kth row
                    element
                for (int j=k+1; j<C; j++)
                {
                        B_mat[i][j] -= B_mat[k][j]*f;
                        //B_mat[i][j] = B_mat[i][j]/B_mat[i][i];
                        // filling lower triangular matrix with zeros
                }
                B_mat[i][k] = 0;
        }
}

cout << "A (in reduced row form) :\n" << B_mat <<endl;

for (int i = 0; i < R; i++)
{
        Symbolic f2;
        if (B_mat[i][i] != 0)
        {
                f2 = B_mat[i][i];
        }
        else if (B_mat[i][i] == 0)
        {
                int j = 0;
                while (j < C)
                {
                        if (B_mat[i][j] !=0)
                        {
                                f2 = B_mat[i][j];
                        }
                        j++;
                }
        }

        for (int j = 0; j < C; j++)
        {

                B_mat[i][j] = B_mat[i][j]/f2;
        }
}
cout << "A (in row reduced echelon form) :\n" << B_mat <<endl;
cout<<endl;
return 0;
```

```
    }
```

**C++ Code 192:** *main.cpp "Gauss-jordan elimination to obtain basis for an orthogonal complement"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- After we do Gaussian elimination phase 1, we still need to make leading 1's in all rows, the last **for** loop in this section is better than previous code, since if the diagonal entry has value of 0, it will look for the entry in the next column that is not 0 to make it as a pivot.

```
        for (int i = 0; i < R; i++)
        {
                Symbolic f2;
                if (B_mat[i][i] != 0)
                {
                        f2 = B_mat[i][i];
                }
                else if (B_mat[i][i] == 0)
                {
                        int j = 0;
                        while (j < C)
                        {
                                if (B_mat[i][j] !=0)
                                {
                                        f2 = B_mat[i][j];
                                }
                                j++;
                        }
                }

                for (int j = 0; j < C; j++)
                {

                        B_mat[i][j] = B_mat[i][j]/f2;
                }
        }
```

```
The matrix A:
[ 1  3 -2  0  2  0]
[ 2  6 -5 -2  4 -3]
[ 0  0  5 10  0 15]
[ 2  6  0  8  4 18]


A (in reduced row form) :
[ 1  3 -2  0  2  0]
[ 0  0 -1 -2  0 -3]
[ 0  0  5 10  0 15]
[ 0  0  0  0  0  6]

A (in row reduced echelon form) :
[ 1   3   -2  0   2   0 ]
[ 0   0  1/3 2/3  0   1 ]
[ 0   0   1   2   0   3 ]
[ 0   0   0   0   0   1 ]
```

**Figure 23.145:** *The computation to obtain the reduced row echelonf form of matrix A that is underdetermined (not a square matrix) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gauss-Jordan Elimination for Underdetermined Symbolic Matrix with SymbolicC++ (July 6, 2024)/main.cpp**).*

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include "symbolicc++.h"

using namespace std;
using namespace arma;

#define R 4 // number of rows
#define C 6 // number of columns

// Driver code
int main(int argc, char** argv)
{
       arma::mat A(R,C,fill::zeros); //declare matrix A with size of R X C
           all have 0 entries with Armadillo.
       A.load("matrixA.txt"); // load matrixA.txt with Armadillo

       // Optional: Armadillo' computation
       cout << endl;
       cout << "****************Armadillo Computation of Matrix A
           ****************" << endl;
       cout << endl;

       cout <<"Rank of matrix A :" << arma::rank(A) << endl;
       cout << endl;
```

```
        cout <<"Nullity of matrix A :" << A.n_cols - arma::rank(A) << endl;
        cout << endl;

        arma::mat nullA(R,C,fill::zeros);
        nullA = arma::null(A);
        cout <<"Orthonormal basis of the null space of matrix A :\n" <<
            nullA << endl;
        cout << endl;

        arma::vec V1(C,1,fill::zeros);
        arma::vec V2(C,1,fill::zeros);
        arma::vec V3(C,1,fill::zeros);
        V1 = {-3,1,0,0,0,0} ; // extract the first column of A as vector V1
        V2 = {-4,0,-2,1,0,0} ; // extract the second column of A as vector
            V2
        V3 = {2,0,0,0,1,0} ; // extract the third column of A as vector V3

        cout <<"Normalize v1 from Armadillo : \n" << nullA.col(0) << endl;
        cout <<"Normalize v2 from Armadillo : \n" << nullA.col(1) << endl;
        cout <<"Normalize v3 from Armadillo : \n" << nullA.col(2) << endl;

        cout <<"v1 from manual computation:\n " << V1 << endl;
        cout <<"Norm of v1 : " << norm(V1) << endl;
        cout <<"Normalize v1 manual computation: \n" << V1/norm(V1)<< endl;
        cout << endl;

        cout <<"v2 from manual computation:\n " << V2 << endl;
        cout <<"Norm of v2 : " << norm(V2) << endl;
        cout <<"Normalize v2 manual computation: \n" << V2/norm(V2)<< endl;
        cout << endl;

        cout <<"v3 from manual computation:\n " << V3 << endl;
        cout <<"Norm of v3 : " << norm(V3) << endl;
        cout <<"Normalize v3 manual computation: \n" << V3/norm(V3)<< endl;
        cout << endl;

        cout <<"Orthonormal basis of the range space of matrix A :\n" <<
            arma::orth(A) << endl;
        cout << endl;
    }
```

**C++ Code 193:** *main.cpp "Orthonormal basis for null space of an underdetermined matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**

**./main**

```
****************Armadillo Computation of Matrix A****************

Rank of matrix A :3

Nullity of matrix A :3

Orthonormal basis of the null space of matrix A :
  -9.0414e-01  -7.6287e-02   3.4434e-01
  -4.9828e-02  -1.5342e-01  -6.7136e-01
  -3.4135e-01   5.2390e-01  -5.1007e-01
   1.7067e-01  -2.6195e-01   2.5503e-01
   1.8547e-01   7.9218e-01   3.2480e-01
  -9.9920e-16            0            0


Normalize v1 from Armadillo :
  -9.0414e-01
  -4.9828e-02
  -3.4135e-01
   1.7067e-01
   1.8547e-01
  -9.9920e-16

Normalize v2 from Armadillo :
  -0.0763
  -0.1534
   0.5239
  -0.2620
   0.7922
        0

Normalize v3 from Armadillo :
   0.3443
  -0.6714
  -0.5101
   0.2550
   0.3248
        0

v1 from manual computation:
   -3.0000
   1.0000
        0
        0
        0
        0

Norm of v1 : 3.16228
Normalize v1 manual computation:
  -0.9487
   0.3162
```

```
Norm of v1 : 3.16228
Normalize v1 manual computation:
  -0.9487
   0.3162
        0
        0
        0
        0


v2 from manual computation:
   -4.0000
        0
  -2.0000
   1.0000
        0
        0

Norm of v2 : 4.58258
Normalize v2 manual computation:
  -0.8729
        0
  -0.4364
   0.2182
        0
        0


v3 from manual computation:
    2.0000
        0
        0
        0
   1.0000
        0

Norm of v3 : 2.23607
Normalize v3 manual computation:
   0.8944
        0
        0
        0
   0.4472
        0


Orthonormal basis of the range space of matrix A :
  -0.0138   0.3482  -0.2914
   0.1045   0.7663  -0.4511
  -0.6608  -0.3493  -0.6584
  -0.7432   0.4118   0.5274
```

**Figure 23.146:** *The computation to obtain orthonormal basis for null space from* ***Armadillo*** *compared with orthonormal basis that we obtain from manual computation (****DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Basis for an Orthogonal Complement and Orthonormal Basis with Armadillo/main.cpp****).*

## v. Sentinel' Number

Sentinel is the name of a tree / Nature entity in Valhalla Projection (a forest) where DS Glanzsche always go there everyday to work and help Nature. Sentinel is her science tutor, so this number will be used to remember Sentinel' kindness of teaching her science.

We will talk about the world we live in, which is a 3-dimensional world / space. In this world we have the Euler' angles that are famously known as **pitch, yaw**, and **roll**. They were made from the rotations that are made by axes that are orthogonal toward each other. Sentinel' number wants to represent how many possible rotations in 2-dimension, 3-dimension, up to $n$-th dimension, for example in the 2-dimension we have the old clockwise and counterclockwise rotation, but the basic rotation is still the same between the $x$-axis and $y$-axis.

Now, suppose we will use the context of **OpenGL** coordinate system which state like this to imagine the world: You are seeing the monitor / scene and

- To your left is the negative $x$-axis.
- To your right is the positive $x$-axis.
- To the above is the positive $y$-axis.
- To the below is the negative $y$-axis.
- Further to the deep of the monitor is the negative $z$-axis.
- Going toward you is the positive $z$-axis.

Assume and imagine we have an airplane that the head is facing toward negative $z$-axis, we know that the Euler' angles will be like this:

1. **Pitch**
   The rotational movement of an airplane between two orthogonal axis, which are between $z$-axis and $y$-axis.
2. **Yaw**
   The rotational movement of an airplane between two orthogonal axis, which are between $z$-axis and $x$-axis.
3. **Roll**
   The rotational movement of an airplane between two orthogonal axis, which are between $y$-axis and $x$-axis.

These movements that make an airplane able to flies and be flexible in the air coming from rotation between two orthogonal axes in the 3-D Euclidean coordinate system. Now we want to break apart from 1-dimension:

- **1-Dimension**
  It is only a line, which has: 1 orthogonal vector that will be perpendicular to all vectors $\boldsymbol{v} \in V$ with $V$ is the vector space of 1-dimensional space.
  $$\boldsymbol{v} \perp \{\boldsymbol{0}\}$$
- **2-Dimension**
  It is a plane consists of two major axis the $x$-axis and the $y$-axis (these axes are orthogonal), to all vector $v \in V$ with $V$ is the vector space of 2-dimensional space.
  $$\boldsymbol{v} \perp \{\boldsymbol{0}\}$$
  $$\boldsymbol{v} \perp \boldsymbol{v}^{\perp}$$

- **3-Dimension**
  It is a 3-d world consists of three major axis the $x$-axis, $y$-axis and the $z$-axis (these axes are orthogonal), to all vector $v \in V$ with $V$ is the vector space of 3-dimensional space.

  $$\boldsymbol{v} \perp \{\mathbf{0}\}$$

  Suppose we take the vector $\boldsymbol{v}$ as the vector coincide or in line with positive $y$-axis, then

  $$\boldsymbol{v} \perp x - \textbf{axis}$$

  $$\boldsymbol{v} \perp z - \textbf{axis}$$

  if we take the vector $\boldsymbol{v}$ as the vector coincide or in line with positive $x$-axis, then

  $$\boldsymbol{v} \perp y - \textbf{axis}$$

  $$\boldsymbol{v} \perp z - \textbf{axis}$$

  if we take the vector $\boldsymbol{v}$ as the vector coincide or in line with positive $z$-axis, then

  $$\boldsymbol{v} \perp x - \textbf{axis}$$

  $$\boldsymbol{v} \perp y - \textbf{axis}$$

  we cannot create a tangible moving rotation between $\boldsymbol{v}$ and $\mathbf{0}$, thus for 3-d world we will only have 3 major rotations.

For generalization, we can use Sentinel' number to determine the number of possible rotations in any $\mathbb{R}^n$ or $n$-dimensional space. This number makes use of combination formula $C_r^n = \frac{n!}{r!(n-r)!}$, thus

$$S_n = \frac{(n+1)!}{2!((n+1)-2)!} \tag{23.159}$$

with $n = 1, 2, 3, 4, \ldots$.

Be aware that we are not going to use 1-dimension, since when we try to define for the 1-dimension we will have

$$\frac{1!}{2!(1-2)!} = \frac{1!}{2!(-1)!} = ?$$

We know from the Physics law that there is a normal force which is perpendicular to the object, even if it is only standing or not making any movement in a line, there is always a normal force, this could the representation for the 1-dimension here and to the $\mathbf{0}$. But, the problem with the mathematics computation is the factorial of a negative number that is undefined.

That is why, Sentinel' number starts with 2-dimension or $\mathbb{R}^2$. For the 2-dimension we will have

$$S_1 = \frac{2!}{2!(2-2)!} = 1$$

it is true, since the only possible rotation is between $x$-axis and $y$-axis, whether clockwise or counterclockwise.

Now for 3-dimension or $\mathbb{R}^3$, we will have

$$S_2 = \frac{3!}{2!(1)!} = 3$$

true, 3 rotations here are known as **pitch, yaw**, and **roll**.

For 4-dimension or $\mathbb{R}^4$, that is beyond human' senses, we will have

$$S_3 = \frac{4!}{2!(2)!} = 6$$

6 major rotations between each orthogonal axis in the $\mathbb{R}^4$.

For 5-dimension or $\mathbb{R}^5$, that is far beyond human' senses, we will have

$$S_4 = \frac{5!}{2!(3)!} = 10$$

For 6-dimension or $\mathbb{R}^6$, that is more far beyond human' senses, we will have

$$S_5 = \frac{6!}{2!(4)!} = 15$$

Now, we want to make a sequence for Sentinel number like Fibonacci number, it will be like this:

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, \ldots$$

notice that the difference between the each Sentinel' number is increasing with increment of $+1$.

$$\begin{aligned} 3 &= 1 + 2 \\ 6 &= 3 + 3 \\ 10 &= 6 + 4 \\ &\vdots \end{aligned}$$

or in more general term

$$S_m = S_{m-1} + m \tag{23.160}$$

Above is the second general formula if one wants to compute the Sentinel' number like the Golden ratio / Fibonacci number, that is, the previous Sentinel' number added with $m$, with $m = 2, 3, 4, \ldots$ and $S_1 = 1$.

The computational results will stay the same as the computation with Sentinel' number first general formula:

$$S_n = \frac{(n+1)!}{2!((n+1)-2)!}$$

with $n = 1, 2, 3, 4, \ldots$.

We can say that $S_n$ **is the number of major rotations in** $(n+1)$**-dimensional space or in** $\mathbb{R}^{n+1}$.

## vi. C++ Plot and Computation: Orthogonal Complement and Parametric Equation in $\mathbb{R}^3$

(a) Let $W$ be the plane in $\mathbb{R}^3$ with equation $2x - y - 4z = 0$. Find parametric equations for $W^\perp$.

(b) Let $W$ be the line in $\mathbb{R}^3$ with parametric equations

$$x = 2t, \quad y = -5t, \quad z = 4t$$

find an equation for $W^\perp$.

(c) Let $W$ be the intersection of the two planes

$$x - y - z = 0$$

and

$$x + 2y + z = 0$$

in $\mathbb{R}^3$. Find an equation for $W^\perp$.

**Solution:**

(a) We can represent the equation $2x - y - 4z = 0$ with matrix

$$\begin{bmatrix} 2 & -1 & -4 \end{bmatrix}$$

The corresponding system of equation is

$$2x_1 - x_2 - 4x_3 = 0$$

the general solution will be

$$\begin{aligned} x_1 &= \frac{1}{2}x_2 + 2x_3 = \frac{1}{2}r + 2s \\ x_2 &= r \\ x_3 &= s \end{aligned}$$

in column vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = r \begin{bmatrix} \frac{1}{2} \\ 1 \\ 0 \end{bmatrix} + s \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

so the vectors

$$\begin{bmatrix} \frac{1}{2} \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

are the basis that span the plane with equation $2x - y - 4z = 0$. In $\mathbb{R}^3$ if $W$ is a subspace of $\mathbb{R}^3$ with two basis vectors then $W^\perp$ will have dimension of 1, or in other words only have 1 basis vector.

The plane $2x - y - 4z = 0$ has a normal $\boldsymbol{n} = (2, -1, -4)$, this normal is perpendicular to the plane, thus this normal is in $W^\perp$, so $W^\perp$ will consist of all scalar multiples of $\boldsymbol{n}$ or $t\boldsymbol{n} = (2t, -t, -4t)$. Thus the parametric equation for $W^\perp$ is

$$x = 2t, \quad y = -t, \quad z = -4t$$

(b) Let $W$ be the line in $\mathbb{R}^3$ with parametric equations

$$x = 2t, \quad y = -5t, \quad z = 4t$$

or in vector column form

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = t \begin{bmatrix} 2 \\ -5 \\ 4 \end{bmatrix}$$

we then create it into a matrix $1 \times 3$

$$A = \begin{bmatrix} 2 & -5 & 4 \end{bmatrix}$$

we solve $A\boldsymbol{x} = \mathbf{0}$ to obtain the basis vectors that span the orthogonal complement ($W^{\perp}$). The corresponding system of equation for $A\boldsymbol{x} = \mathbf{0}$ is

$$2x_1 - 5x_2 + 4x_3 = 0$$

the general solution will be

$$\begin{aligned} x_1 &= \frac{5}{2}x_2 - 2x_3 = \frac{5}{2}r - 2s \\ x_2 &= r \\ x_3 &= s \end{aligned}$$

in column vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = r \begin{bmatrix} \frac{5}{2} \\ 1 \\ 0 \end{bmatrix} + s \begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix}$$

so the vectors

$$\begin{bmatrix} \frac{5}{2} \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix}$$

are the basis that span the plane $W^{\perp}$.

If $W$ in $\mathbb{R}^3$ is a line, then $W^{\perp}$ is a plane and will have two basis vectors that span it, the basis vector for $W$ will be just 1 and it is the normal for the plane $W^{\perp}$, this normal is a line that contains the vector $(2, -5, 4)$. Thus $W^{\perp}$ is a plane with equation

$$2x - 5y + 4z = 0$$

(c) Let $W$ be the intersection of the two planes

$$x - y - z = 0$$

and

$$x + 2y + z = 0$$

in $\mathbb{R}^3$. In order to determine the line of the intersection between two planes we have to solve the two equations

$$\begin{aligned} x - y - z &= 0 \\ x + 2y + z &= 0 \end{aligned}$$

There are two equations and three variables, so we have a free variable, let do parameterization

$$z = t$$

with $t$ as scalar, then we will have

$$\begin{aligned} x - y &= t \\ x + 2y &= -t \end{aligned}$$

Solving the equations above we will obtain the general solution

$$\begin{aligned} x &= \frac{1}{3}t \\ y &= -\frac{2}{3}t \\ z &= t \end{aligned}$$

The parametrized equation of the intersecting line between two planes is

$$(x, y, z) = \left(\frac{1}{3}, -\frac{2}{3}, 1\right) t$$

Another way to determine the line of intersection between two planes is to use cross products.

We have two planes, each plane has normal vector, and if the plane do really intersect, then both the normal vectors will meet at one point, and the line of intersection must be orthogonal to both of the normal vectors.

From the definition of cross products: A unique vector orthogonal to two linearly independent vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$ is $\boldsymbol{v}_1 \times \boldsymbol{v}_2$.

$$\boldsymbol{v}_1 \times \boldsymbol{v}_2 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}$$

The cross product resulting in the vector

$$\begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}$$

that is called the directional vector, if the directional vector is $(0, 0, 0)$ that means the two planes are parallel, but since in this case it is not $(0, 0, 0)$ the the two planes do intersect.

Now after we get the directional vector, we need to find a point on the line. We set one of the coordinates in the equations of both planes equal to zero and solve the system of equations.

We set $z = 0$ (for future case, if $z = 0$ cannot return a point on the intersecting line, you can try $x = 0$ or $y = 0$)

$$x - y = 0$$
$$x + 2y = 0$$

we will obtain $x = 0, y = 0$. The point $(0,0,0)$ is in the $(1,-2,3)t$ if we set $t = 0$. This means that the two planes and the intersecting line of the planes contain this point / the origin $(0,0,0)$. Afterwards, combining the point and the directional vector we will obtain the parametric equation for the line

$$(x,y,z) = (0,0,0) + (1,-2,3)t = (1,-2,3)t$$

or in another form

$$x(t) = t, \quad y(t) = -2t, \quad z(t) = 3t$$

Notice that we obtain $(x,y,z) = (\frac{1}{3}, -\frac{2}{3}, 1)t$ from the first method, it is actually one third of the second method solution $(x,y,z) = (1,-2,3)t$, since $t$ is a parameter that can be any real number, the solution stays the same.

The code for part (a) is using **gnuplot** with the terminal set to 'wxt' so the user can move the camera and see the basis vectors that are spanning $W$ and the orthogonal complement vectors that are spanning $W^{\perp}$. We make two options for the plotting part, the first one is to plot the basis vectors that span $W$ and the vector normal $\boldsymbol{n}$ for the plane, the second plotting option is to plot the basis vectors that span $W$ and the orthogonal complement vectors that span $W^{\perp}$, the second option is taking the data from textfiles **vector.dat** and **orthogonalvector.dat**.

```
#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;
```

```
// Point—normal equation of the plane:
// a(x—x0) + b(y—y0) + c(z—z0) = 0
// we choose the value for (x0,y0,z0) = (0,0,0) the origin
float x0 = 0;
float y0 = 0;
float z0 = 0;
// normal n = (a,b,c) = (2,—1,—4) for 2x—y—4z=0
float a = 2;
float b = —1;
float c = —4;
// to determine the vector that is passing through the origin
    (0,0,0) and (1/2,1,0)
float bx = 0.5;
float by = 1;
float bz = 0;
// to determine the vector that is passing through the origin
    (0,0,0) and (2,0,1)
float vx = 2;
float vy = 0;
float vz = 1;

// Create a vector with origin at (0,0,0) and terminal point at (x,y
    ,z) = (bx,by,bz)
pts_B_x .push_back(x0);
pts_B_y .push_back(y0);
pts_B_z .push_back(z0);
pts_B_dx.push_back(bx);
pts_B_dy.push_back(by);
pts_B_dz.push_back(bz);

// Create a vector with origin at (0,0,0) and terminal point at (x,y
    ,z) = (vx,vy,vz)
pts_C_x .push_back(x0);
pts_C_y .push_back(y0);
pts_C_z .push_back(z0);
pts_C_dx.push_back(vx);
pts_C_dy.push_back(vy);
pts_C_dz.push_back(vz);

// Create a vector with origin at (0,0,0) and terminal point at (a,b
    ,c) as the normal
pts_D_x .push_back(x0);
pts_D_y .push_back(y0);
pts_D_z .push_back(z0);
pts_D_dx.push_back(a);
pts_D_dy.push_back(b);
pts_D_dz.push_back(c);
```

```
        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt'\n"; // To make the plot interactive
        gp << "set xrange[-3:3]\n set yrange[-3:3]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 100,5,1\n"; // pitch,yaw,zoom
        gp << "rgb(r,g,b) = int(r)*65536 + int(g)*256 + int(b)\n";
        gp << "f(x,y) = (2*x-y)/4\n";
        gp << "splot '-' with vectors filled head lw 3 lc rgb 'dark-green'
            title 'v_{1} = (1/2,1,0)',\
        '-' with vectors filled head lw 3 lc rgb 'forest-green' title 'v_
            {2} = (2,0,1)',\
        '-' with vectors filled head lw 3 lc rgb 'blue' title 'n', \
        f(x,y) title 'plane (2x-y-4z=0)'\n";

        // Option 2 : plot from vector.dat and orthogonalvector.dat
        //gp << "splot f(x,y) title 'plane (2x-y-4z=0)',\
        'vector.dat' using 1:2:3:4:5:6 with vectors filled head lw 3 lc rgb
            'forest-green' title 'Vectors spanning W', \
        'orthogonalvector.dat' using 1:2:3:4:5:6 with vectors filled head lw
             3 lc rgb 'blue' title 'Vector spanning W^{perp}' \n";

        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));

        return 0;
    }
```

**C++ Code 194:** *main.cpp "Plane in 3d and the orthogonal complement vectors"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In the option 2 for plotting, we take the data from **vector.dat** and **orthogonalvector.dat** that consist of 6 columns to represent
$$x\ y\ z\ \delta x\ \delta y\ \delta z$$
with $(x, y, z)$ is the coordinate for the tail of the vector and $(\delta x, \delta y, \delta z)$ is the coordinate for head of the vector. The first row will represent the first vector, and so on.

```
gp << "splot f(x,y) title 'plane (2x-y-4z=0)',\
'vector.dat' using 1:2:3:4:5:6 with vectors filled head lw 3 lc
     rgb 'forest-green' title 'Vectors spanning W', \
'orthogonalvector.dat' using 1:2:3:4:5:6 with vectors filled
    head lw 3 lc rgb 'blue' title 'Vector spanning W^{perp}' \n
    ";
```

**Figure 23.147:** *The first option of plotting the plane with equation* $2x - y - 4z = 0$ *along with the basis vectors that span the plane and the normal vector for the plane (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Plane and Its Orthogonal Complement Vectors/main.cpp**).*

**Figure 23.148:** *The first option of plotting the plane with equation $2x - y - 4z = 0$ along with the basis vectors that span the plane and the normal vector for the plane (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Plane and Its Orthogonal Complement Vectors/main.cpp**).*

The code for part (b) is not too many different from part (a), if in part(a) we start with $W$ as a plane, this time we start with $W$ as a line to determine / compute $W^{\perp}$.

```
#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_z;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_B_dz;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_z;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_C_dz;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;

        // Point-normal equation of the plane:
        // a(x-x0) + b(y-y0) + c(z-z0) = 0
        // we choose the value for (x0,y0,z0) = (0,0,0) the origin
        float x0 = 0;
        float y0 = 0;
        float z0 = 0;
        // The line W with parametric equation x=2t, y = -5t, z=4t
        float wx = 2;
        float wy = -5;
        float wz = 4;
        // to determine the first basis vector for W^{perp} that is passing
            through the origin (0,0,0) and (2.5,1,0)
        float bx = 2.5;
        float by = 1;
        float bz = 0;
        // to determine the second basis vector for W^{perp} that is passing
             through the origin (0,0,0) and (-2,0,1)
        float vx = -2;
        float vy = 0;
        float vz = 1;
```

```
// Create a vector with origin at (0,0,0) and terminal point at (x,y
    ,z) = (bx,by,bz)
pts_B_x .push_back(x0);
pts_B_y .push_back(y0);
pts_B_z .push_back(z0);
pts_B_dx.push_back(bx);
pts_B_dy.push_back(by);
pts_B_dz.push_back(bz);

// Create a vector with origin at (0,0,0) and terminal point at (x,y
    ,z) = (vx,vy,vz)
pts_C_x .push_back(x0);
pts_C_y .push_back(y0);
pts_C_z .push_back(z0);
pts_C_dx.push_back(vx);
pts_C_dy.push_back(vy);
pts_C_dz.push_back(vz);

// Create the line W
pts_D_x .push_back(x0);
pts_D_y .push_back(y0);
pts_D_z .push_back(z0);
pts_D_dx.push_back(wx);
pts_D_dy.push_back(wy);
pts_D_dz.push_back(wz);

// Don't forget to put "\n" at the end of each line!
gp << "set terminal 'wxt'\n"; // To make the plot interactive
gp << "set xrange[-2:2]\n set yrange[-2:2]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
gp << "set view 100,5,1\n"; // pitch,yaw,zoom
gp << "rgb(r,g,b) = int(r)*65536 + int(g)*256 + int(b)\n";
gp << "fwt(x,y) = (-2*x + 5*y)/4\n";
gp << "splot '-' with vectors filled head lw 3 lc rgb 'dark-green'
    title 'v_{1} = (2.5,1,0)', \
'-' with vectors filled head lw 3 lc rgb 'forest-green' title 'v_
    {2} = (-2,0,1)', \
'-' with vectors filled head lw 3 lc rgb 'blue' title 'W', \
fwt(x,y) title 'W^{perp} Plane (2x - 5y + 4z = 0)'\n";

// Option 2 : plot from vector.dat and orthogonalvector.dat
//gp << "splot fwt(x,y) title 'W^{perp} plane (2x-5y+4z=0)',\
'vector.dat' using 1:2:3:4:5:6 with vectors filled head lw 3 lc rgb
    'forest-green' title 'Vector spanning W', \
'orthogonalvector.dat' using 1:2:3:4:5:6 with vectors filled head lw
     3 lc rgb 'blue' title 'Orthogonal vectors spanning W^{perp}' \n
    ";
```

```
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));

        return 0;
    }
```

**C++ Code 195:** *main.cpp "Line in 3d and the orthogonal complement vectors"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.149:** *The first option of plotting the line W and the plane $W^{\perp}$ for part (b) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of a Line and Its Orthogonal Complement Vectors/main.cpp**).*

The code for part (c) will be making use of the parameter plot in **gnuplot** along with some conditional statements to draw the intersecting line.

```
#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set parametric\n";
        gp << "set hidden3d\n"; // to apply the right setting for
            intersecting surfaces
        gp << "set samples 101\n";
        gp << "set isosamples 101\n";
        //gp << "set hidd front\n"; // to apply the right setting for
            intersecting surfaces
        gp << "set xrange[-5:5]\n set yrange[-5:5]\n set zrange[-5:5]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 100,5,1\n"; // pitch,yaw,zoom
        gp << "set multiplot layout 2,1 title 'Intersection of Two Planes in
             R^{3}'\n"; // multiplot
        gp << "rgb(r,g,b) = int(r)*65536 + int(g)*256 + int(b)\n";
        gp << "splot u+v,u,v title 'Plane 1 (u+v,u,v) or (x-y-z=0)' lw 2 lc
             rgb 'forest-green' ,\
        -2*u-v,u,v title 'Plane 2 (-2u-v,u,v) or (x+2y+z=0)' lw 2 lc rgb
            'sienna1'\n";
        gp << "splot (abs(u+v-(-2*u-v)) < 0.3 ? u+v : 1/0), \
        (abs(u-u) < 0.1 ? u : 1/0), \
        (abs(v-v) < 0.1 ? v : 1/0) title 'Intersecting line' lc rgb 'sea-
            green'\n";

        return 0;
}
```

**C++ Code 196:** *main.cpp "Intersecting line of two planes"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We use **multiplot** feature to create two plots at once, the first one is to plot two planes, and

the second one is to plot the intersecting line of the two planes.

The plot to create the intersecting line is using conditional statements that will need a threshold number that can be obtained from the number of the **isosamples** and the range of the **xrange, yrange, zrange**.

First we need to convert the plane equation into parametric. In **gnuplot** we use **u** and **v**, so if we have a plane equation like this:

$$x - y - z = 0$$

the plane can be plotted with parametric in **gnuplot** with this formula:

$$\textbf{splot } u + v, u, v$$

where did $(u + t, u, v)$ comes from? From setting two variables as parameters, we set

$$\begin{aligned} y &= u \\ z &= v \end{aligned}$$

then we will obtain

$$\begin{aligned} x - y - z &= 0 \\ x &= y + z \\ x &= u + v \end{aligned}$$

And for the second plane equation

$$x + 2y + z = 0$$

the plane can be plotted with parametric in **gnuplot** with this formula:

$$\textbf{splot } -2u - v, u, v$$

Then, we must set the same interval for **xrange, yrange, zrange**. Then we can get the threshold number with

$$\frac{maxrange - minrange}{\#isosamples - 1}$$

The conditional statements are to measure the distance between two planes at each corresponding entries

$$|(u + v, u, v) - (-2u - v, u, v)| = |(3u + 2v, u - u, v - v)|$$

at each corresponding entries we will see if the absolute value is below the threshold number, if the condition fulfilled the intersecting line will be plotted, otherwise ignore the current point.

```
gp << "set multiplot layout 2,1 title 'Intersection of Two
    Planes in R^{3}'\n"; // multiplot
gp << "rgb(r,g,b) = int(r)*65536 + int(g)*256 + int(b)\n";
gp << "splot u+v,u,v title 'Plane 1 (u+v,u,v) or (x-y-z=0)' lw
     2 lc rgb 'forest-green' ,\
```

```
            -2*u-v,u,v title 'Plane 2 (-2u-v,u,v) or (x+2y+z=0)' lw 2 lc
                rgb 'sienna1'\n";
            gp << "splot (abs(u+v-(-2*u-v)) < 0.3 ? u+v : 1/0), \
            (abs(u-u) < 0.1 ? u : 1/0), \
            (abs(v-v) < 0.1 ? v : 1/0) title 'Intersecting line' lc rgb '
                sea-green'\n";
```

**Figure 23.150:** *The plot of two intersecting planes with equations $x - y - z = 0$ and $x + 2y + z = 0$ (above) and the intersecting line (below) for part (c) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Two Intersecting Planes/main.cpp**).*

**Figure 23.151:** *The plot of two intersecting planes with equations $x - y - z = 0$ and $x + 2y + z = 0$ without parametric (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Two Intersecting Planes Type 2/main.cpp**).*

## X. Gram-Schmidt Process

[DF*] In many problems involving vector spaces, we are free to choose any basis for the vector space that seems appropriate. In inner product spaces, the solution of a problem is often greatly simplified by choosing a basis in which the vectors are orthogonal to one another.

**Definition 23.54: Orthogonal and Orthonormal**

A set of two or more vectors in a real inner product space is said to be orthogonal if all pairs of distinct vectors in the set are orthogonal. An orthogonal set in which each vector has norm 1 is said to be orthonormal.

[DF*] **Example:**
Let

$$\boldsymbol{u}_1 = (0,1,0), \quad \boldsymbol{u}_2 = (1,0,1), \quad \boldsymbol{u}_3 = (1,0,-1)$$

and assume that $\mathbb{R}^3$ has the Euclidean inner product. It follows that the set of vectors $S = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ is orthogonal since

$$< \boldsymbol{u}_1, \boldsymbol{u}_2 > = < \boldsymbol{u}_1, \boldsymbol{u}_3 > = < \boldsymbol{u}_2, \boldsymbol{u}_3 >$$

[DF*] If $\boldsymbol{v}$ is a nonzero vector in an inner product space, then with $k = ||\boldsymbol{v}||$

$$\left|\left|\frac{1}{||\boldsymbol{v}||}\boldsymbol{v}\right|\right| = \left|\frac{1}{||\boldsymbol{v}||}\right| ||\boldsymbol{v}|| = \frac{1}{||\boldsymbol{v}||}||\boldsymbol{v}|| = 1$$

from which we see that multiplying a nonzero vector by the reciprocal of its norm produces a vector of norm 1. This process is called normalizing $\boldsymbol{v}$.
Any orthogonal set of nonzero vectors can be converted to an orthonormal set by normalizing each of its vectors.

[DF*] The set $S = \{v_1, v_2, v_3\}$ is orthonormal if

$$< v_1, v_2 > =< v_1, v_3 >=< v_2, v_3 >= 0$$
$$||v_1|| = ||v_2|| = ||v_3|| = 1$$

[DF*] In $\mathbb{R}^2$ any two nonzero perpendicular vectors are linearly independent because neither is a scalar multiple of the other. In $\mathbb{R}^3$ any three nonzero mutually perpendicular vectors are linearly independent because no one lies in the plane of the other two.

**Theorem 23.109: Orthogonal and Linear Independent**

If $S = \{v_1, v_2, \ldots, v_n\}$ is an orthogonal set of nonzero vectors in an inner product space, then $S$ is linearly independent.

[DF*] In an inner product space, a basis consisting of orthonormal vectors is called an orthonormal basis, and a basis consisting of orthogonal vectors is called an orthogonal basis. An example of an orthonormal basis is the standard basis for $\mathbb{R}^n$ with the Euclidean inner product.

[DF*] If we want to express a vector $\boldsymbol{u}$ as a linear combination of basis vectors

$$S = \{v_1, v_2, \ldots, v_n\}$$

we need to convert the vector equation

$$\boldsymbol{u} = c_1 v_1 + c_2 v_2 + \cdots + c_n v_n$$

to a linear system and solve for the coefficients $c_1, c_2, \ldots, c_n$.

**Theorem 23.110: Coordinates Relative to Orthonormal Bases**

(a) If $S = \{v_1, v_2, \ldots, v_n\}$ is an orthogonal basis for an inner product space $V$, and if $\boldsymbol{u}$ is any vector in $V$, then

$$\boldsymbol{u} = \frac{< \boldsymbol{u}, v_1 >}{||v_1||^2} v_1 + \frac{< \boldsymbol{u}, v_2 >}{||v_2||^2} v_2 + \cdots + \frac{< \boldsymbol{u}, v_n >}{||v_n||^2} v_n \tag{23.161}$$

(b) If $S = \{v_1, v_2, \ldots, v_n\}$ is an orthonormal basis for an inner product space $V$, and if $\boldsymbol{u}$ is any vector in $V$, then

$$\boldsymbol{u} =< \boldsymbol{u}, v_1 > v_1 + < \boldsymbol{u}, v_2 > v_2 + \cdots + < \boldsymbol{u}, v_n > v_n \tag{23.162}$$

[DF*] The coordinate vector of a vector $\boldsymbol{u}$ in $V$ relative to an orthogonal basis $S = \{v_1, v_2, \ldots, v_n\}$ is

$$(\boldsymbol{u})_S = \left( \frac{< \boldsymbol{u}, v_1 >}{||v_1||^2}, \frac{< \boldsymbol{u}, v_2 >}{||v_2||^2}, \ldots, \frac{< \boldsymbol{u}, v_n >}{||v_n||^2} \right) \tag{23.163}$$

and relative to an orthonormal basis $S = \{v_1, v_2, \ldots, v_n\}$ is

$$\boldsymbol{u} =< \boldsymbol{u}, v_1 >, < \boldsymbol{u}, v_2 >, \ldots, < \boldsymbol{u}, v_n > \tag{23.164}$$

**Theorem 23.111: Projection Theorem**

If $W$ is a finite-dimensional subspace of an inner product space $V$, then every vector $\boldsymbol{u}$ in $V$ can be expressed in exactly one way as

$$\boldsymbol{u} = \boldsymbol{w}_1 + \boldsymbol{w}_2 \tag{23.165}$$

where $\boldsymbol{w}_1$ is in $W$ and $\boldsymbol{w}_2$ is in $W^{\perp}$.

[DF*] The vector $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are commonly denoted by

$$\begin{aligned} \boldsymbol{w}_1 &= \text{proj}_{\boldsymbol{w}}\boldsymbol{u} \\ \boldsymbol{w}_2 &= \text{proj}_{\boldsymbol{w}^{\perp}}\boldsymbol{u} \end{aligned} \tag{23.166}$$

They are called the orthogonal projection of $\boldsymbol{u}$ on $W$ and the orthogonal projection of $\boldsymbol{u}$ on $W^{\perp}$, respectively.

The vector $\boldsymbol{w}_2$ is also called the component of $\boldsymbol{u}$ orthogonal to $W$. The formula for $\boldsymbol{u}$ in term of orthogonal projection is

$$\boldsymbol{u} = \text{proj}_{\boldsymbol{w}}\boldsymbol{u} + \text{proj}_{\boldsymbol{w}^{\perp}}\boldsymbol{u} \tag{23.167}$$

or it can also be written as

$$\boldsymbol{u} = \text{proj}_{\boldsymbol{w}}\boldsymbol{u} + (\boldsymbol{u} - \text{proj}_{\boldsymbol{w}}\boldsymbol{u}) \tag{23.168}$$

**Theorem 23.112: Formulas for Calculating Orthogonal Projections**

Let $W$ be a finite-dimensional subspace of an inner product space $V$.

(a) If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ is an orthogonal basis for $W$, and $\boldsymbol{u}$ is any vector in $V$, then

$$\text{proj}_{\boldsymbol{w}}\boldsymbol{u} = \frac{<\boldsymbol{u}, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 + \frac{<\boldsymbol{u}, \boldsymbol{v}_2>}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 + \cdots + \frac{<\boldsymbol{u}, \boldsymbol{v}_r>}{||\boldsymbol{v}_r||^2}\boldsymbol{v}_r \tag{23.169}$$

(b) If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ is an orthonormal basis for $W$, and $\boldsymbol{u}$ is any vector in $V$, then

$$\text{proj}_{\boldsymbol{w}}\boldsymbol{u} = <\boldsymbol{u}, \boldsymbol{v}_1> \boldsymbol{v}_1 + <\boldsymbol{u}, \boldsymbol{v}_2> \boldsymbol{v}_2 + \cdots + <\boldsymbol{u}, \boldsymbol{v}_r> \boldsymbol{v}_r \tag{23.170}$$

**Theorem 23.113: Orthonormal Basis in Inner Product Space**

Every nonzero finite-dimensional inner product space has an orthonormal basis.

[DF*] Let $W$ be any nonzero finite-dimensional subspace of an inner product space, and suppose that $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$ is any basis for $W$. The following steps will produce an orthogonal basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ for $W$.

1. Let $\boldsymbol{v}_1 = \boldsymbol{u}_1$
2. We can obtain a vector $\boldsymbol{v}_2$ that is orthogonal to $\boldsymbol{v}_1$ by computing the component $\boldsymbol{u}_2$ that is orthogonal to the space $W_1$ spanned by $\boldsymbol{v}_1$.

$$\boldsymbol{v}_2 = \boldsymbol{u}_2 - \text{proj}_{W_1}\boldsymbol{u}_2 = \boldsymbol{u}_2 - \frac{<\boldsymbol{u}_2, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1$$

**Figure 23.152:** *The illustration of Gram-Schmidt process to obtain the the second orthogonal vector $\boldsymbol{v}_2$.*

3. To construct a vector $\boldsymbol{v}_3$ that is orthogonal to both $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, we compute the component of $\boldsymbol{u}_3$ orthogonal to the space $W_2$ spanned by $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$.

$$\boldsymbol{v}_3 = \boldsymbol{u}_3 - \text{proj}_{W_2}\boldsymbol{u}_3 = \boldsymbol{u}_3 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_2>}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2$$

the linear independence of $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$ ensures that $\boldsymbol{v}_3 \neq \boldsymbol{0}$. Continuing in this way

**Figure 23.153:** *The illustration of Gram-Schmidt process to obtain the third orthogonal vector $\boldsymbol{v}_3$.*

we will produce an orthogonal set of vectors after $r$ steps. Since orthogonal sets are linearly independet, this set will be an orthogonal basis for the $r$-dimensinoal space $W$.

**Definition 23.55: The Gram-Schmidt Process**

To convert a basis $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$ into an orthogonal basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$, perform the following computations:

$$\begin{aligned}
\boldsymbol{v}_1 &= \boldsymbol{u}_1 \\
\boldsymbol{v}_2 &= \boldsymbol{u}_2 - \frac{< \boldsymbol{u}_2, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 \\
\boldsymbol{v}_3 &= \boldsymbol{u}_3 - \frac{< \boldsymbol{u}_3, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{< \boldsymbol{u}_3, \boldsymbol{v}_2 >}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 \\
\boldsymbol{v}_4 &= \boldsymbol{u}_4 - \frac{< \boldsymbol{u}_4, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{< \boldsymbol{u}_4, \boldsymbol{v}_2 >}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 - \frac{< \boldsymbol{u}_4, \boldsymbol{v}_3 >}{||\boldsymbol{v}_3||^2}\boldsymbol{v}_3 \\
&\vdots \\
\boldsymbol{v}_r &= \boldsymbol{u}_r - \frac{< \boldsymbol{u}_r, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{< \boldsymbol{u}_r, \boldsymbol{v}_2 >}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 - \cdots - \frac{< \boldsymbol{u}_r, \boldsymbol{v}_{r-1} >}{||\boldsymbol{v}_{r-1}||^2}\boldsymbol{v}_{r-1}
\end{aligned} \tag{23.171}$$

To convert the orthogonal basis into an orthonormal basis $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_r\}$, normalize the orthogonal basis vectors.

**Theorem 23.114: Extending Orthonormal Sets to Orthonormal Bases**

If $W$ is a finite-dimensional inner product space, then

(a) Every orthogonal set of nonzero vectors in $W$ can be enlarged to an orthogonal basis for $W$.

(b) Every orthonormal set in $W$ can be enlarged to an orthonormal basis for $W$.

## i. C++ Computation: A Coordinate Vector Relative to an Orthonormal Basis

Let

$$\boldsymbol{v}_1 = (0,1,0), \quad \boldsymbol{v}_2 = \left(-\frac{4}{5}, 0, \frac{3}{5}\right), \quad \boldsymbol{v}_3 = \left(\frac{3}{5}, 0, \frac{4}{5}\right)$$

be an orthonormal basis for $\mathbb{R}^3$, can be notated with basis $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$. Express the vector $\boldsymbol{u} = (1,1,1)$ as a linear combination of the vectors in $S$, and find the coordinate vector $(\boldsymbol{u})_S$.

**Solution:**

First we compute the Euclidean inner product

$$\begin{aligned} <\boldsymbol{u}, \boldsymbol{v}_1> &= 1 \\ <\boldsymbol{u}, \boldsymbol{v}_2> &= -\frac{1}{5} \\ <\boldsymbol{u}, \boldsymbol{v}_3> &= \frac{7}{5} \end{aligned}$$

Therefore, we have

$$\boldsymbol{u} = \boldsymbol{v}_1 - \frac{1}{5}\boldsymbol{v}_2 + \frac{7}{5}\boldsymbol{v}_3$$

that is,

$$(1,1,1) = (0,1,0) - \frac{1}{5}\left(-\frac{4}{5}, 0, \frac{3}{5}\right) + \frac{7}{5}\left(\frac{3}{5}, 0, \frac{4}{5}\right)$$

The C++ code is using **Armadillo** and the input for the vector can also be fraction entries.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

using namespace std;
using namespace arma;

double division(double x, double y)
{
        return x/y;
}

int main() {
        int N = 3;
        arma::vec u(N,fill::zeros);
        u[0] = 1;
        u[1] = 1;
        u[2] = 1;
```

```
        arma::vec v1(N,fill::zeros);
        v1[0] = 0;
        v1[1] = division(1,1);
        v1[2] = 0;
        arma::vec v2(N,fill::zeros);
        v2[0] = division(-4,5);
        v2[1] = 0;
        v2[2] = division(3,5);
        arma::vec v3(N,fill::zeros);
        v3[0] = division(3,5);
        v3[1] = 0;
        v3[2] = division(4,5);

        cout <<"Vector v1 :" << "\n" << v1 <<endl;
        cout <<"|| v1 || :" << norm(v1) <<endl;
        cout <<"\nVector v2 :" << "\n" << v2 <<endl;
        cout <<"|| v2 || :" << norm(v2) <<endl;
        cout <<"\nVector v3 :" << "\n" << v3 <<endl;
        cout <<"|| v3 || :" << norm(v3) <<endl;
        cout <<"\nVector u :" << "\n" << u <<endl;
        cout <<"|| u || :" << norm(u) <<endl;

        double uv1 = dot(u,v1);
        double uv2 = dot(u,v2);
        double uv3 = dot(u,v3);

        cout <<"\n<u,v1> :" << uv1 <<endl;
        cout <<"<u,v2> :" << uv2 <<endl;
        cout <<"<u,v3> :" << uv3 <<endl;

        arma::vec uS(N,fill::zeros);
        uS[0] = uv1;
        uS[1] = uv2;
        uS[2] = uv3;

        cout <<"\nThe coordinate vector of u relative to S is:" <<endl;
        cout <<"(u)_{S} =\n" << uS <<endl;

        return 0;
    }
```

**C++ Code 197:** *main.cpp "Coordinate vector relative to an orthonormal basis"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**

**./main**

```
Vector v1 :
         0
    1.0000
         0

|| v1 || :1

Vector v2 :
   -0.8000
         0
    0.6000

|| v2 || :1

Vector v3 :
    0.6000
         0
    0.8000

|| v3 || :1

Vector u :
    1.0000
    1.0000
    1.0000

|| u || :1.73205

<u,v1> :1
<u,v2> :-0.2
<u,v3> :1.4

The coordinate vector of u relative to S is:
(u)_{S} =
    1.0000
   -0.2000
    1.4000
```

**Figure 23.154:** *The computation to obtain a coordinate vector* $\boldsymbol{u}$ *relative to an orthonormal basis* $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Coordinate Vector Relative to an Orthonormal Basis with Fraction Entries with Armadillo/main.cpp**).*

## ii. C++ Plot and Computation: Orthogonal Projection of vector $\boldsymbol{u}$ on plane $W$

Let $\mathbb{R}^3$ have the Euclidean inner product, and let $W$ be the subspace spanned by the orthonormal vectors $\boldsymbol{v}_1 = (0,1,0)$ and $\boldsymbol{v}_2 = \left(-\frac{4}{5}, 0, \frac{3}{5}\right)$. The orthogonal projection of $\boldsymbol{u} = (1,1,1)$ on $W$ is

$$\begin{aligned}
\text{proj}_W \boldsymbol{u} &= < \boldsymbol{u}, \boldsymbol{v}_1 > \boldsymbol{v}_1 + < \boldsymbol{u}, \boldsymbol{v}_2 > \boldsymbol{v}_2 \\
&= (1)(0,1,0) + \left(-\frac{1}{5}\right)\left(-\frac{4}{5}, 0, \frac{3}{5}\right) \\
&= \left(\frac{4}{25}, 1, -\frac{3}{25}\right)
\end{aligned}$$

The component of $\boldsymbol{u}$ orthogonal to $W$ is

$$\begin{aligned}
\text{proj}_{W^\perp} \boldsymbol{u} &= \boldsymbol{u} - \text{proj}_W \boldsymbol{u} \\
&= (1,1,1) - \left(\frac{4}{25}, 1, -\frac{3}{25}\right) \\
&= \left(\frac{21}{25}, 0, \frac{28}{25}\right)
\end{aligned}$$

We use **Armadillo** for the computation and **gnuplot** to plot the vectors and the plane $W$.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

#include "gnuplot-iostream.h"

using namespace std;
using namespace arma;

double division(double x, double y)
{
       return x/y;
}

int main() {
       int N = 3;
       arma::vec u(N,fill::zeros);
       u[0] = 1;
       u[1] = 1;
       u[2] = 1;
       arma::vec v1(N,fill::zeros);
       v1[0] = 0;
```

```
v1[1] = division(1,1);
v1[2] = 0;
arma::vec v2(N,fill::zeros);
v2[0] = division(-4,5);
v2[1] = 0;
v2[2] = division(3,5);

cout <<"Vector v1 :" << "\n" << v1 <<endl;
cout <<"|| v1 || :" << norm(v1) <<endl;
cout <<"\nVector v2 :" << "\n" << v2 <<endl;
cout <<"|| v2 || :" << norm(v2) <<endl;
cout <<"\nVector u :" << "\n" << u <<endl;
cout <<"|| u || :" << norm(u) <<endl;

double uv1 = dot(u,v1);
double uv2 = dot(u,v2);

cout <<"\n<u,v1> :" << uv1 <<endl;
cout <<"<u,v2> :" << uv2 <<endl;

arma::vec uprojW(N,fill::zeros);
arma::vec uprojWperp(N,fill::zeros);
uprojW = uv1*v1 + uv2*v2;
uprojWperp = u - uprojW;

cout <<"\nThe orthogonal projection of u on W:" <<endl;
cout <<"proj_{W} u =\n" << uprojW <<endl;
cout <<"\nThe orthogonal projection of u on W:" <<endl;
cout <<"proj_{Wperp} u =\n" << uprojWperp <<endl;

Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_z;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_B_dz;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_z;
```

```
            std::vector<double> pts_C_dx;
            std::vector<double> pts_C_dy;
            std::vector<double> pts_C_dz;
            std::vector<double> pts_D_x;
            std::vector<double> pts_D_y;
            std::vector<double> pts_D_z;
            std::vector<double> pts_D_dx;
            std::vector<double> pts_D_dy;
            std::vector<double> pts_D_dz;
            std::vector<double> pts_E_x;
            std::vector<double> pts_E_y;
            std::vector<double> pts_E_z;
            std::vector<double> pts_E_dx;
            std::vector<double> pts_E_dy;
            std::vector<double> pts_E_dz;

            float o = 0;

            // Create a vector with origin at (0,0,0) and terminal point at (u1,
                u2,u3)
            pts_A_x .push_back(o);
            pts_A_y .push_back(o);
            pts_A_z .push_back(o);
            pts_A_dx.push_back(u[0]);
            pts_A_dy.push_back(u[1]);
            pts_A_dz.push_back(u[2]);
            // Create a vector v1 that span W
            pts_B_x .push_back(o);
            pts_B_y .push_back(o);
            pts_B_z .push_back(o);
            pts_B_dx.push_back(v1[0]);
            pts_B_dy.push_back(v1[1]);
            pts_B_dz.push_back(v1[2]);
            // Create a vector v2 that span W
            pts_C_x .push_back(o);
            pts_C_y .push_back(o);
            pts_C_z .push_back(o);
            pts_C_dx.push_back(v2[0]);
            pts_C_dy.push_back(v2[1]);
            pts_C_dz.push_back(v2[2]);
            // Create an orthogonal projection of u on W
            pts_D_x .push_back(o);
            pts_D_y .push_back(o);
            pts_D_z .push_back(o);
            pts_D_dx.push_back(uprojW[0]);
            pts_D_dy.push_back(uprojW[1]);
            pts_D_dz.push_back(uprojW[2]);
            // Create the component of u orthogonal to W
```

```
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_z .push_back(o);
        pts_E_dx.push_back(uprojWperp[0]);
        pts_E_dy.push_back(uprojWperp[1]);
        pts_E_dz.push_back(uprojWperp[2]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-1:2]\nset yrange [-1:2]\nset zrange [-1:2]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "f(x,y) = (0.8*x-y)/(0.6)\n";
        // If you want to plot xy-plane (u,v,0) use this: splot '++' using
            1:2:(0) with lines
        // A plane spanned by (0,1,0) and (-4/5,0,3/5) can be written as
            (-4/5 c2, c1, 3/5 c2)
        // To plot use this: splot '++' using ($1*-0.8):2:($1*0.6)
        gp << "splot '++' using ($1*-0.8):2:($1*0.6) with lines lw 2 lc rgb
             'gray' title 'W = c_{1} v_{1} + c_{2} v_{2}', \
        '-' with vectors lw 3 lc rgb 'light-green' title 'u', '-' with
            vectors lw 3 lc rgb 'skyblue' title 'v_{1}', \
        '-' with vectors lw 3 lc rgb 'navy' title 'v_{2}', '-' with vectors
             lw 3 lc rgb 'forest-green' title 'proj_{W} u',\
        '-' with vectors lw 3 lc rgb 'plum' title 'proj_{W^{perp}} u'\n";
        gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
            pts_A_dy, pts_A_dz));
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
            pts_B_dy, pts_B_dz));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
            pts_C_dy, pts_C_dz));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
            pts_E_dy, pts_E_dz));

        return 0;
    }
```

**C++ Code 198:** *main.cpp "Orthogonal projection of vector u on plane W"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:

**make**
**./main**

Explanation for the codes:

- We are going to explain how to convert from vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ that span a plane into **gnuplot** code that is able to plot the plane $W$. First we have

$$\begin{aligned} W &= c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 \\ &= c_1(0,1,0) + c_2\left(-\frac{4}{5}, 0, \frac{3}{5}\right) \\ &= \left(-\frac{4}{5}c_2, c_1, \frac{3}{5}c_2\right) \end{aligned}$$

  with $c_1$ and $c_2$ as scalars or can be treated as parameters too. In **gnuplot** we have a special filename called '++', it is a rectangular grid of $x, y$ of the same distance of 1. The columns \$1 and \$2 refer to the $x$ and $y$ coordinates, respectively. So if you use the command :
  **splot '++' using 1:2:(sin($2))**
  it will plot a plane with equation of

$$f(x,y) = \sin(y)$$

  now if you use the command :
  **splot '++' using 1:2:(sin($1+$2))**
  it will plot a plane with equation of

$$f(x,y) = \sin(x+y)$$

  Now if we have two vectors that span a plane $W$ with parametric equation like this:

$$W = \left(-\frac{4}{5}c_2, c_1, \frac{3}{5}c_2\right)$$

  The command in **gnuplot** to plot the plane is:
  **splot '++' using ($1*-0.8):2:($1*0.6)**
  we treat columns \$1 as $c_2$, replacing $x$ coordinate, and columns \$2 as $c_1$, replacing $y$ coordinate.

  By logic, we can also use this command:
  **splot '++' using ($2*-0.8):1:($2*0.6)**
  we treat columns \$1 as $c_1$, replacing $x$ coordinate, and columns \$2 as $c_2$, replacing $y$ coordinate.

  Both commands will plot the same plane that are spanned by vectors $\boldsymbol{v}_1 = (0,1,0)$ and $\boldsymbol{v}_2 = \left(-\frac{4}{5}, 0, \frac{3}{5}\right)$.

```
gp << "splot '++' using ($1*-0.8):1:($1*0.6) with lines
    lw 2 lc rgb 'gray' title 'W = c_{1} v_{1} + c_{2} v_
    {2}', \
'-' with vectors lw 3 lc rgb 'light-green' title 'u',
    '-' with vectors lw 3 lc rgb 'skyblue' title 'v_
    {1}', \
```

```
                    '—' with vectors lw 3 lc rgb 'navy' title 'v_{2}', '—'
                        with vectors lw 3 lc rgb 'forest—green' title '
                        proj_{W} u',\
                    '—' with vectors lw 3 lc rgb 'plum' title 'proj_{W^{perp
                        }} u'\n";
```

**Figure 23.155:** *The plot of vector* $\boldsymbol{u}$*, the orthogonal projection of* $\boldsymbol{u}$ *on the plane* $W$ *spanned by* $\boldsymbol{v}_1 = (0, 1, 0)$ *and* $\boldsymbol{v}_2 = \left(-\frac{4}{5}, 0, \frac{3}{5}\right)$*, and the component of* $\boldsymbol{u}$ *orthogonal to* $W$*(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection of Vector u on W/main.cpp**).*

**Figure 23.156:** *Another view for the plot (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection of Vector u on W/main.cpp**).*

## iii. C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^2$

Let $\mathbb{R}^2$ have the Euclidean inner product. Use the Gram-Schmidt process to transform the basis

$$\boldsymbol{u}_1 = (1, -3), \quad \boldsymbol{u}_2 = (2, 2)$$

into an orthonormal basis. Draw both sets of basis vectors in the $xy$-plane.

**Solution:**

First for the step 1

$$\boldsymbol{v}_1 = \boldsymbol{u}_1 = (1, -3)$$

Then we go to step 2

$$\begin{aligned} \boldsymbol{v}_2 &= \boldsymbol{u}_2 - \text{proj}_{W_1} \boldsymbol{u}_2 \\ &\boldsymbol{u}_2 - \frac{< \boldsymbol{u}_2, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2} \boldsymbol{v}_1 \\ &= (2, 2) - \frac{-4}{10}(1, -3) \\ &= (2.4, 0.8) \end{aligned}$$

form an orthogonal basis for $\mathbb{R}^2$. The norms of these vectors are

$$\begin{aligned} ||\boldsymbol{v}_1|| &= 3.16228 \\ ||\boldsymbol{v}_2|| &= 2.52982 \end{aligned}$$

so an orthonormal basis for $\mathbb{R}^2$ is

$$\begin{aligned} \boldsymbol{q}_1 &= \frac{\boldsymbol{v}_1}{||\boldsymbol{v}_1||} = (0.3162, -0.9487) \\ \boldsymbol{q}_2 &= \frac{\boldsymbol{v}_2}{||\boldsymbol{v}_2||} = (0.9487, 0.3162) \end{aligned}$$

Both the $||\boldsymbol{q}_1||$ and $||\boldsymbol{q}_2||$ have value of 1.

The C++ code is using **Armadillo** for computation and **gnuplot** for plotting.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

#include "gnuplot−iostream.h"
```

```
using namespace std;
using namespace arma;

double division(double x, double y)
{
       return x/y;
}

int main() {
       int N = 2;
       arma::vec u1(N,fill::zeros);
       u1[0] = 1;
       u1[1] = -3;
       arma::vec u2(N,fill::zeros);
       u2[0] = 2;
       u2[1] = division(2,1);

       cout <<"Vector u1 :" << "\n" << u1 <<endl;
       cout <<"|| u1 || :" << norm(u1) <<endl;
       cout <<"\nVector u2 :" << "\n" << u2 <<endl;
       cout <<"|| u2 || :" << norm(u2) <<endl;

       arma::vec v1(N,fill::zeros);
       arma::vec v2(N,fill::zeros);
       arma::vec q1(N,fill::zeros);
       arma::vec q2(N,fill::zeros);
       arma::vec u2projW1(N,fill::zeros);

       // The Gram-Schmidt Process
       v1 = u1;

       u2projW1 = (dot(u2,v1)/(pow(norm(v1),2)))*v1;
       v2 = u2 - u2projW1;

       cout <<"\nv1 :\n" << v1 <<endl;
       cout <<"v2 :\n" << v2 <<endl;
       cout <<"\n|| v1 || :" << norm(v1) <<endl;
       cout <<"|| v2 || :" << norm(v2) <<endl;
       cout <<"\n<v1,v2> :" << dot(v1,v2) <<endl;

       q1 = v1/norm(v1);
       q2 = v2/norm(v2);

       cout <<"\nq1 :\n" << q1 <<endl;
       cout <<"q2 :\n" << q2 <<endl;
       cout <<"\n|| q1 || :" << norm(q1) <<endl;
       cout <<"|| q2 || :" << norm(q2) <<endl;
```

```
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_A_x;
        std::vector<double> pts_A_y;
        std::vector<double> pts_A_dx;
        std::vector<double> pts_A_dy;
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;

        float o = 0;

        // Create a vector u1 basis in R^2
        pts_A_x .push_back(o);
        pts_A_y .push_back(o);
        pts_A_dx.push_back(u1[0]);
        pts_A_dy.push_back(u1[1]);
        // Create a vector u2 basis in R^2
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_dx.push_back(u2[0]);
        pts_B_dy.push_back(u2[1]);
        // Create a vector v1 new orthogonal basis in R^2
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_dx.push_back(v1[0]);
        pts_C_dy.push_back(v1[1]);
        // Create a vector v2 new orthogonal basis in R^2
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
```

```
        pts_D_dx.push_back(v2[0]);
        pts_D_dy.push_back(v2[1]);
        // Create a vector q1 orthonormal basis in R^2
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(q1[0]);
        pts_E_dy.push_back(q1[1]);
        // Create a vector q2 orthonormal basis in R^2
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(q2[0]);
        pts_F_dy.push_back(q2[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-5:5]\nset yrange [-5:5]\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        gp << "set multiplot layout 2,2 title 'Gram-Schmidt Process in R
            ^{2}'\n"; // multiplot

        // To plot u1,u2
        gp << " plot '-' with vectors lw 3 lc rgb 'medium-blue' title 'u_
            {1}',\
        '-' with vectors lw 3 lc rgb 'skyblue' title 'u_{2}'\n";
        gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_dx, pts_A_dy));
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));

        // To plot v1,v2
        gp << "plot '-' with vectors lw 3 lc rgb 'forest-green' title 'v_
            {1}',\
        '-' with vectors lw 3 lc rgb 'spring-green' title 'v_{2}'\n";
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));

        // To plot q1,q2
        gp << "plot '-' with vectors lw 3 lc rgb 'salmon' title 'q_{1}', \
        '-' with vectors lw 3 lc rgb 'magenta' title 'q_{2}'\n";
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));

        return 0;
    }
```

**C++ Code 199:** *main.cpp "Gram-Schmidt process in R2"*

To compile it, type:

**g++ -o main main.cpp -lboost_iostreams-larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We use the **multiplot** feature from **gnuplot** to plot 3 $xy$-plane at once. It will plot the original basis, the orthogonal basis, then the orthonormal basis, respectively.

```
...
...

gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana
    ,10' persist\n"; // To make the plot interactive
gp << "set xrange [-5:5]\nset yrange [-5:5]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
gp << "set multiplot layout 2,2 title 'Gram-Schmidt Process in
     R^{2}'\n"; // multiplot

// To plot u1,u2
gp << " plot '-' with vectors lw 3 lc rgb 'medium-blue' title
    'u_{1}',\
'-' with vectors lw 3 lc rgb 'skyblue' title 'u_{2}'\n";
gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_dx,
    pts_A_dy));
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx,
    pts_B_dy));

// To plot v1,v2
gp << "plot '-' with vectors lw 3 lc rgb 'forest-green' title
    'v_{1}',\
'-' with vectors lw 3 lc rgb 'spring-green' title 'v_{2}'\n";
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx,
    pts_C_dy));
gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx,
    pts_D_dy));

// To plot q1,q2
gp << "plot '-' with vectors lw 3 lc rgb 'salmon' title 'q_
    {1}', \
'-' with vectors lw 3 lc rgb 'magenta' title 'q_{2}'\n";
gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx,
    pts_E_dy));
gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx,
    pts_F_dy));
```

**Figure 23.157:** *The plot of original basis, orthogonal basis, and the orthonormal basis in* $\mathbb{R}^2$ *(**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Gram-Schmidt Process/main.cpp**).*

```
Vector u1 :
   1.0000
  -3.0000

|| u1 || :3.16228

Vector u2 :
   2.0000
   2.0000

|| u2 || :2.82843

v1 :
   1.0000
  -3.0000

v2 :
   2.4000
   0.8000


|| v1 || :3.16228
|| v2 || :2.52982

<v1,v2> :-8.88178e-16

q1 :
   0.3162
  -0.9487

q2 :
   0.9487
   0.3162


|| q1 || :1
|| q2 || :1
```

**Figure 23.158:** *The computation for Gram-Schmidt process to obtain the orthogonal basis and the orthonormal basis in* $\mathbb{R}^2$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Gram-Schmidt Process/main.cpp**).*

## iv. C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^3$

Assume that the vector space $\mathbb{R}^3$ has the Euclidean inner product. Apply the Gram-Schmidt process to transform the basis vectors

$$\boldsymbol{u}_1 = (1,1,1), \quad \boldsymbol{u}_2 = (0,1,1), \quad \boldsymbol{u}_3 = (0,0,1)$$

into an orthogonal basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$, and then normalize the orthogonal basis vectors to obtain an orthonormal basis $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3\}$.

**Solution:**
First for the step 1

$$\boldsymbol{v}_1 = \boldsymbol{u}_1 = (1,1,1)$$

Then we go to step 2

$$\begin{aligned}
\boldsymbol{v}_2 &= \boldsymbol{u}_2 - \text{proj}_{W_1}\boldsymbol{u}_2 \\
&\boldsymbol{u}_2 - \frac{<\boldsymbol{u}_2, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 \\
&= (0,1,1) - \frac{2}{3}(1,1,1) \\
&= \left(-\frac{2}{3}, \frac{1}{3}, \frac{1}{3}\right)
\end{aligned}$$

And finally we go to step 3

$$\begin{aligned}
\boldsymbol{v}_3 &= \boldsymbol{u}_3 - \text{proj}_{W_2}\boldsymbol{u}_3 \\
&= \boldsymbol{u}_3 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_2>}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 \\
&= (0,0,1) - \frac{1}{3}(1,1,1) - \frac{1/3}{2/3}\left(-\frac{2}{3}, \frac{1}{3}, \frac{1}{3}\right) \\
&= \left(0, -\frac{1}{2}, \frac{1}{2}\right)
\end{aligned}$$

Thus,

$$\begin{aligned}
\boldsymbol{v}_1 &= (1,1,1) \\
\boldsymbol{v}_2 &= \left(-\frac{2}{3}, \frac{1}{3}, \frac{1}{3}\right) \\
\boldsymbol{v}_3 &= \left(0, -\frac{1}{2}, \frac{1}{2}\right)
\end{aligned}$$

form an orthogonal basis for $\mathbb{R}^3$. The norms of these vectors are

$$\begin{aligned}
||\boldsymbol{v}_1|| &= \sqrt{3} \\
||\boldsymbol{v}_2|| &= \frac{\sqrt{6}}{3} \\
||\boldsymbol{v}_3|| &= \frac{1}{\sqrt{2}}
\end{aligned}$$

so an orthonormal basis for $\mathbb{R}^3$ is

$$\boldsymbol{q}_1 = \frac{\boldsymbol{v}_1}{||\boldsymbol{v}_1||} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$
$$\boldsymbol{q}_2 = \frac{\boldsymbol{v}_2}{||\boldsymbol{v}_2||} = \left(-\frac{2}{\sqrt{6}}, \frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}\right)$$
$$\boldsymbol{q}_3 = \frac{\boldsymbol{v}_3}{||\boldsymbol{v}_3||} = \left(0, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$

the C++ code is using **Armadillo** for computation and **gnuplot** for plotting.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

#include "gnuplot-iostream.h"

using namespace std;
using namespace arma;

double division(double x, double y)
{
       return x/y;
}

int main() {
       int N = 3;
       arma::vec u1(N,fill::zeros);
       u1[0] = 1;
       u1[1] = 1;
       u1[2] = 1;
       arma::vec u2(N,fill::zeros);
       u2[0] = 0;
       u2[1] = division(1,1);
       u2[2] = 1;
       arma::vec u3(N,fill::zeros);
       u3[0] = 0;
       u3[1] = 0;
       u3[2] = division(1,1);

       cout <<"Vector u1 :" << "\n" << u1 <<endl;
       cout <<"|| u1 || :" << norm(u1) <<endl;
       cout <<"\nVector u2 :" << "\n" << u2 <<endl;
```

```
cout <<"|| u2 || :" << norm(u2) <<endl;
cout <<"\nVector u3 :" << "\n" << u3 <<endl;
cout <<"|| u3 || :" << norm(u3) <<endl;

arma::vec v1(N,fill::zeros);
arma::vec v2(N,fill::zeros);
arma::vec v3(N,fill::zeros);
arma::vec q1(N,fill::zeros);
arma::vec q2(N,fill::zeros);
arma::vec q3(N,fill::zeros);
arma::vec u2projW1(N,fill::zeros);
arma::vec u3projW2(N,fill::zeros);

v1 = u1;

u2projW1 = (dot(u2,v1)/(pow(norm(v1),2)))*v1;
v2 = u2 — u2projW1;

u3projW2 = (dot(u3,v1)/(pow(norm(v1),2)))*v1 + (dot(u3,v2)/(pow(norm
    (v2),2)))*v2;
v3 = u3 — u3projW2;

cout <<"\nv1 :\n" << v1 <<endl;
cout <<"v2 :\n" << v2 <<endl;
cout <<"v3 :\n" << v3 <<endl;
cout <<"\n|| v1 || :" << norm(v1) <<endl;
cout <<"|| v2 || :" << norm(v2) <<endl;
cout <<"|| v3 || :" << norm(v3) <<endl;
cout <<"\n<v1,v2> :" << dot(v1,v2) <<endl;
cout <<"<v2,v3> :" << dot(v2,v3) <<endl;
cout <<"<v1,v3> :" << dot(v1,v3) <<endl;

q1 = v1/norm(v1);
q2 = v2/norm(v2);
q3 = v3/norm(v3);

cout <<"\nq1 :\n" << q1 <<endl;
cout <<"q2 :\n" << q2 <<endl;
cout <<"q3 :\n" << q3 <<endl;
cout <<"\n|| q1 || :" << norm(q1) <<endl;
cout <<"|| q2 || :" << norm(q2) <<endl;
cout <<"|| q3 || :" << norm(q3) <<endl;

Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
```

```
            std::vector<double> pts_A_z;
            std::vector<double> pts_A_dx;
            std::vector<double> pts_A_dy;
            std::vector<double> pts_A_dz;
            std::vector<double> pts_B_x;
            std::vector<double> pts_B_y;
            std::vector<double> pts_B_z;
            std::vector<double> pts_B_dx;
            std::vector<double> pts_B_dy;
            std::vector<double> pts_B_dz;
            std::vector<double> pts_C_x;
            std::vector<double> pts_C_y;
            std::vector<double> pts_C_z;
            std::vector<double> pts_C_dx;
            std::vector<double> pts_C_dy;
            std::vector<double> pts_C_dz;
            std::vector<double> pts_D_x;
            std::vector<double> pts_D_y;
            std::vector<double> pts_D_z;
            std::vector<double> pts_D_dx;
            std::vector<double> pts_D_dy;
            std::vector<double> pts_D_dz;
            std::vector<double> pts_E_x;
            std::vector<double> pts_E_y;
            std::vector<double> pts_E_z;
            std::vector<double> pts_E_dx;
            std::vector<double> pts_E_dy;
            std::vector<double> pts_E_dz;
            std::vector<double> pts_F_x;
            std::vector<double> pts_F_y;
            std::vector<double> pts_F_z;
            std::vector<double> pts_F_dx;
            std::vector<double> pts_F_dy;
            std::vector<double> pts_F_dz;
            std::vector<double> pts_G_x;
            std::vector<double> pts_G_y;
            std::vector<double> pts_G_z;
            std::vector<double> pts_G_dx;
            std::vector<double> pts_G_dy;
            std::vector<double> pts_G_dz;
            std::vector<double> pts_H_x;
            std::vector<double> pts_H_y;
            std::vector<double> pts_H_z;
            std::vector<double> pts_H_dx;
            std::vector<double> pts_H_dy;
            std::vector<double> pts_H_dz;
            std::vector<double> pts_I_x;
            std::vector<double> pts_I_y;
```

```
            std::vector<double> pts_I_z;
            std::vector<double> pts_I_dx;
            std::vector<double> pts_I_dy;
            std::vector<double> pts_I_dz;

            float o = 0;

            // Create a vector u1 basis in R^3
            pts_A_x .push_back(o);
            pts_A_y .push_back(o);
            pts_A_z .push_back(o);
            pts_A_dx.push_back(u1[0]);
            pts_A_dy.push_back(u1[1]);
            pts_A_dz.push_back(u1[2]);
            // Create a vector u2 basis in R^3
            pts_B_x .push_back(o);
            pts_B_y .push_back(o);
            pts_B_z .push_back(o);
            pts_B_dx.push_back(u2[0]);
            pts_B_dy.push_back(u2[1]);
            pts_B_dz.push_back(u2[2]);
            // Create a vector u3 basis in R^3
            pts_C_x .push_back(o);
            pts_C_y .push_back(o);
            pts_C_z .push_back(o);
            pts_C_dx.push_back(u3[0]);
            pts_C_dy.push_back(u3[1]);
            pts_C_dz.push_back(u3[2]);
            // Create the first orthogonal basis for R^3: v1
            pts_D_x .push_back(o);
            pts_D_y .push_back(o);
            pts_D_z .push_back(o);
            pts_D_dx.push_back(v1[0]);
            pts_D_dy.push_back(v1[1]);
            pts_D_dz.push_back(v1[2]);
            // Create the second orthogonal basis for R^3: v2
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(v2[0]);
            pts_E_dy.push_back(v2[1]);
            pts_E_dz.push_back(v2[2]);
            // Create the third orthogonal basis for R^3: v3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(v3[0]);
            pts_F_dy.push_back(v3[1]);
```

```
pts_F_dz.push_back(v3[2]);
// Create the first orthonormal basis for R^3: q1
pts_G_x .push_back(o);
pts_G_y .push_back(o);
pts_G_z .push_back(o);
pts_G_dx.push_back(q1[0]);
pts_G_dy.push_back(q1[1]);
pts_G_dz.push_back(q1[2]);
// Create the second orthonormal basis for R^3: q2
pts_H_x .push_back(o);
pts_H_y .push_back(o);
pts_H_z .push_back(o);
pts_H_dx.push_back(q2[0]);
pts_H_dy.push_back(q2[1]);
pts_H_dz.push_back(q2[2]);
// Create the third orthonormal basis for R^3: q3
pts_I_x .push_back(o);
pts_I_y .push_back(o);
pts_I_z .push_back(o);
pts_I_dx.push_back(q3[0]);
pts_I_dy.push_back(q3[1]);
pts_I_dz.push_back(q3[2]);

// Don't forget to put "\n" at the end of each line!
gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
    persist\n"; // To make the plot interactive
gp << "set xrange [-1:2]\nset yrange [-1:2]\nset zrange [-1:2]\n";
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
    axis'\n";
gp << "f(x,y) = (0.8*x-y)/(0.6)\n";

// To plot u1,u2,u3 (comment / uncomment)
/*
gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
    'W = (x,y,0)', \
'-' with vectors lw 3 lc rgb 'medium-blue' title 'u_{1}', '-' with
     vectors lw 3 lc rgb 'skyblue' title 'u_{2}', \
'-' with vectors lw 3 lc rgb 'navy' title 'u_{3}'\n";
gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
    pts_A_dy, pts_A_dz));
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
    pts_B_dy, pts_B_dz));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
    pts_C_dy, pts_C_dz));
*/
```

```
        // To plot v1,v2,v3 (comment / uncomment)
        ///*
        gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
            'W = (x,y,0)', \
        '-' with vectors lw 3 lc rgb 'forest-green' title 'v_{1}',\
        '-' with vectors lw 3 lc rgb 'dark-green' title 'v_{2}', '-' with
            vectors lw 3 lc rgb 'spring-green' title 'v_{3}'\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
            pts_E_dy, pts_E_dz));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
            pts_F_dy, pts_F_dz));
        //*/

        // To plot q1,q2,q3 (comment / uncomment)
        /*
        gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
            'W = (x,y,0)', \
        '-' with vectors lw 3 lc rgb 'forest-green' title 'q_{1}',\
        '-' with vectors lw 3 lc rgb 'dark-green' title 'q_{2}', '-' with
            vectors lw 3 lc rgb 'spring-green' title 'q_{3}'\n";
        gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
            pts_G_dy, pts_G_dz));
        gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx,
            pts_H_dy, pts_H_dz));
        gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_z, pts_I_dx,
            pts_I_dy, pts_I_dz));
        */
        return 0;
    }
```

**C++ Code 200:** *main.cpp "Gram-Schmidt process in R3"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- If you want to see vectors $\{u_1, u_2, u_3\}$ or $\{v_1, v_2, v_3\}$ or $\{q_1, q_2, q_3\}$ you just need to add **//** before **/*** and ***/** to uncomment the block of codes that have been commented. The codes below is going to plot $\{v_1, v_2, v_3\}$.

```
        ...
        ...
```

```
// To plot v1,v2,v3 (comment / uncomment)
///*
gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray'
    title 'W = (x,y,0)', \
'-' with vectors lw 3 lc rgb 'forest-green' title 'v_{1}',\
'-' with vectors lw 3 lc rgb 'dark-green' title 'v_{2}', '-'
    with vectors lw 3 lc rgb 'spring-green' title 'v_{3}'\n";
gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx
    , pts_D_dy, pts_D_dz));
gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx
    , pts_E_dy, pts_E_dz));
gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx
    , pts_F_dy, pts_F_dz));
//*/

// To plot q1,q2,q3 (comment / uncomment)
/*
gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray'
    title 'W = (x,y,0)', \
'-' with vectors lw 3 lc rgb 'forest-green' title 'q_{1}',\
'-' with vectors lw 3 lc rgb 'dark-green' title 'q_{2}', '-'
    with vectors lw 3 lc rgb 'spring-green' title 'q_{3}'\n";
gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx
    , pts_G_dy, pts_G_dz));
gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx
    , pts_H_dy, pts_H_dz));
gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_z, pts_I_dx
    , pts_I_dy, pts_I_dz));
*/
```

**Figure 23.159:** *THe original basis for* $\mathbb{R}^3$ *in this case* $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process/main.cpp**).*

**Figure 23.160:** *The orthogonal basis after performing Gram-Schmidt process* $\{v_1, v_2, v_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process/main.cpp**).*

**Figure 23.161:** *The orthonormal basis* $\{q_1, q_2, q_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process/main.cpp**).*

## v. C++ Plot and Computation: Gram-Schmidt Process in $\mathbb{R}^3$ with Weighted Inner Product

Let $\mathbb{R}^3$ have the Euclidean inner product

$$< \boldsymbol{u}, \boldsymbol{v} >= u_1 v_1 + 2u_2 v_2 + 3u_3 v_3$$

Use the Gram-Schmidt process to transform $\boldsymbol{u}_1 = (1,1,1), \boldsymbol{u}_2 = (1,1,0), \boldsymbol{u}_3 = (1,0,0)$ into an orthonormal basis.

**Solution:**

First, transform the given basis into an orthogonal basis $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$.

Step 1:

$$\boldsymbol{v}_1 = \boldsymbol{u}_1 = (1,1,1)$$

$$\begin{aligned} ||\boldsymbol{v}_1|| &= \sqrt{< \boldsymbol{v}_1, \boldsymbol{v}_1 >} \\ &= \sqrt{1^2 + 2(1)^2 + 3(1)^2} \\ &= \sqrt{6} \end{aligned}$$

Step 2:

$$\begin{aligned} \boldsymbol{v}_2 &= \boldsymbol{u}_2 - \text{proj}_{W_1} \boldsymbol{u}_2 \\ &\boldsymbol{u}_2 - \frac{< \boldsymbol{u}_2, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2} \boldsymbol{v}_1 \\ &= (1,1,0) - \frac{1(1) + 2(1)(1) + 3(0)(1)}{6}(1,1,1) \\ &= \left(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\right) \end{aligned}$$

$$\begin{aligned} ||\boldsymbol{v}_2|| &= \sqrt{< \boldsymbol{v}_2, \boldsymbol{v}_2 >} \\ &= \sqrt{\left(\frac{1}{2}\right)^2 + 2\left(\frac{1}{2}\right)^2 + 3\left(-\frac{1}{2}\right)^2} \\ &= \frac{\sqrt{6}}{2} \end{aligned}$$

Step 3:

$$
\begin{aligned}
\boldsymbol{v}_3 &= \boldsymbol{u}_3 - \text{proj}_{W_2}\boldsymbol{u}_3 \\
&= \boldsymbol{u}_3 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_1>}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 - \frac{<\boldsymbol{u}_3, \boldsymbol{v}_2>}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 \\
&= (1,0,0) - \frac{1+0+0}{6}(1,1,1) - \frac{\frac{1}{2}+0+0}{\frac{6}{4}}\left(\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right) \\
&= \left(\frac{2}{3},-\frac{1}{3},0\right)
\end{aligned}
$$

$$
\begin{aligned}
||\boldsymbol{v}_3|| &= \sqrt{<\boldsymbol{v}_3, \boldsymbol{v}_3>} \\
&= \sqrt{\left(\frac{2}{3}\right)^2 + 2\left(-\frac{1}{3}\right)^2 + 3\,(0)^2} \\
&= \frac{\sqrt{6}}{3}
\end{aligned}
$$

Thus,

$$
\begin{aligned}
\boldsymbol{v}_1 &= (1,1,1) \\
\boldsymbol{v}_2 &= \left(\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right) \\
\boldsymbol{v}_3 &= \left(\frac{2}{3},-\frac{1}{3},0\right)
\end{aligned}
$$

The orthonormal basis is

$$
\begin{aligned}
\boldsymbol{q}_1 &= \frac{\boldsymbol{v}_1}{||\boldsymbol{v}_1||} = \frac{(1,1,1)}{\sqrt{6}} = \left(\frac{1}{\sqrt{6}},\frac{1}{\sqrt{6}},\frac{1}{\sqrt{6}}\right) \\
\boldsymbol{q}_2 &= \frac{\boldsymbol{v}_2}{||\boldsymbol{v}_2||} = \frac{\left(\frac{1}{2},\frac{1}{2},-\frac{1}{2}\right)}{\frac{\sqrt{6}}{2}} = \left(\frac{1}{\sqrt{6}},\frac{1}{\sqrt{6}},-\frac{1}{\sqrt{6}}\right) \\
\boldsymbol{q}_3 &= \frac{\boldsymbol{v}_3}{||\boldsymbol{v}_3||} = \frac{\left(\frac{2}{3},-\frac{1}{3},0\right)}{\frac{\sqrt{6}}{3}} = \left(\frac{2}{\sqrt{6}},-\frac{1}{\sqrt{6}},0\right)
\end{aligned}
$$

The C++ code is still using **Armadillo** for computation and **gnuplot** to do the plotting.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>
#include <iostream>
#include <fstream>
#include <bits/stdc++.h>
#include <armadillo>

#include "gnuplot-iostream.h"
```

```
using namespace std;
using namespace arma;

double division(double x, double y)
{
       return x/y;
}

int main() {
       int N = 3;
       arma::vec u1(N,fill::zeros);
       u1[0] = 1;
       u1[1] = 1;
       u1[2] = 1;
       arma::vec u2(N,fill::zeros);
       u2[0] = 1;
       u2[1] = division(1,1);
       u2[2] = 0;
       arma::vec u3(N,fill::zeros);
       u3[0] = 1;
       u3[1] = 0;
       u3[2] = 0;

       cout <<"Vector u1 :" << "\n" << u1 <<endl;
       cout <<"|| u1 || :" << norm(u1) <<endl;
       cout <<"\nVector u2 :" << "\n" << u2 <<endl;
       cout <<"|| u2 || :" << norm(u2) <<endl;
       cout <<"\nVector u3 :" << "\n" << u3 <<endl;
       cout <<"|| u3 || :" << norm(u3) <<endl;

       arma::vec v1(N,fill::zeros);
       arma::vec v2(N,fill::zeros);
       arma::vec v3(N,fill::zeros);
       arma::vec q1(N,fill::zeros);
       arma::vec q2(N,fill::zeros);
       arma::vec q3(N,fill::zeros);
       arma::vec u2projW1(N,fill::zeros);
       arma::vec u3projW2(N,fill::zeros);

       // The Gram—Schmidt Process
       v1 = u1;
       float w1 = 1;
       float w2 = 2;
       float w3 = 3;
       float innerproduct_u2v1 = w1*u1[0]*v1[0] + w2*u2[1]*v1[1] + w3*u2
           [2]*v1[2];
       float norm_v1 = sqrt(w1*v1[0]*v1[0] + w2*v1[1]*v1[1] + w3*v1[2]*v1
```

```
    [2]);

u2projW1 = (innerproduct_u2v1 / (pow(norm_v1,2)))*v1;
v2 = u2 — u2projW1;
float norm_v2 = sqrt(w1*v2[0]*v2[0] + w2*v2[1]*v2[1] + w3*v2[2]*v2
    [2]);

float innerproduct_u3v1 = w1*u3[0]*v1[0] + w2*u3[1]*v1[1] + w3*u3
    [2]*v1[2];
float innerproduct_u3v2 = w1*u3[0]*v2[0] + w2*u3[1]*v2[1] + w3*u3
    [2]*v2[2];

u3projW2 = (innerproduct_u3v1 / (pow(norm_v1,2)))*v1 + (
    innerproduct_u3v2 / (pow(norm_v2,2)))*v2;
v3 = u3 — u3projW2;
float norm_v3 = sqrt(w1*v3[0]*v3[0] + w2*v3[1]*v3[1] + w3*v3[2]*v3
    [2]);

cout <<"\nv1 :\n" << v1 <<endl;
cout <<"v2 :\n" << v2 <<endl;
cout <<"v3 :\n" << v3 <<endl;
cout <<"\n|| v1 || :" << norm_v1 <<endl;
cout <<"|| v2 || :" << norm_v2 <<endl;
cout <<"|| v3 || :" << norm_v3 <<endl;

float innerproduct_v1v2 = w1*v1[0]*v2[0] + w2*v1[1]*v2[1] + w3*v1
    [2]*v2[2];
float innerproduct_v1v3 = w1*v1[0]*v3[0] + w2*v1[1]*v3[1] + w3*v1
    [2]*v3[2];
float innerproduct_v2v3 = w1*v2[0]*v3[0] + w2*v2[1]*v3[1] + w3*v2
    [2]*v3[2];

cout <<"\n<v1,v2> :" << innerproduct_v1v2 <<endl;
cout <<"<v2,v3> :" << innerproduct_v2v3 <<endl;
cout <<"<v1,v3> :" << innerproduct_v1v3 <<endl;

q1 = v1/norm_v1;
q2 = v2/norm_v2;
q3 = v3/norm_v3;

cout <<"\nq1 :\n" << q1 <<endl;
cout <<"q2 :\n" << q2 <<endl;
cout <<"q3 :\n" << q3 <<endl;

float norm_q1 = sqrt(w1*q1[0]*q1[0] + w2*q1[1]*q1[1] + w3*q1[2]*q1
    [2]);
float norm_q2 = sqrt(w1*q2[0]*q2[0] + w2*q2[1]*q2[1] + w3*q2[2]*q2
    [2]);
```

```
float norm_q3 = sqrt(w1*q3[0]*q3[0] + w2*q3[1]*q3[1] + w3*q3[2]*q3
    [2]);

cout <<"\n|| q1 || :" << norm_q1 <<endl;
cout <<"|| q2 || :" << norm_q2 <<endl;
cout <<"|| q3 || :" << norm_q3 <<endl;

Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_z;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_B_dz;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_z;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_C_dz;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
```

```
        std::vector<double> pts_G_z;
        std::vector<double> pts_G_dx;
        std::vector<double> pts_G_dy;
        std::vector<double> pts_G_dz;
        std::vector<double> pts_H_x;
        std::vector<double> pts_H_y;
        std::vector<double> pts_H_z;
        std::vector<double> pts_H_dx;
        std::vector<double> pts_H_dy;
        std::vector<double> pts_H_dz;
        std::vector<double> pts_I_x;
        std::vector<double> pts_I_y;
        std::vector<double> pts_I_z;
        std::vector<double> pts_I_dx;
        std::vector<double> pts_I_dy;
        std::vector<double> pts_I_dz;

        float o = 0;

        // Create a vector u1 basis in R^3
        pts_A_x .push_back(o);
        pts_A_y .push_back(o);
        pts_A_z .push_back(o);
        pts_A_dx.push_back(u1[0]);
        pts_A_dy.push_back(u1[1]);
        pts_A_dz.push_back(u1[2]);
        // Create a vector u2 basis in R^3
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_z .push_back(o);
        pts_B_dx.push_back(u2[0]);
        pts_B_dy.push_back(u2[1]);
        pts_B_dz.push_back(u2[2]);
        // Create a vector u3 basis in R^3
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_z .push_back(o);
        pts_C_dx.push_back(u3[0]);
        pts_C_dy.push_back(u3[1]);
        pts_C_dz.push_back(u3[2]);
        // Create the first orthogonal basis for R^3: v1
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_z .push_back(o);
        pts_D_dx.push_back(v1[0]);
        pts_D_dy.push_back(v1[1]);
        pts_D_dz.push_back(v1[2]);
        // Create the second orthogonal basis for R^3: v2
```

```
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(v2[0]);
            pts_E_dy.push_back(v2[1]);
            pts_E_dz.push_back(v2[2]);
            // Create the third orthogonal basis for R^3: v3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(v3[0]);
            pts_F_dy.push_back(v3[1]);
            pts_F_dz.push_back(v3[2]);
            // Create the first orthonormal basis for R^3: q1
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(q1[0]);
            pts_G_dy.push_back(q1[1]);
            pts_G_dz.push_back(q1[2]);
            // Create the second orthonormal basis for R^3: q2
            pts_H_x .push_back(o);
            pts_H_y .push_back(o);
            pts_H_z .push_back(o);
            pts_H_dx.push_back(q2[0]);
            pts_H_dy.push_back(q2[1]);
            pts_H_dz.push_back(q2[2]);
            // Create the third orthonormal basis for R^3: q3
            pts_I_x .push_back(o);
            pts_I_y .push_back(o);
            pts_I_z .push_back(o);
            pts_I_dx.push_back(q3[0]);
            pts_I_dy.push_back(q3[1]);
            pts_I_dz.push_back(q3[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [-1:2]\nset yrange [-1:2]\nset zrange [-1:2]\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";

            // To plot u1,u2,u3 (comment / uncomment)
            /*
            gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
                'W = (x,y,0)', \
```

```
            '-' with vectors lw 3 lc rgb 'medium-blue' title 'u_{1}', '-' with
                 vectors lw 3 lc rgb 'skyblue' title 'u_{2}', \
            '-' with vectors lw 3 lc rgb 'navy' title 'u_{3}'\n";
            gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
                pts_A_dy, pts_A_dz));
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
                pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
                pts_C_dy, pts_C_dz));
            */

            // To plot v1,v2,v3 (comment / uncomment)
            ///*
            gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
                'W = (x,y,0)', \
            '-' with vectors lw 3 lc rgb 'forest-green' title 'v_{1}',\
            '-' with vectors lw 3 lc rgb 'dark-green' title 'v_{2}', '-' with
                vectors lw 3 lc rgb 'spring-green' title 'v_{3}'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            //*/

            // To plot q1,q2,q3 (comment / uncomment)
            /*
            gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray' title
                'W = (x,y,0)', \
            '-' with vectors lw 3 lc rgb 'salmon' title 'q_{1}',\
            '-' with vectors lw 3 lc rgb 'magenta' title 'q_{2}', '-' with
                vectors lw 3 lc rgb 'aquamarine' title 'q_{3}'\n";
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
            gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx,
                pts_H_dy, pts_H_dz));
            gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_z, pts_I_dx,
                pts_I_dy, pts_I_dz));
            */
            return 0;
        }
```

**C++ Code 201:** *main.cpp "Gram-Schmidt process in R3 with weighted inner product"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In this Gram-Schmidt process we are not using **norm** and **dot** functions from **Armadillo** because the inner product is not Euclidean inner product, but a weighted inner product with

$$w_1 = 1,\ w_2 = 2,\ w_3 = 3$$

Thus we define new float variables to replace the **norm** and **dot**, which are:
**float norm_v1**
**float norm_v2**
**float norm_v3**

**float innerproduct_u2v1**
**float innerproduct_u3v1**
**float innerproduct_u3v2**

**float innerproduct_v1v2**
**float innerproduct_v1v3**
**float innerproduct_v2v3**

**float norm_q1**
**float norm_q2**
**float norm_q3**

```
...
...

// The Gram—Schmidt Process
v1 = u1;
float w1 = 1;
float w2 = 2;
float w3 = 3;
float innerproduct_u2v1 = w1*u1[0]*v1[0] + w2*u2[1]*v1[1] + w3*
    u2[2]*v1[2];
float norm_v1 = sqrt(w1*v1[0]*v1[0] + w2*v1[1]*v1[1] + w3*v1
    [2]*v1[2]);

u2projW1 = (innerproduct_u2v1 / (pow(norm_v1,2)))*v1;
v2 = u2 — u2projW1;
float norm_v2 = sqrt(w1*v2[0]*v2[0] + w2*v2[1]*v2[1] + w3*v2
    [2]*v2[2]);

float innerproduct_u3v1 = w1*u3[0]*v1[0] + w2*u3[1]*v1[1] + w3*
    u3[2]*v1[2];
```

```
float innerproduct_u3v2 = w1*u3[0]*v2[0] + w2*u3[1]*v2[1] + w3*
    u3[2]*v2[2];

u3projW2 = (innerproduct_u3v1 / (pow(norm_v1,2)))*v1 + (
    innerproduct_u3v2 / (pow(norm_v2,2)))*v2;
v3 = u3 - u3projW2;
float norm_v3 = sqrt(w1*v3[0]*v3[0] + w2*v3[1]*v3[1] + w3*v3
    [2]*v3[2]);

cout <<"\nv1 :\n" << v1 <<endl;
cout <<"v2 :\n" << v2 <<endl;
cout <<"v3 :\n" << v3 <<endl;
cout <<"\n|| v1 || :" << norm_v1 <<endl;
cout <<"|| v2 || :" << norm_v2 <<endl;
cout <<"|| v3 || :" << norm_v3 <<endl;

float innerproduct_v1v2 = w1*v1[0]*v2[0] + w2*v1[1]*v2[1] + w3*
    v1[2]*v2[2];
float innerproduct_v1v3 = w1*v1[0]*v3[0] + w2*v1[1]*v3[1] + w3*
    v1[2]*v3[2];
float innerproduct_v2v3 = w1*v2[0]*v3[0] + w2*v2[1]*v3[1] + w3*
    v2[2]*v3[2];

cout <<"\n<v1,v2> :" << innerproduct_v1v2 <<endl;
cout <<"<v2,v3> :" << innerproduct_v2v3 <<endl;
cout <<"<v1,v3> :" << innerproduct_v1v3 <<endl;

q1 = v1/norm_v1;
q2 = v2/norm_v2;
q3 = v3/norm_v3;

cout <<"\nq1 :\n" << q1 <<endl;
cout <<"q2 :\n" << q2 <<endl;
cout <<"q3 :\n" << q3 <<endl;

float norm_q1 = sqrt(w1*q1[0]*q1[0] + w2*q1[1]*q1[1] + w3*q1
    [2]*q1[2]);
float norm_q2 = sqrt(w1*q2[0]*q2[0] + w2*q2[1]*q2[1] + w3*q2
    [2]*q2[2]);
float norm_q3 = sqrt(w1*q3[0]*q3[0] + w2*q3[1]*q3[1] + w3*q3
    [2]*q3[2]);

cout <<"\n|| q1 || :" << norm_q1 <<endl;
cout <<"|| q2 || :" << norm_q2 <<endl;
cout <<"|| q3 || :" << norm_q3 <<endl;

...
...
```

**Figure 23.162:** *The original basis* $\{u_1, u_2, u_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process with Weighted Inner Product/main.cpp**).*

**Figure 23.163:** *The orthogonal basis* $\{v_1, v_2, v_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process with Weighted Inner Product/main.cpp**).*

**Figure 23.164:** *The orthonormal basis* $\{q_1, q_2, q_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process with Weighted Inner Product/main.cpp**).*

```
v1 :
   1.0000
   1.0000
   1.0000

v2 :
   0.5000
   0.5000
  -0.5000

v3 :
   6.6667e-01
  -3.3333e-01
   7.7376e-09


|| v1 || :2.44949
|| v2 || :1.22474
|| v3 || :0.816497

<v1,v2> :2.18351e-07
<v2,v3> :-2.32129e-08
<v1,v3> :-3.21965e-15

q1 :
   0.4082
   0.4082
   0.4082

q2 :
   0.4082
   0.4082
  -0.4082

q3 :
   8.1650e-01
  -4.0825e-01
   9.4766e-09


|| q1 || :1
|| q2 || :1
|| q3 || :1
```

**Figure 23.165:** *The computation for Gram-Schmidt process with weighted Euclidean inner product to obtain the orthonormal basis* $\{q_1, q_2, q_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Gram-Schmidt Process with Weighted Inner Product/main.cpp**).*

## vi. C++ Plot and Computation: Gram-Schmidt Process and Legendre Polynomials

Let the vector space $P_2$ have the inner product

$$< \boldsymbol{p}, \boldsymbol{q} >= \int_{-1}^{1} p(x)q(x)\ dx$$

Apply the Gram-Schmidt process to transform the standard basis $\{1, x, x^2\}$ for $P_2$ into an orthogonal basis $\{\phi_1(x), \phi_2(x), \phi_3(x)\}$.

**Solution:**

Take $\boldsymbol{u}_1 = 1$, $\boldsymbol{u}_2 = x$, and $\boldsymbol{u}_3 = x^2$.

**Step 1**

$$\boldsymbol{v}_1 = \boldsymbol{u}_1 = 1$$

**Step 2**

We have

$$< \boldsymbol{u}_2, \boldsymbol{v}_1 > = \int_{-1}^{1} x\ dx = 0$$

so

$$\boldsymbol{v}_2 = \boldsymbol{u}_2 - \frac{< \boldsymbol{u}_2, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 = \boldsymbol{u}_2 = x$$

**Step 3**

We have

$$\begin{aligned}
< \boldsymbol{u}_3, \boldsymbol{v}_1 > &= \int_{-1}^{1} x^2\ dx = \frac{2}{3} \\
< \boldsymbol{u}_3, \boldsymbol{v}_2 > &= \int_{-1}^{1} x^3\ dx = 0 \\
||\boldsymbol{v}_1||^2 &=< \boldsymbol{v}_1, \boldsymbol{v}_1 >= \int_{-1}^{1} 1\ dx = 2 \\
||\boldsymbol{v}_2||^2 &=< \boldsymbol{v}_2, \boldsymbol{v}_2 >= \int_{-1}^{1} x^2\ dx = \frac{2}{3}
\end{aligned}$$

so

$$\boldsymbol{v}_3 = \boldsymbol{u}_3 - \frac{< \boldsymbol{u}_3, \boldsymbol{v}_1 >}{||\boldsymbol{v}_1||^2}\boldsymbol{v}_1 = \boldsymbol{u}_2 - \frac{< \boldsymbol{u}_3, \boldsymbol{v}_2 >}{||\boldsymbol{v}_2||^2}\boldsymbol{v}_2 = x^2 - \frac{1}{3}$$

The C++ code is using **GiNaC** to do the symbolic integration computation, since that library can do polynomial integration very well, and the plot is done with **gnuplot**.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include "gnuplot-iostream.h"

using namespace std;
using namespace GiNaC;

// Driver code
int main(int argc, char** argv)
{
       // GiNaC computation starts here for symbolic computation
       Digits = 5; // define maximum decimal digits
       symbol x("x") ;

       // Define the upper and lower bound for the integral
       float x0 = -1;
       float x1 = 1;
       // Define the polynomials
       ex u1 = 1;
       ex u2 = x ;
       ex u3 = pow(x,2) ;
       cout << endl;

       cout << "u1 = " << u1 <<endl;
       cout << "u2 = " << u2 <<endl;
       cout << "u3 = " << u3 <<endl;
       cout << endl;

       ex v1 = u1;
       ex u2v1 = integral(x, x0,x1, u2*v1) ;
       ex v1_normpow2 = integral(x, x0,x1, pow(v1,2)) ;

       ex v2 = (u2 - (u2v1/v1_normpow2)*v1).evalf();
       ex v2_normpow2 =integral(x, x0,x1, pow(v2,2)) ;

       ex u3v1 = integral(x, x0,x1, u3*v1) ;
       ex u3v2 = integral(x, x0,x1, u3*v2) ;
       ex v3 = u3 - (u3v1.evalf()/v1_normpow2.evalf())*v1 - (u3v2.evalf()/
           v2_normpow2.evalf())*v2;

       cout << "<u2,v1> = " << u2v1.evalf() << endl;
       cout << "<u3,v1> = " << u3v1.evalf() << endl;
       cout << "<u3,v2> = " << u3v2.evalf() << endl;
       cout << "\n||v1||^{2} = " << v1_normpow2.evalf() << endl;
       cout << "||v2||^{2} = " << v2_normpow2.evalf() << endl;
```

```
        cout << "\nv1 = " << v1 <<endl;
        cout << "v2 = " << v2 <<endl;
        cout << "v3 = " << v3 <<endl;

        Gnuplot gp;
        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,1000 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-1.5:3]\nset yrange [-1.5:6]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f1(x) = 1\n";
        gp << "f2(x) = x\n";
        gp << "f3(x) = x*x\n";
        gp << "g1(x) = 1\n";
        gp << "g2(x) = x\n";
        gp << "g3(x) = x*x - 0.333\n";
        gp << "set multiplot layout 2,1 title 'Gram-Schmidt Process and
            Legendre Polynomials'\n"; // multiplot
        gp << "plot f1(x) title 'u_{1} = 1' lw 2 lc rgb 'dark-turquoise',
            f2(x) title 'u_{2} = x' lw 2 lc rgb 'forest-green',\
        f3(x) title 'u_{3} = x^{2}' lw 2 lc rgb 'spring-green'\n";
        gp << "plot g1(x) title 'v_{1} = 1' lw 2 lc rgb 'green', g2(x) title
             'v_{2} = x' lw 2 lc rgb 'forest-green',\
        g3(x) title 'v_{3} = x^{2} - 1/3' lw 2 lc rgb 'spring-green'\n";

        return 0;
    }
```

**C++ Code 202:** *main.cpp "Legendre polynomials"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We do the computation for $v_2, v_2$, and $v_3$ by adding **.evalf()** to compute the definite integral so we can get the result directly.

```
        ...
        ...

        ex v1 = u1;
```

```
ex u2v1 = integral(x, x0,x1, u2*v1) ;
ex v1_normpow2 = integral(x, x0,x1, pow(v1,2)) ;

ex v2 = (u2 — (u2v1/v1_normpow2)*v1).evalf();
ex v2_normpow2 =integral(x, x0,x1, pow(v2,2)) ;

ex u3v1 = integral(x, x0,x1, u3*v1) ;
ex u3v2 = integral(x, x0,x1, u3*v2) ;
ex v3 = u3 — (u3v1.evalf()/v1_normpow2.evalf())*v1 — (u3v2.
    evalf()/v2_normpow2.evalf())*v2;

cout << "<u2,v1> = " << u2v1.evalf() << endl;
cout << "<u3,v1> = " << u3v1.evalf() << endl;
cout << "<u3,v2> = " << u3v2.evalf() << endl;
cout << "\n||v1||^{2} = " << v1_normpow2.evalf() << endl;
cout << "||v2||^{2} = " << v2_normpow2.evalf() << endl;

...
...
```

```
u1 = 1
u2 = x
u3 = x^2

<u2,v1> = 0.0
<u3,v1> = 0.6666667
<u3,v2> = 0.0

||v1||^{2} = 2.0
||v2||^{2} = 0.6666667

v1 = 1
v2 = x
v3 = -0.33333334+x^2
```

**Figure 23.166:** *The computation to obtain the orthogonal basis* $\{\phi_1, \phi_2, \phi_3\}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Gram-Schmidt Process and Legendre Polynomials/main.cpp**).*

**Figure 23.167:** *The plot of the standard basis $\{1, x, x^2\}$ for $P_2$ and the orthogonal basis $\{\phi_1, \phi_2, \phi_3\}$ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Gram-Schmidt Process and Legendre Polynomials/main.cpp**).*

# XI. $QR$-Decomposition

[DF*] The $QR$-decomposition / factorization is the thread that connects most of the algorithms of numerical linear algebra [15], including methods for least squares, eigenvalue, and singular value problems, as well as iterative methods for all of these and also for systems of equations.

[DF*] QR-decomposition is very important in the field of numerical algorithms, including those for computing eigenvalues of large matrices.

[DF*] If $A$ is an $m \times n$ matrix with linearly independent column vectors, and if $Q$ is the matrix that results by applying Gram-Schmidt process to the column vectors of $A$, what relationship, if any exist between $A$ and $Q$?

To solve the problem above, suppose that the column vectors of $A$ are $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n$ and the orthonormal column vectors of $Q$ are $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$. Thus, $A$ and $Q$ can be written in partitioned form as

$$A = [\boldsymbol{u}_1 \mid \boldsymbol{u}_2 \mid \ldots \mid \boldsymbol{u}_n]$$

and

$$Q = [\boldsymbol{q}_1 \mid \boldsymbol{q}_2 \mid \ldots \mid \boldsymbol{q}_n]$$

it follows that $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n$ are expressible in terms of the vectors $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$ as

$$\begin{aligned}
\boldsymbol{u}_1 &= <\boldsymbol{u}_1, \boldsymbol{q}_1> \boldsymbol{q}_1 + <\boldsymbol{u}_1, \boldsymbol{q}_2> \boldsymbol{q}_2 + \cdots + <\boldsymbol{u}_1, \boldsymbol{q}_n> \boldsymbol{q}_n \\
\boldsymbol{u}_2 &= <\boldsymbol{u}_2, \boldsymbol{q}_1> \boldsymbol{q}_1 + <\boldsymbol{u}_2, \boldsymbol{q}_2> \boldsymbol{q}_2 + \cdots + <\boldsymbol{u}_2, \boldsymbol{q}_n> \boldsymbol{q}_n \\
&\vdots \\
\boldsymbol{u}_n &= <\boldsymbol{u}_n, \boldsymbol{q}_1> \boldsymbol{q}_1 + <\boldsymbol{u}_n, \boldsymbol{q}_2> \boldsymbol{q}_2 + \cdots + <\boldsymbol{u}_n, \boldsymbol{q}_n> \boldsymbol{q}_n
\end{aligned}$$

Recall that the $j$-th column vector of a matrix product is a linear combination of the column vectors of the first factor with coefficients coming from the $j$-th column of the second factor, it follows that these relationships can be expressed in matrix form as

$$[\boldsymbol{u}_1 \mid \boldsymbol{u}_2 \mid \ldots \mid \boldsymbol{u}_n] = [\boldsymbol{q}_1 \mid \boldsymbol{q}_2 \mid \ldots \mid \boldsymbol{q}_n] \begin{bmatrix} <\boldsymbol{u}_1, \boldsymbol{q}_1> & <\boldsymbol{u}_2, \boldsymbol{q}_1> & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_1> \\ <\boldsymbol{u}_1, \boldsymbol{q}_2> & <\boldsymbol{u}_2, \boldsymbol{q}_2> & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_2> \\ \vdots & \vdots & & \vdots \\ <\boldsymbol{u}_1, \boldsymbol{q}_n> & <\boldsymbol{u}_2, \boldsymbol{q}_n> & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_n> \end{bmatrix}$$

or more briefly as

$$A = QR \tag{23.172}$$

where $R$ is the second factor in the product. This equation tells that it is a factorization of $A$ into the product of a matrix $Q$ with orthonormal column vectors and an invertible upper triangular matrix $R$. However, it is a property of the Gram-Schmidt process that for $j \geq 2$, the vector $\boldsymbol{q}_j$ is orthogonal to $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{j-1}$. Thus, all entries below the main diagonal of $R$ are zero, and $R$ has the form

$$\begin{bmatrix} <\boldsymbol{u}_1, \boldsymbol{q}_1> & <\boldsymbol{u}_2, \boldsymbol{q}_1> & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_1> \\ 0 & <\boldsymbol{u}_2, \boldsymbol{q}_2> & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_2> \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & <\boldsymbol{u}_n, \boldsymbol{q}_n> \end{bmatrix} \tag{23.173}$$

$R$ is invertible and the diagonal entries are nonzero.

**Theorem 23.115: $QR$-Decomposition**

If $A$ is an $m \times n$ matrix with linearly independent column vectors, then $A$ can be factored as

$$A = QR$$

where $Q$ is an $m \times n$ matrix with orthonormal column vectors, and $R$ is an $n \times n$ invertible upper triangular matrix.

## i. C++ Computation: $QR$-Decomposition

Find the QR-decomposition of

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

**Solution:**

The column vectros of $A$ are

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Applying the Gram-Schmidt process with normalization to these column vectors yields the orthonormal vectors

$$\boldsymbol{q}_1 = \begin{bmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}, \quad \boldsymbol{q}_2 = \begin{bmatrix} -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix}, \quad \boldsymbol{q}_3 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Thus

$$R = \begin{bmatrix} < \boldsymbol{u}_1, \boldsymbol{q}_1 > & < \boldsymbol{u}_2, \boldsymbol{q}_1 > & < \boldsymbol{u}_3, \boldsymbol{q}_1 > \\ 0 & < \boldsymbol{u}_2, \boldsymbol{q}_2 > & < \boldsymbol{u}_3, \boldsymbol{q}_2 > \\ 0 & 0 & < \boldsymbol{u}_3, \boldsymbol{q}_3 > \end{bmatrix} = \begin{bmatrix} \frac{3}{\sqrt{3}} & \frac{2}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

It follows that the $QR$-decomposition of $A$ is

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{3}} & -\frac{2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{3}} & \frac{2}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

The C++ code depends only with **Armadillo** library to do the $QR$-decomposition, we even compare the result if we compute it manually, or automatically by calling a function **qr(Q,R,A)** to get the matrices $Q$ and $R$.

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;
```

```
int main(int argc, char** argv)
{
        mat A;
        int n = 3;
        A.load("MatrixA.txt");

        cout << "Matrix A:" << endl;
        A.print();
        cout << endl;

        // determinant
        cout << "det(A): " << det(A) << endl;

        cout <<"QR-decomposition with manual computation :\n"<<endl;

        arma::vec u1(n,fill::zeros);
        u1[0] = A.col(0)[0];
        u1[1] = A.col(0)[1];
        u1[2] = A.col(0)[2];
        arma::vec u2(n,fill::zeros);
        u2[0] = A.col(1)[0];
        u2[1] = A.col(1)[1];
        u2[2] = A.col(1)[2];
        arma::vec u3(n,fill::zeros);
        u3[0] = A.col(2)[0];
        u3[1] = A.col(2)[1];
        u3[2] = A.col(2)[2];

        cout <<"Vector u1 :" << "\n" << u1 <<endl;
        cout <<"|| u1 || :" << norm(u1) <<endl;
        cout <<"\nVector u2 :" << "\n" << u2 <<endl;
        cout <<"|| u2 || :" << norm(u2) <<endl;
        cout <<"\nVector u3 :" << "\n" << u3 <<endl;
        cout <<"|| u3 || :" << norm(u3) <<endl;

        arma::vec v1(n,fill::zeros);
        arma::vec v2(n,fill::zeros);
        arma::vec v3(n,fill::zeros);
        arma::vec q1(n,fill::zeros);
        arma::vec q2(n,fill::zeros);
        arma::vec q3(n,fill::zeros);
        arma::vec u2projW1(n,fill::zeros);
        arma::vec u3projW2(n,fill::zeros);
        arma::mat Q(n,n,fill::zeros);

        // The Gram-Schmidt Process
        v1 = u1;
```

```
u2projW1 = (dot(u2,v1)/(pow(norm(v1),2)))*v1;
v2 = u2 - u2projW1;

u3projW2 = (dot(u3,v1)/(pow(norm(v1),2)))*v1 + (dot(u3,v2)/(pow(norm
    (v2),2)))*v2;
v3 = u3 - u3projW2;

q1 = v1/norm(v1);
q2 = v2/norm(v2);
q3 = v3/norm(v3);

cout <<"\n q1 :\n" << q1 <<endl;
cout <<"q2 :\n" << q2 <<endl;
cout <<"q3 :\n" << q3 <<endl;

Q.col(0) = q1;
Q.col(1) = q2;
Q.col(2) = q3;

cout <<"Q :\n" << Q <<endl;

// Create matrix R
arma::mat R(n,n,fill::zeros);

for (int i = 0; i <n; ++i)
{
       for(int j = 0; j<n; ++j)
       {
              R[i+j*n] = dot(Q.col(i),A.col(j));
       }
}

cout <<"R :\n" << R <<endl;

cout <<"Q*R :\n" << Q*R <<endl;

// QR-Decomposition with Armadillo library
mat Q1;
mat R1;
qr(Q1,R1,A);

cout <<"QR-decomposition with automatic Armadillo function :\n"<<
    endl;
cout <<"Q :\n" << Q1 <<endl;
cout <<"R :\n" << R1 <<endl;

cout << endl;
```

```
        return 0;
    }
```

**C++ Code 203:** *main.cpp "QR-decomposition"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- For the manual computation when computing matrix $R$ we just need to compute the dot product, because of the assumption that the inner product is the Euclidean inner product.

$$< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{u} cdot \boldsymbol{v}$$

```
        ...
        ...

        arma::mat R(n,n,fill::zeros);

        for (int i = 0; i <n; ++i)
        {
            for(int j = 0; j<n; ++j)
            {
                R[i+j*n] = dot(Q.col(i),A.col(j));
            }
        }

        ...
        ...
```

```
..
Matrix A:
   1.0000        0        0
   1.0000   1.0000        0
   1.0000   1.0000   1.0000

det(A): 1
QR-decomposition with manual computation :

Vector u1 :
   1.0000
   1.0000
   1.0000

|| u1 || :1.73205

Vector u2 :
        0
   1.0000
   1.0000

|| u2 || :1.41421

Vector u3 :
        0
        0
   1.0000

|| u3 || :1

q1 :
   0.5774
   0.5774
   0.5774

q2 :
  -0.8165
   0.4082
   0.4082

q3 :
  -7.8505e-17
  -7.0711e-01
   7.0711e-01
```

```
q1 :
   0.5774
   0.5774
   0.5774

q2 :
  -0.8165
   0.4082
   0.4082

q3 :
  -7.8505e-17
  -7.0711e-01
   7.0711e-01

Q :
   5.7735e-01  -8.1650e-01  -7.8505e-17
   5.7735e-01   4.0825e-01  -7.0711e-01
   5.7735e-01   4.0825e-01   7.0711e-01

R :
   1.7321e+00   1.1547e+00   5.7735e-01
  -3.3307e-16   8.1650e-01   4.0825e-01
  -1.1102e-16            0   7.0711e-01

Q*R :
   1.0000e+00   2.2204e-16   5.5511e-17
   1.0000e+00   1.0000e+00   1.1102e-16
   1.0000e+00   1.0000e+00   1.0000e+00

QR-decomposition with automatic Armadillo function :

Q :
  -5.7735e-01   8.1650e-01  -8.7561e-17
  -5.7735e-01  -4.0825e-01  -7.0711e-01
  -5.7735e-01  -4.0825e-01   7.0711e-01

R :
  -1.7321  -1.1547  -0.5774
        0  -0.8165  -0.4082
        0        0   0.7071
```

**Figure 23.168:** *The QR-decomposition to obtain matrices Q and R as the the factorization of matrix A (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/QR-Decomposition with Armadillo/main.cpp**).*

# XII. Diagonalization and Quadratic Forms

[DF*] We have found conditions that guaranteed the diagonalizability of an $n \times n$ matrix, now we will check what class or classes of matrices might actually satisfy those conditions.

**Definition 23.56: Orthogonal Matrix**

A square matrix $A$ is said to be orthogonal if its transpose is the same as its inverse, that is, if

$$A^{-1} = A^T$$

or, equivalently, if

$$AA^T = A^T A = I$$

[DF*] The standard matrix for the counterclockwise rotation of $\mathbb{R}^2$ through an angle $\theta$ is

$$A = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

This matrix is orthogonal for all choices of $\theta$ since

$$A^T A = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

**Theorem 23.116: Equivalent Statement for Orthogonal Matrix**

The following are equivalent for an $n \times n$ matrix $A$.

(a) $A$ is orthogonal.

(b) The row vectors of $A$ form an orthonormal set in $\mathbb{R}^n$ with the Euclidean inner product.

(c) The column vectors of $A$ form an orthonormal set in $\mathbb{R}^n$ with the Euclidean inner product.

[DF*] If the row vectors of $A$ are $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_n$ then $AA^T$ can be expressed as

$$AA^T = \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \vdots \\ \boldsymbol{r}_n \end{bmatrix} \begin{bmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & \ldots & \boldsymbol{r}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_1 \cdot \boldsymbol{r}_1 & \boldsymbol{r}_1 \cdot \boldsymbol{r}_2 & \ldots & \boldsymbol{r}_1 \cdot \boldsymbol{r}_n \\ \boldsymbol{r}_2 \cdot \boldsymbol{r}_1 & \boldsymbol{r}_2 \cdot \boldsymbol{r}_2 & \ldots & \boldsymbol{r}_2 \cdot \boldsymbol{r}_n \\ \vdots & \vdots & & \vdots \\ \boldsymbol{r}_n \cdot \boldsymbol{r}_1 & \boldsymbol{r}_n \cdot \boldsymbol{r}_2 & \ldots & \boldsymbol{r}_n \cdot \boldsymbol{r}_n \end{bmatrix}$$

it follows that $AA^T = I$ if and only if

$$\boldsymbol{r}_1 \cdot \boldsymbol{r}_1 = \boldsymbol{r}_2 \cdot \boldsymbol{r}_2 = \cdots = \boldsymbol{r}_n \cdot \boldsymbol{r}_n = 1$$

and

$$\boldsymbol{r}_i \cdot \boldsymbol{r}_j = 0, \quad i \neq j$$

which are true if and only if $\{\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_n\}$ is an orthonormal set in $\mathbb{R}^n$.

**Theorem 23.117: Fundamental Properties of Orthogonal Matrices**

(a) The inverse of an orthogonal matrix is orthogonal.

(b) A product of orthogonal matrices is orthogonal.

(c) If $A$ is orthogonal, then $\det(A) = 1$ or $\det(A) = -1$.

**Theorem 23.118: Orthogonal Matrices as Linear Operators**

(a) $A$ is orthogonal.

(b) $||A\boldsymbol{x}|| = ||\boldsymbol{x}||$ for all $\boldsymbol{x}$ in $\mathbb{R}^n$.

(c) $A\boldsymbol{x} \cdot A\boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{y}$ for all $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{R}^n$.

[DF*] If $A$ is orthogonal matrix and $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is multiplication by $A$, then we will call $T_A$ orthogonal operator on $\mathbb{R}^n$.

The orthogonal operators on $\mathbb{R}^n$ are precisely those operators that leave the lengths of all vectors unchanged.

[DF*] Orthonormal bases for inner product spaces are convenient because many familiar formulas hold for such bases.

**Theorem 23.119: Change of Orthonormal Basis**

If $S$ is an orthonormal basis for an $n$-dimensional inner product space $V$, and if

$$(\boldsymbol{u})_S = (u_1, u_2, \ldots, u_n)$$
$$(\boldsymbol{v})_S = (v_1, v_2, \ldots, v_n)$$

then:

(a) $||\boldsymbol{u}|| = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2}$

(b) $d(\boldsymbol{u}, \boldsymbol{v}) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_n - v_n)^2}$

(c) $< \boldsymbol{u}, \boldsymbol{v} >= u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$

[DF*] Transitions between orthonormal bases for an inner product space are of special importance in geometry and various applications.

**Theorem 23.120: Transition Matrix for Orthonormal Basis**

Let $V$ be a finite-dimensional inner product space. If $P$ is the transition matrix from one orthonormal basis for $V$ to another orthonormal basis for $V$, then $P$ is an orthogonal matrix.

[DF*] **Example:**

What conditions must $a$ and $b$ satisfy for the matrix

$$\begin{bmatrix} a+b & b-a \\ a-b & b+a \end{bmatrix}$$

to be orthogonal?
**Solution:**
We will compute $A^T A$ then determine $a$ and $b$ from there

$$A^T A = \begin{bmatrix} 2(a^2+b^2) & 0 \\ 0 & 2(a^2+b^2) \end{bmatrix}$$

To be orthogonal the condition of $A^T A = I$ has to be fulfilled, thus

$$\begin{aligned} 2(a^2+b^2) &= 1 \\ a^2+b^2 &= \frac{1}{2} \end{aligned}$$

so $a$ and $b$ must satisfy $a^2+b^2 = \frac{1}{2}$.

[DF*] **Example:**
Find $a, b$, and $c$ for which the matrix

$$\begin{bmatrix} a & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ b & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ c & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{bmatrix}$$

is orthogonal. Are the values of $a, b$, and $c$ are unique?
**Solution:**
The row vectors of an orthogonal matrix are an orthonormal set. Thus

$$\begin{aligned} ||\boldsymbol{r}_1||^2 &= 1 \\ a^2 + \left(\frac{1}{\sqrt{2}}\right)^2 + \left(-\frac{1}{\sqrt{2}}\right)^2 &= 1 \\ a^2+1 &= 1 \\ a &= 0 \end{aligned}$$

$$\begin{aligned} ||\boldsymbol{r}_2||^2 &= 1 \\ b^2 + \left(\frac{1}{\sqrt{6}}\right)^2 + \left(\frac{1}{\sqrt{6}}\right)^2 &= 1 \\ b^2 + \frac{2}{6} &= 1 \\ b &= \pm\frac{2}{\sqrt{6}} \end{aligned}$$

$$\begin{aligned} ||\boldsymbol{r}_3||^2 &= 1 \\ c^2 + \left(\frac{1}{\sqrt{3}}\right)^2 + \left(\frac{1}{\sqrt{3}}\right)^2 &= 1 \\ c^2 + \frac{2}{3} &= 1 \\ c &= \pm\frac{1}{\sqrt{3}} \end{aligned}$$

The column vectors of an orthogonal matrix are an orthonormal set, form which it is clear that $b$ and $c$ must have opposite signs. Thus the only possibilities are

$$a = 0, \quad b = \frac{2}{\sqrt{6}}, \quad c = -\frac{1}{\sqrt{3}}$$

or

$$a = 0, \quad b = -\frac{2}{\sqrt{6}}, \quad c = \frac{1}{\sqrt{3}}$$

[DF*] Multiplication by a $3 \times 3$ orthogonal matrix $A$ is a rotation about some axis if $\det(A) = 1$ and is a rotation about some axis followed by a reflection about some coordinate plane if $\det(A) = -1$.

[DF*] A linear operator on $\mathbb{R}^2$ is called rigid if it does not change the lengths of vectors, and it is called angle preserving if it does not change the angle between nonzero vectors.

1. Linear operators that are rigid: rotations about the origin, reflections about any line through the origin.
2. Linear operators that are angle preserving: All rigid operators on $\mathbb{R}^2$ are angle preserving. Dilations and contractions are angle preserving operators that are not rigid.

**Definition 23.57: Orthogonally Similar**

If $A$ and $B$ are suare matrices, then we say that $A$ and $B$ are orthogonally similar if there is an orthogonal matrix $P$ such that $P^T AP = B$.

[DF*] If $A$ is orthogonally similar to some diagonal matrix, say

$$P^T AP = D$$

then we say that $A$ is orthogonally diagonalizable and that $P$ orthogonally diagonalizes $A$.

**Theorem 23.121: Conditions for Orthogonal Diagonalizability**

If $A$ is an $n \times n$ matrix, then the following are equivalent

(a) $A$ is orthogonally diagonalizable.

(b) $A$ has an orthonormal set of $n$ eigenvectors.

(c) $A$ is symmetric.

**Theorem 23.122: Properties of Symmetric Matrices**

If $A$ is a symmetric matrix, then:

(a) The eigenvalues of $A$ are all real numbers.

(b) Eigenvectors from different eigenspaces are orthogonal.

[DF*] Procedure for orthogonally diagonalizing a symmetric matrix.

1. Find a basis for each eigenspace of $A$.

2. Apply the Gram-Schmidt process to each of these bases to obtain an orthonormal basis for each eigenspace.

3. Form the matrix $P$ whose columns are the vectors constructed in step 2. This matrix will orthogonally diagonalize $A$, and the eigenvalues on the diagonal of $D = P^T AP$ will be in the same order as their corresponding eigenvectors in $P$.

Eigenvectors from different eigenspaces are orthogonal, and applying Gram-Schmidt process ensures that the eigenvectors within the same eigenspace are orthonormal. It follows that the entire set of eigenvectors obtained by this procedure will be orthonormal.

[DF*] If $A$ is a symmetric matrix that is orthogonally diagonalized by

$$P = [\boldsymbol{u}_1 \ \boldsymbol{u}_2 \ \dots \ \boldsymbol{u}_n]$$

and if $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of $A$ corresponding to the unit eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_n$, then we know that $D = P^T AP$, where $D$ is a diagonal matrix with the eigenvalues in the diagonal positions. It follows from this that the matrix $A$ can be expressed as

$$A = PDP^T = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \dots & \boldsymbol{u}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1^T \\ \boldsymbol{u}_2^T \\ \vdots \\ \boldsymbol{u}_n^T \end{bmatrix} = \begin{bmatrix} \lambda_1 \boldsymbol{u}_1 & \lambda_2 \boldsymbol{u}_2 & \dots & \lambda_n \boldsymbol{u}_n \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1^T \\ \boldsymbol{u}_2^T \\ \vdots \\ \boldsymbol{u}_n^T \end{bmatrix}$$

Multiplying out, we obtain the formula

$$A = \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T + \cdots + \lambda_n \boldsymbol{u}_n \boldsymbol{u}_n^T \tag{23.174}$$

which is called a spectral decomposition of $A$.

Note that in each term of the spectral decomposition of $A$ has the form $\lambda \boldsymbol{u}\boldsymbol{u}^T$, where $\boldsymbol{u}$ is a unit eigenvector of $A$ in column form, and $\lambda$ is an eigenvalue of $A$ correponding to $\boldsymbol{u}$. Since $\boldsymbol{u}$ has size $n \times 1$, it follows that the product $\boldsymbol{u}\boldsymbol{u}^T$ has size $n \times n$.

It can be proved that $\boldsymbol{u}\boldsymbol{u}^T$ is the standard matrix for the orthogonal projection of $\mathbb{R}^n$ on the subspace spanned by the vector $\boldsymbol{u}$.

The spectral decomposition of $A$ tells that the image of a vector $\boldsymbol{x}$ under multiplication by a symmetric matrix $A$ can be obtained by projecting $\boldsymbol{x}$ orthogonally on the lines (one-dimensional subspaces) determined by the eigenvectors of $A$, then scaling those projections by the eigenvalues, and then adding the scaled projections.

The terminology spectral decomposition is derived from the fact that the set of all eigenvalues of a matrix $A$ is sometimes called the spectrum of $A$. The terminology eigenvalue decomposition is due to a paper entitled "A Singularly Valuable Decomposition: The SVD of a Matrix," written by Professor Dan Kalman.

[DF*] If $A$ is an $n \times n$ matrix that is not orthogonally diagonalizable, it may still be possible to achieve considerable simplification in the form of $P^T AP$ by choosing the orthogonal matrix $P$ appropriately.

**Theorem 23.123: Schur's Theorem**

If $A$ is an $n \times n$ matrix with real entries and real eigenvalues, then there is an orthogonal matrix $P$ such that $P^T AP$ is an upper triangular matrix of the form

$$P^T AP = \begin{bmatrix} \lambda_1 & X & X & \dots & X \\ 0 & \lambda_2 & X & \dots & X \\ 0 & 0 & \lambda_3 & \dots & X \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \lambda_n \end{bmatrix} \tag{23.175}$$

in which $\lambda_1, \lambda_2, \dots, \lambda_n$ are eigenvalues of the matrix $A$ repeated according to multiplicity.

[DF*] It is common to denote the upper triangular matrix in Schur's Theorem by $S$ in which case that the equation from Schur's Theorem can be rewritten as

$$A = PSP^T \tag{23.176}$$

which is called a Schur decomposition of $A$.

**Theorem 23.124: Hessenberg's Theorem**

If $A$ is an $n \times n$ matrix, then there is an orthogonal matrix $P$ such that $P^T AP$

$$P^T AP = \begin{bmatrix} X & X & \dots & X & X & X \\ X & X & \dots & X & X & X \\ 0 & X & \ddots & X & X & X \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & X & X & X \\ 0 & 0 & \dots & 0 & X & X \end{bmatrix} \tag{23.177}$$

or in other words every square matrix with real entries is orthogonally similar to a matrix in which each entry below the first subdiagonal is zero. Such a matrix is said to be in upper Hessenberg form.

* The first subdiagonal is $\{a_{i+1,i}\}$ with $i = 1, 2, \dots, n$.

[DF*] It is common to denote the upper Hessenberg matrix by $H$, in which case that equation can be rewritten as

$$A = PHP^T \tag{23.178}$$

which is called an upper Hessenberg decomposition of $A$.

In many numerical algoriths the initial matrix is first converted to upper Hessenberg form to reduce the amount of computation in subsequent parts of the algorithm.

[DF*] **Example:**
Let $A$ be a diagonalizable matrix with the property that eigenvectors from distinct eigenvalues are orthogonal. Must $A$ be symmetric?

**Solution:**
Suppose $A$ is a matrix with size of $n \times n$, if we want to diagonalize $A$, then we will have to:

1. Compute the eigenvectors of $A$:
$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$$
2. Find the basis for each eigenspace of $A$ by solving
$$(\lambda_i I - A)\boldsymbol{x} = \mathbf{0}$$
with $i = 1, 2, \ldots, n$. The basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ will be called eigenvectors of $A$.
3. Apply the Gram-Schmidt process to each of the bases / the eigenvectors we obtain from step (2) to convert $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ into an orthogonal set $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$.

   We can then normalize $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ to become an orthonormal set.
4. The matrix $P$ that diagonalizes $A$ consist of the orthonormal set $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ as the column vectors.
$$P = [\boldsymbol{u}_1 \mid \boldsymbol{u}_2 \mid \ \ldots \ \mid \boldsymbol{u}_n]$$

The Gram-Schmidt process will ensure that columns of P corresponding to the same eigenvalue are an orthonormal set. Since eigenvectors from distinct eigenvalues are orthogonal, this means that $P$ will be an orthogonal matrix. Then since $A$ is orthogonally diagonalizable, it must be symmetric. From Theorem that states if $A$ is an $n \times n$ matrix, whenever $A$ is orthogonally diagonalizable with $n$ orthonormal set of eigenvectors then $A$ is symmetric as well.

[DF*] **Example:**
We will establish that a matrix $A$ is orthogonaly diagonalizable if and only if it is symmetric. We have shown that an orthogonally diagonalizable matrix is symmetric. The harder part is to prove that a symmetric matrix $A$ is orthogonally diagonalizable. We will proceed in two steps: first we will show that $A$ is diagonalizable, and then we will build on that result to show that $A$ is orthogonally diagonalizable.

(a) Assume that $A$ is a symmetric $n \times n$. One way to prove that $A$ is diagonalizable is to show that for each eigenvalue $\lambda_0$ the geometric multiplicity is equal to the algebraic multiplicity. For this purpose, assume that the geometric multiplicity of $\lambda_0$ is $k$, let $B_0 = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k\}$ be an orthonormal basis for the eigenspace corresponding to $\lambda_0$, extend this to an orthonormal basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ for $\mathbb{R}^n$, and let $P$ be the amtrix having the vectors of $B$ as columns. The product of $AP$ can be written as
$$AP = P \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$
Use the fact that $B$ is an orthonormal basis to prove that $X = 0$ (a zero matrix of size $n \times (n-k)$)

(b) It follows from part (a) that $A$ has the same characteristic polynomial as
$$C = \begin{bmatrix} \lambda_0 I_k & 0 \\ 0 & Y \end{bmatrix}$$
Use this fact to prove that the algebraic multiplicity of $\lambda_0$ is the same as the geometric multiplicity of $\lambda_0$. This establishes that $A$ is diagonalizable.

(c) Use Theorem for Properties of Symmetric Matrices and the fact that $A$ is diagonalizable to prove that $A$ is orthogonally diagonalizable.

**Solution:**

(a) If $\lambda_0$ is an eigenvalue of an $n \times n$ matrix $A$, then the dimension of the eigenspace corresponding to $\lambda_0$ is called the geometric multiplicity of $\lambda_0$. In simpler words, when we compute the solution (the nontrivial solution) for

$$(\lambda_0 I - A)\boldsymbol{x} = \mathbf{0}$$

The geometric multiplicity is the number of eigenvectors / the bases for the eigenspace corresponding to $\lambda = \lambda_0$.

The number of times that $\lambda - \lambda_0$ appears as a factor in the characteristic polynomial of $A$ is called the algebraic multiplicity of $\lambda_0$.

We know that $A$ is a symmetric matrix, thus

$$A = A^T$$

Assuming that the geometric multiplicity of $\lambda_0$ is $k$, we will have the bases for the eigenspace corresponding to $\lambda = \lambda_0$ as $B_0 = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k\}$, these basis is an orthonormal basis.

We can then extend $B_0$ to become an orthonormal basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ for $\mathbb{R}^n$. We can create a matrix $P$ with vectors of $B$ as columns,

$$P = [\boldsymbol{u}_1 \ \boldsymbol{u}_2 \ \ldots \ \boldsymbol{u}_n]$$

Since the orthonormal basis $B$ / the eigenvectors of $A$ are orthonormal, then $P$ is orthogonal (column vectors of $P$ form an orthonormal set in $\mathbb{R}^n$ with the Euclidean inner product) thus making $P$ orthogonally diagonalizes $A$

$$P^T A P = C$$

Since $P$ is orthogonal then

$$P^T = P^{-1}$$

multiplying with $P$ at the left to both side of the equation will result in

$$AP = PC$$

We know that $C$ is a diagonal matrix and it is written as

$$AP = P \begin{bmatrix} \lambda_0 I_k & X \\ 0 & Y \end{bmatrix}$$

Since $C$ is a diagonal matrix then $X = 0$ (a zero matrix with size of $(n-k) \times (n-k)$). The product of $AP$ is $P$ times a diagonal matrix $C$.

(b) We have

$$C = \begin{bmatrix} \lambda_0 I_k & 0 \\ 0 & Y \end{bmatrix}$$

with $Y \neq 0$ with size of $(n-k) \times (n-k)$, this can be established from the theorem that states $\lambda = 0$ is not an eigenvalue. So we will have

$$\det(\lambda I - C) = (\lambda - \lambda_0)^k \det(\lambda I_{n-k} - Y)$$

By similarity invariants, when $A$ is diagonalizable, then the characteristic polynomial of $A$ will be the same with the characteristic polynomial of $C$, this will also works for orthogonally diagonalizable matrix $A$. Thus

$$\begin{aligned}\det(\lambda I - A) &= \det(\lambda I - C) \\ &= (\lambda - \lambda_0)^k \det(\lambda I_{n-k} - Y)\end{aligned}$$

This means that the algebraic multiplicity of $\lambda_0$ is $k$, the same as the geometric multiplicity of $\lambda_0$, when the the geometric multiplicity of every eigenvalue is equal to the algebraic multiplicity then $A$ is diagonalizable.

(c) We know that $A$ is diagonalizable from part (b), $A$ is symmetric, and that $A$ has an orthonormal set of $n$ eigenvectors that can be written as orthonormal basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$, these orthonormal eigenvectors are the columns for matrix $P$ that orthogonally diagonalizes $A$ so that

$$P^T A P = \begin{bmatrix} \lambda_0 I_k & 0 \\ 0 & Y \end{bmatrix}$$

$A$ is orthogonally diagonalizable.

## i. C++ Plot and Computation: Rotation of Axes in 2-Space

In many problems a rectangular $xy$-coordinate system is given, and a new $x'y'$-coordinate system is obtained by rotating the $xy$-system counterclockwise about the origin through an angle $\theta$. When this is done, each point $Q$ in the plane has two sets of coordinates: coordinates $(x, y)$ relative to the $xy$-system and coordinates $(x', y')$ relative to the $x'y'$-system.

By introducing unit vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ along the positive $x$- and $y$-axes and unit vectors $\boldsymbol{u}_1'$ and $\boldsymbol{u}_2'$ along the positive $x'$- and $y'$-axes, we can regard this rotation as a change from an old basis $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ to a new basis $B' = \{\boldsymbol{u}_1', \boldsymbol{u}_2'\}$. Thus, the new coordinates $(x', y')$ and the old coordinates $(x, y)$ of a point $Q$ will be related by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = P^{-1} \begin{bmatrix} x \\ y \end{bmatrix} \tag{23.179}$$

where $P$ is the transition from $B'$ to $B$. To find $P$ we must determine the coordinate matrices of the new basis vectors $\boldsymbol{u}_1'$ and $\boldsymbol{u}_2'$ relative to the old basis.

As indicated, the components of $\boldsymbol{u}_1'$ in the old basis are $\cos\ \theta$ and $\sin\ \theta$,

$$\begin{aligned} x'_{\boldsymbol{u}_1'} &= \cos\ \theta \\ y'_{\boldsymbol{u}_1'} &= \sin\ \theta \end{aligned}$$

so

$$[\boldsymbol{u}_1']_B = \begin{bmatrix} \cos\ \theta \\ \sin\ \theta \end{bmatrix}$$

Similarly, the components of $\boldsymbol{u}_2'$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}_2'} &= \cos\ (\theta + \pi/2) = -\sin\ \theta \\ y'_{\boldsymbol{u}_2'} &= \sin\ (\theta + \pi/2) = \cos\ \theta \end{aligned}$$

so

$$[\boldsymbol{u}_2']_B = \begin{bmatrix} -\sin\ \theta \\ \cos\ \theta \end{bmatrix}$$

Thus the transition matrix from $B'$ to $B$ is

$$P = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \tag{23.180}$$

Observe that $P$ is an orthogonal matrix, since $B$ and $B'$ are orthonormal bases. Thus

$$P^{-1} = P^T = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix}$$

this yields

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{23.181}$$

or, equivalently,

$$\begin{aligned} x' &= x\ \cos\ \theta + y\ \sin\ \theta \\ y' &= -x\ \sin\ \theta + y\ \cos\ \theta \end{aligned} \tag{23.182}$$

These are sometimes called the rotation equations for $\mathbb{R}^2$.

We will try with a point $Q(2, -1)$ in the $xy$-system, if the coordinate axes of a rectangular coordinate system are rotated through an angle of $\theta = \frac{\pi}{4}$, the new coordinate of the point $Q$ will be

$$\begin{aligned} x' &= 2\ \cos\ \frac{\pi}{4} + (-1)\ \sin\ \frac{\pi}{4} = \frac{1}{\sqrt{2}} = 0.707107 \\ y' &= -2\ \sin\ \frac{\pi}{4} + (-1)\ \cos\ \frac{\pi}{4} = -\frac{3}{\sqrt{2}} = -2.12132 \end{aligned} \tag{23.183}$$

The C++ code is using **Armadillo** for the matrix and vector computation, and **gnuplot** to do the plotting. We create 2 file inputs for point $Q(2, -1)$ the first one is **vectorQ.txt** that will be loaded by **Armadillo** for computation purpose, the second one is **vectorQ.dat** that will be called by **gnuplot** to plot it as a point.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define theta 45 // Define the degree of rotation
#define N 2 // Define the number of dimension
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

void PrintMatrix(float a[][N])
{
	int r = N;
	int c = N;
	for (int i = 0; i < r; i++)
	{
		for (int j = 0; j < c; j++)
		cout <<setw(6) << setprecision(0) << a[i][j] << "\t";
		cout << endl;
	}
}

int main() {
	Gnuplot gp;

	// Armadillo
```

```
arma::mat X;
arma::mat Y;
arma::vec Q;
X.load("vectorX.txt");
Y.load("vectorY.txt");
Q.load("vectorQ.txt");
cout <<"The xy-system:" << endl;
cout <<"Vector x:" << "\n" << X <<endl;
cout <<"Vector y:" << "\n" << Y <<endl;
cout <<"Point Q: ( " << Q[0] << ", " << Q[1] << " )" << endl;

float matrixA[N][N] = {};
matrixA[0][0] = cos(theta*DEGTORAD);
matrixA[0][1] = -sin(theta*DEGTORAD);
matrixA[1][0] = sin(theta*DEGTORAD);
matrixA[1][1] = cos(theta*DEGTORAD);

arma::mat ArmaA(N,N,fill::zeros); // Declare matrix ArmaA of size N
    X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaA[i+j*N] = matrixA[i][j] ;
       }
}

cout <<"P (Transition matrix from B' to B)" << theta << " :" << "\n"
     << ArmaA <<endl;

arma::mat Xrotate = ArmaA*X ;
arma::mat Yrotate = ArmaA*Y ;

cout <<"The x'y'-system:" << endl;
cout <<"The rotated vector x / x':" << "\n" << Xrotate <<endl;
cout <<"The rotated vector y / y':" << "\n" << Yrotate <<endl;

arma::mat ArmaA_Transpose(N,N,fill::zeros);
ArmaA_Transpose = ArmaA.t();
cout <<"P^{-1} = P^{T} (Transition matrix from B to B') " << "\n"
    << ArmaA_Transpose <<endl;
arma::mat PTQ;
PTQ = ArmaA_Transpose*Q;
cout << endl;
cout <<"Q relative to the xy-system:" << endl;
cout <<"(x, y) = " << "( " << Q[0] << ", " << Q[1] << " )"<<endl;
cout <<"Q relative to the x'y'-system:" << endl;
cout <<"(x', y') = " << "( " << PTQ[0] << ", " << PTQ[1] << " )"<<
```

```
            endl;

        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;
        std::vector<double> pts_G_x;
        std::vector<double> pts_G_y;
        std::vector<double> pts_G_dx;
        std::vector<double> pts_G_dy;

        float o = 0;

        // Create a vector x
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_dx.push_back(X[0]);
        pts_D_dy.push_back(X[1]);
        // The counterclockwise rotation of vector x through angle theta
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(Xrotate[0]);
        pts_E_dy.push_back(Xrotate[1]);
        // Create a vector y
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(Y[0]);
        pts_F_dy.push_back(Y[1]);
        // The counterclockwise rotation of vector y through angle theta
        pts_G_x .push_back(o);
        pts_G_y .push_back(o);
        pts_G_dx.push_back(Yrotate[0]);
        pts_G_dy.push_back(Yrotate[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-1.1:2.1]\nset yrange [-1.5:1.5]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
```

```
            // '—' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "f(x) = x\n";
            gp << "plot 'vectorQ.dat' with points pt 7 ps 2 title 'Q', \
            '—' with vectors lw 1 lc rgb 'dark—green' title '{/:Bold {u}_
                {1}}','—' with vectors lw 3 lc rgb 'dark—green' title '{/:Bold
                 u_{1}‘}',\
            '—' with vectors lw 1 lc rgb 'dark—turquoise' title '{/:Bold {u}_
                {2}}','—' with vectors lw 3 lc rgb 'dark—turquoise' title '{/:
                Bold u_{2}‘}'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_dx, pts_G_dy));
        }
```

**C++ Code 204:** *main.cpp "Rotation of axes in 2 dimensional space"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Vector x:
   1.0000
        0

Vector y:
        0
   1.0000

Point Q: ( 2, -1 )
P (Transition matrix from B' to B)45 :
   0.7071  -0.7071
   0.7071   0.7071

The x'y'-system:
The rotated vector x / x':
   0.7071
   0.7071

The rotated vector y / y':
  -0.7071
   0.7071

P^{-1} = P^{T} (Transition matrix from B to B')
   0.7071   0.7071
  -0.7071   0.7071


Q relative to the xy-system:
(x, y) = ( 2, -1 )
Q relative to the x'y'-system:
(x', y') = ( 0.707107, -2.12132 )
```

**Figure 23.169:** *The computation to obtain the new basis $\{\boldsymbol{u}_1', \boldsymbol{u}_2'\}$ from rotating the old basis counterclockwise about the origin through an angle $\theta = \frac{\pi}{4}$ and the new coordinates of the point $Q$ relative to the $x'y'$-system. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Rotation of Axes in 2-Space/main.cpp**).*

**Figure 23.170:** *The plot of the old basis* $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ *and new basis* $\{\boldsymbol{u}_1', \boldsymbol{u}_2'\}$ *from rotating the old basis counterclockwise about the origin through an angle* $\theta = \frac{\pi}{4}$. *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Rotation of Axes in 2-Space/main.cpp**).*

## ii. C++ Plot and Computation: Rotation of Axes in 3-Space Around $z$-Axis

Suppose that a rectangular $xyz$-coordinate system is rotated around its $z$-axis counterclockwise (looking toward the origin from the positive $z$-axis) through an angle $\theta$. If we introduce unit vectors $\boldsymbol{u}_1, \boldsymbol{u}_2$, and $\boldsymbol{u}_3$ along the positive $x$-, $y$-, and $z$-axes and unit vectors $\boldsymbol{u}'_1, \boldsymbol{u}'_2$, and $\boldsymbol{u}'_3$ along the positive $x'$-, $y'$-, and $z'$-axes, we can regard the rotation as a change from the old basis

$$B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$$

to the new basis

$$B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \boldsymbol{u}'_3\}$$

As indicated, the components of $\boldsymbol{u}'_1$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}'_1} &= \cos\ \theta \\ y'_{\boldsymbol{u}'_1} &= \sin\ \theta \\ z'_{\boldsymbol{u}'_1} &= 0 \end{aligned}$$

so

$$[\boldsymbol{u}'_1]_B = \begin{bmatrix} \cos\ \theta \\ \sin\ \theta \\ 0 \end{bmatrix}$$

the components of $\boldsymbol{u}'_2$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}'_2} &= -\sin\ \theta \\ y'_{\boldsymbol{u}'_2} &= \cos\ \theta \\ z'_{\boldsymbol{u}'_2} &= 0 \end{aligned}$$

so

$$[\boldsymbol{u}'_2]_B = \begin{bmatrix} -\sin\ \theta \\ \cos\ \theta \\ 0 \end{bmatrix}$$

since the rotation is around the $z$ axis counterclockwise, thus the components of $\boldsymbol{u}'_3$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}'_3} &= 0 \\ y'_{\boldsymbol{u}'_3} &= 0 \\ z'_{\boldsymbol{u}'_3} &= 1 \end{aligned}$$

so

$$[\boldsymbol{u}'_3]_B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus the transition matrix from $B'$ to $B$ is

$$P = \begin{bmatrix} \cos\ \theta & -\sin\ \theta & 0 \\ \sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23.184}$$

Observe that $P$ is an orthogonal matrix, since $B$ and $B'$ are orthonormal bases. Thus

$$P^{-1} = P^T = \begin{bmatrix} \cos\ \theta & \sin\ \theta & 0 \\ -\sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, the new coordinates $(x', y', z')$ of a point $Q$ can be computed from it sold coordinates $(x, y, z)$ by

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\ \theta & \sin\ \theta & 0 \\ -\sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{23.185}$$

We will try with a point $Q(0.6, 0.2, 0.5)$ in the $xyz$-system, if the coordinate axes of a rectangular coordinate system are rotated around its $z$-axis coutnerclockwise through an angle of $\theta = \frac{\pi}{3}$, the new coordinate of the point $Q$ will be

$$\begin{aligned} x' &= (0.6)\ \cos\ \frac{\pi}{3} + (0.2)\ \sin\ \frac{\pi}{3} = 0.473205 \\ y' &= -(0.6)\ \sin\ \frac{\pi}{3} + (0.2)\ \cos\ \frac{\pi}{3} = -0.419615 \\ z' &= z = 0.5 \end{aligned} \tag{23.186}$$

Since the rotation is around the $z$-axis, then the value of $z$ doesn't change. We can see the point $Q$ with respect to $x'y'z'$-system has the $y'$ value as negative after the rotation.

If you want to dig deeper, this can be connected to planet rotation toward the sun, when the Earth is revolving around the sun for 365 days, it is also rotating everyday, the position of the sun stays the same, like the point $Q$ here, but the rotation that occurs make a periodic $x'y'z'$-system for the Earth till the next day, that is why we have day and night, we see the Sun as point $Q$ in new $x'y'z'$-system that changes every seconds or the smallest time interval we can count, and repeated after 24 hours.

Almost the same like previous C++ code, the C++ code here is using **Armadillo** for the matrix and vector computation, and **gnuplot** to do the plotting. We create 2 file inputs for point $Q(0.6, 0.2, 0.5)$ the first one is **vectorQ.txt** that will be loaded by **Armadillo** for computation purpose, the second one is **vectorQ.dat** that will be called by **gnuplot** to plot it as a point.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension
```

```
#define theta 60 // Define the degree of rotation
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

int main() {
	Gnuplot gp;

	// Armadillo
	arma::mat X;
	arma::mat Y;
	arma::mat Z;
	arma::mat Q;
	X.load("vectorX.txt");
	Y.load("vectorY.txt");
	Z.load("vectorZ.txt");
	Q.load("vectorQ.txt");
	cout <<"Vector u1:" << "\n" << X <<endl;
	cout <<"Vector u2:" << "\n" << Y <<endl;
	cout <<"Vector u3:" << "\n" << Z <<endl;

	// Create standard matrix for rotation about x-axis
	float matrixAx[N][N] = {};
	// Create standard matrix for rotation about y-axis
	float matrixAy[N][N] = {};
	// Create standard matrix for rotation about z-axis
	float matrixAz[N][N] = {};
	for (int i = 0; i < N; ++i)
	{
		for(int j = 0; j<N; ++j)
		{
			if (i == j)
			{
				matrixAx[i][j] = 1;
				matrixAy[i][j] = 1;
				matrixAz[i][j] = 1;
			}
			else
			{
				matrixAx[i][j] = 0;
				matrixAy[i][j] = 0;
				matrixAz[i][j] = 0;
			}
		}
	}
	matrixAx[1][1] = cos(theta*DEGTORAD);
	matrixAx[1][2] = -sin(theta*DEGTORAD);
```

```
matrixAx[2][1] = sin(theta*DEGTORAD);
matrixAx[2][2] = cos(theta*DEGTORAD);

matrixAy[0][0] = cos(theta*DEGTORAD);
matrixAy[0][2] = sin(theta*DEGTORAD);
matrixAy[2][0] = -sin(theta*DEGTORAD);
matrixAy[2][2] = cos(theta*DEGTORAD);

matrixAz[0][0] = cos(theta*DEGTORAD);
matrixAz[0][1] = -sin(theta*DEGTORAD);
matrixAz[1][0] = sin(theta*DEGTORAD);
matrixAz[1][1] = cos(theta*DEGTORAD);

arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
arma::mat ArmaAy(N,N,fill::zeros); // Declare matrix ArmaAy of size
    N X N with Armadillo
arma::mat ArmaAz(N,N,fill::zeros); // Declare matrix ArmaAz of size
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAx[i+j*N] = matrixAx[i][j] ; // Standard Matrix
                  for rotation about x-axis
              ArmaAy[i+j*N] = matrixAy[i][j] ; // Standard Matrix
                  for rotation about y-axis
              ArmaAz[i+j*N] = matrixAz[i][j] ; // Standard Matrix
                  for rotation about z-axis
       }
}
// Standard Matrix for rotation about z-axis = Transition matrix
    from B' to B
arma::mat ArmaAz_Transpose = ArmaAz.t();
cout <<"P (Transition matrix from B' to B):" << "\n" << ArmaAz <<
    endl;
cout <<"P^{T} (Transition matrix from B to B'):" << "\n" <<
    ArmaAz_Transpose <<endl;

arma::mat Xrotatez = ArmaAz*X ;
arma::mat Yrotatez = ArmaAz*Y ;
cout <<"The rotated vector u1 about positive z-axis:" << "\n" <<
    Xrotatez <<endl;
cout <<"The rotated vector u2 about positive z-axis:" << "\n" <<
    Yrotatez <<endl;

arma::mat PTQ;
PTQ = ArmaAz_Transpose*Q;
```

```
cout <<"Q relative to the xyz—system:" << endl;
cout <<"(x, y,z) = " << "( " << Q[0] << ", " << Q[1] << ", " << Q[2]
     << " )"<<endl;
cout <<"Q relative to the x'y'z'—system:" << endl;
cout <<"(x', y', z') = " << "( " << PTQ[0] << ", " << PTQ[1] << ", "
     << PTQ[2] << " )"<<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;
std::vector<double> pts_H_x;
std::vector<double> pts_H_y;
std::vector<double> pts_H_z;
std::vector<double> pts_H_dx;
std::vector<double> pts_H_dy;
std::vector<double> pts_H_dz;
float o = 0;

// Create a vector u1
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(X[0]);
pts_D_dy.push_back(X[1]);
pts_D_dz.push_back(X[2]);
```

```
            // Create a vector u2
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(Y[0]);
            pts_E_dy.push_back(Y[1]);
            pts_E_dz.push_back(Y[2]);
            // Create a vector u3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(Z[0]);
            pts_F_dy.push_back(Z[1]);
            pts_F_dz.push_back(Z[2]);
            // Rotated vector u1 about positive z-axis
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(Xrotatez[0]);
            pts_G_dy.push_back(Xrotatez[1]);
            pts_G_dz.push_back(Xrotatez[2]);
            // Rotated vector u2 about positive z-axis
            pts_H_x .push_back(o);
            pts_H_y .push_back(o);
            pts_H_z .push_back(o);
            pts_H_dx.push_back(Yrotatez[0]);
            pts_H_dy.push_back(Yrotatez[1]);
            pts_H_dz.push_back(Yrotatez[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [-0.5:1]\nset yrange [-0.5:1]\nset zrange
                [-1:1]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";
            //gp << "set view 80,45,1\n"; // pitch,yaw,zoom for diagonal x and y
                 axis
            gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
            //gp << "set view 80,0,1\n"; // pitch,yaw,zoom for x axis on front
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "splot '-' with vectors lw 1 lc rgb 'dark-green' title '{/:
                Bold {u}_{1}}',\
            '-' with vectors lw 1 lc rgb 'dark-turquoise' title '{/:Bold {u}_
                {2}}',\
            '-' with vectors lw 1 lc rgb 'dark-red' title '{/:Bold {u}_{3}}',\
            '-' with vectors lw 3 lc rgb 'dark-green' title 'rotated {/:Bold {u
```

```
                }_{1}}',\
            '-' with vectors lw 3 lc rgb 'dark-turquoise' title 'rotated {/:
                Bold {u}_{2}}',\
            'vectorQ.dat' with points pt 7 ps 2 title 'Q'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
            gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx,
                pts_H_dy, pts_H_dz));
    }
```

**C++ Code 205:** *main.cpp "Rotation of axes in 3 dimensional space around z-axis"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- If you wish to see the rotation about another axis besides $z$ axis, like rotation about $x$-axis or $y$-axis then you need to change **ArmaAz** matrix here with:

  **ArmaAx** (for rotation about $x$-axis, the counterclockwise rotation can be imagined while looking at the negative $x$-axis from positive $x$-axis)

  **ArmaAy** (for rotation about $y$-axis, the counterclockwise rotation can be imagined while looking at the negative $y$-axis from positive $y$-axis)

  Then adjust the rest of the code with the rotation you want

```
            arma::mat ArmaAz_Transpose = ArmaAz.t();
            cout <<"P (Transition matrix from B' to B):" << "\n" << ArmaAz
                <<endl;
            cout <<"P^{T} (Transition matrix from B to B'):" << "\n" <<
                ArmaAz_Transpose <<endl;

            arma::mat Xrotatez = ArmaAz*X ;
            arma::mat Yrotatez = ArmaAz*Y ;
            cout <<"The rotated vector u1 about positive z-axis:" << "\n"
                << Xrotatez <<endl;
```

```
cout <<"The rotated vector u2 about positive z—axis:" << "\n"
    << Yrotatez <<endl;

arma::mat PTQ;
PTQ = ArmaAz_Transpose*Q;

cout <<"Q relative to the xyz—system:" << endl;
cout <<"(x, y,z) = " << "( " << Q[0] << ", " << Q[1] << ", " <<
     Q[2] << " )"<<endl;
cout <<"Q relative to the x'y'z'—system:" << endl;
cout <<"(x', y', z') = " << "( " << PTQ[0] << ", " << PTQ[1] <<
     ", " << PTQ[2] << " )"<<endl;
```

**Figure 23.171:** *The plot of the old basis $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ and new basis $\{\boldsymbol{u}'_1, \boldsymbol{u}'_2, \boldsymbol{u}'_3\}$ from rotating the old basis counterclockwise around the z-axis (looking down) through an angle $\theta = \frac{\pi}{3}$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation of Axes/main.cpp**).*

```
P (Transition matrix from B' to B):
   0.5000  -0.8660        0
   0.8660   0.5000        0
        0        0   1.0000

P^{T} (Transition matrix from B to B'):
   0.5000   0.8660        0
  -0.8660   0.5000        0
        0        0   1.0000

The rotated vector u1 about positive z-axis:
   0.5000
   0.8660
        0

The rotated vector u2 about positive z-axis:
  -0.8660
   0.5000
        0

Q relative to the xyz-system:
(x, y,z) = ( 0.6, 0.2, 0.5 )
Q relative to the x'y'z'-system:
(x', y', z') = ( 0.473205, -0.419615, 0.5 )
```

**Figure 23.172:** *The computation to obtain the new basis $\{\boldsymbol{u}_1', \boldsymbol{u}_2', \boldsymbol{u}_3'\}$ from rotating the old basis counterclockwise around the z-axis (looking down) through an angle $\theta = \frac{\pi}{3}$, and obtain the new coordinates of the point Q relative to the $x'y'z'$-system. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation of Axes/main.cpp**).*

## iii. C++ Plot and Computation: Rotation of Axes in 3-Space Around $z$-Axis then $y'$-Axis

A rectangular $x''y''z''$-coordinate system is obtained by first rotating a rectangular $xyz$-coordinate system $60^0$ counterclockwise about the $z$-axis (looking toward the origin from the positive $z$-axis) to obtain an $x'y'z'$-coordinate system, and then rotating $x'y'z'$-coordinate system $45^0$ counterclockwise about the $y'$-axis (looking toward the origin from positive $y'$-axis). Find a matrix $A$ such that

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = A \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

where $(x, y, z)$ and $(x'', y'', z'')$ are the $xyz$- and $x''y''z''$-coordinates of the same point.

**Solution:**

We will use the same method as the previous section, we set the old basis

$$B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$$

and the new basis

$$B' = \{\boldsymbol{u}_1', \boldsymbol{u}_2', \boldsymbol{u}_3'\}$$

As indicated, the components of $\boldsymbol{u}_1'$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}_1'} &= \cos\ \theta \\ y'_{\boldsymbol{u}_1'} &= \sin\ \theta \\ z'_{\boldsymbol{u}_1'} &= 0 \end{aligned}$$

so

$$[\boldsymbol{u}_1']_B = \begin{bmatrix} \cos\ \theta \\ \sin\ \theta \\ 0 \end{bmatrix}$$

the components of $\boldsymbol{u}_2'$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}_2'} &= -\sin\ \theta \\ y'_{\boldsymbol{u}_2'} &= \cos\ \theta \\ z'_{\boldsymbol{u}_2'} &= 0 \end{aligned}$$

so

$$[\boldsymbol{u}_2']_B = \begin{bmatrix} -\sin\ \theta \\ \cos\ \theta \\ 0 \end{bmatrix}$$

since the rotation is around the $z$ axis counterclockwise, thus the components of $\boldsymbol{u}_3'$ in the old basis are

$$\begin{aligned} x'_{\boldsymbol{u}_3'} &= 0 \\ y'_{\boldsymbol{u}_3'} &= 0 \\ z'_{\boldsymbol{u}_3'} &= 1 \end{aligned}$$

so

$$[\boldsymbol{u}_3']_B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus the transition matrix from $B'$ to $B$ is

$$P_z = \begin{bmatrix} \cos\ \theta & -\sin\ \theta & 0 \\ \sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23.187}$$

Observe that $P$ is an orthogonal matrix, since $B$ and $B'$ are orthonormal bases. Thus

$$P_z^{-1} = P_z^T = \begin{bmatrix} \cos\ \theta & \sin\ \theta & 0 \\ -\sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, the new coordinates $(x', y', z')$ rotated counterclockwise around $z$-axis $60^0$ (we will define this degree with notation of $\theta$) of a point $Q$ can be computed from its' old coordinates $(x, y, z)$ by

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\ \theta & \sin\ \theta & 0 \\ -\sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{23.188}$$

Now from this we want to rotate the $(x', y', z')$-coordinate system $45^0$ (we will define this degree with notation of $\phi$) about the $y'$ axis, so first we need to continue and define the new basis for $(x'', y'', z'')$-coordinate system

$$B'' = \{\boldsymbol{u}_1'', \boldsymbol{u}_2'', \boldsymbol{u}_3''\}$$

To understand the basis $B''$ we have to learn on how to obtain the transition matrix from $B'$ to $B''$, we will use the standard matrix for a counterclockwise rotation through an angle $\phi$ about an axis in $\mathbb{R}^3$, which is determined by the unit vector $\boldsymbol{y'} = (a, b, c) = (-\sin\ \theta, \cos\ \theta, 0)$ that has its' initial point at the origin is

$$\begin{bmatrix} a^2(1-\cos\ \phi)+\cos\ \phi & ab(1-\cos\ \phi)-c\ \sin\ \phi & ac(1-\cos\ \phi)+b\ \sin\ \phi \\ ab(1-\cos\ \phi)+c\ \sin\ \phi & b^2(1-\cos\ \phi)+\cos\ \phi & bc(1-\cos\ \phi)-a\ \sin\ \phi \\ ac(1-\cos\ \phi)-b\ \sin\ \phi & bc(1-\cos\ \phi)+a\ \sin\ \phi & c^2(1-\cos\ \phi)+\cos\ \phi \end{bmatrix}$$

we will have

$$[\boldsymbol{u}_1'']_{B'} = \begin{bmatrix} a^2(1-\cos\ \phi)+\cos\ \phi \\ ab(1-\cos\ \phi)+c\ \sin\ \phi \\ ac(1-\cos\ \phi)-b\ \sin\ \phi \end{bmatrix}, \quad [\boldsymbol{u}_2'']_{B'} = \begin{bmatrix} ab(1-\cos\ \phi)-c\ \sin\ \phi \\ b^2(1-\cos\ \phi)+\cos\ \phi \\ bc(1-\cos\ \phi)+a\ \sin\ \phi \end{bmatrix}, \quad [\boldsymbol{u}_3'']_{B'} = \begin{bmatrix} ac(1-\cos\ \phi)+b\ \sin\ \phi \\ bc(1-\cos\ \phi)-a\ \sin\ \phi \\ c^2(1-\cos\ \phi)+\cos\ \phi \end{bmatrix}$$

Thus

$$P_{y'} = \begin{bmatrix} a^2(1-\cos\ \phi)+\cos\ \phi & ab(1-\cos\ \phi)-c\ \sin\ \phi & ac(1-\cos\ \phi)+b\ \sin\ \phi \\ ab(1-\cos\ \phi)+c\ \sin\ \phi & b^2(1-\cos\ \phi)+\cos\ \phi & bc(1-\cos\ \phi)-a\ \sin\ \phi \\ ac(1-\cos\ \phi)-b\ \sin\ \phi & bc(1-\cos\ \phi)+a\ \sin\ \phi & c^2(1-\cos\ \phi)+\cos\ \phi \end{bmatrix} \tag{23.189}$$

Observe that $P_{y'}$ is an orthogonal matrix, since $B'$ and $B''$ are orthonormal bases. Thus

$$P_{y'}^{-1} = P_{y'}^{T} = \begin{bmatrix} a^2(1-\cos\ \phi)+\cos\ \phi & ab(1-\cos\ \phi)+c\ \sin\ \phi & ac(1-\cos\ \phi)-b\ \sin\ \phi \\ ab(1-\cos\ \phi)-c\ \sin\ \phi & b^2(1-\cos\ \phi)+\cos\ \phi & bc(1-\cos\ \phi)+a\ \sin\ \phi \\ ac(1-\cos\ \phi)+b\ \sin\ \phi & bc(1-\cos\ \phi)-a\ \sin\ \phi & c^2(1-\cos\ \phi)+\cos\ \phi \end{bmatrix}$$

Thus, the new coordinates $(x'', y'', z'')$ of a point $Q$ can be computed from its' old coordinates $(x, y, z)$ by

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = P_{y'}^{-1} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = P_{y'}^{-1} P_z^{-1} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{23.190}$$

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} a^2(1-\cos\ \phi)+\cos\ \phi & ab(1-\cos\ \phi)+c\ \sin\ \phi & ac(1-\cos\ \phi)-b\ \sin\ \phi \\ ab(1-\cos\ \phi)-c\ \sin\ \phi & b^2(1-\cos\ \phi)+\cos\ \phi & bc(1-\cos\ \phi)+a\ \sin\ \phi \\ ac(1-\cos\ \phi)+b\ \sin\ \phi & bc(1-\cos\ \phi)-a\ \sin\ \phi & c^2(1-\cos\ \phi)+\cos\ \phi \end{bmatrix} \begin{bmatrix} \cos\ \theta & \sin\ \theta & 0 \\ -\sin\ \theta & \cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{23.191}$$

thus we will have

$$A = P_{y'}^{-1} P_z^{-1}$$

as a matrix that rotate axes counterclockwise with angle of $\theta$ about the $z$-axis then rotate the axes counterclockwise again about the $y'$-axis with angle of $\phi$.

A rule of thumb for rotation:

1. Rotation about $x$-axis, then the movement will occur on $yz$-plane, if the rotation is about $x'$-axis, then the movement will occur on $y'z'$-plane, and so on.
2. Rotation about $y$-axis, then the movement will occur on $xz$-plane, if the rotation is about $y'$-axis, then the movement will occur on $x'z'$-plane, and so on.
3. Rotation about $z$-axis, then the movement will occur on $xy$-plane, if the rotation is about $z'$-axis, then the movement will occur on $x'y'$-plane, and so on.

Just like previous C++ code, the C++ code here is using **Armadillo** for the matrix and vector computation, and **gnuplot** to do the plotting. We create 2 file inputs for point $Q(0.6, 0.2, 0.5)$ the first one is **vectorQ.txt** that will be loaded by **Armadillo** for computation purpose, the second one is **vectorQ.dat** that will be called by **gnuplot** to plot it as a point.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <vector>
#include <cmath>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

#define N 3 // Define the number of dimension
#define theta 60 // Define the degree of rotation about z-axis
#define phi 45 // Define the degree of rotation about y'-axis
#define DEGTORAD 0.0174532925199432957f

using namespace std;
using namespace arma;

int main() {
       Gnuplot gp;

       // Armadillo
       arma::mat X;
       arma::mat Y;
       arma::mat Z;
       arma::mat Q;
       X.load("vectorX.txt");
       Y.load("vectorY.txt");
       Z.load("vectorZ.txt");
       Q.load("vectorQ.txt");
       cout <<"Vector u1:" << "\n" << X <<endl;
       cout <<"Vector u2:" << "\n" << Y <<endl;
       cout <<"Vector u3:" << "\n" << Z <<endl;

       // Create standard matrix for rotation about x-axis
       float matrixAx[N][N] = {};
       // Create standard matrix for rotation about y-axis
       float matrixAy[N][N] = {};
       // Create standard matrix for rotation about y'-axis
       float matrixAy2[N][N] = {};
       // Create standard matrix for rotation about z-axis
       float matrixAz[N][N] = {};
       for (int i = 0; i < N; ++i)
```

```
{
        for(int j = 0; j<N; ++j)
        {
                if (i == j)
                {
                        matrixAx[i][j] = 1;
                        matrixAy[i][j] = 1;
                        matrixAz[i][j] = 1;
                }
                else
                {
                        matrixAx[i][j] = 0;
                        matrixAy[i][j] = 0;
                        matrixAz[i][j] = 0;
                }
        }
}
matrixAx[1][1] = cos(theta*DEGTORAD);
matrixAx[1][2] = -sin(theta*DEGTORAD);
matrixAx[2][1] = sin(theta*DEGTORAD);
matrixAx[2][2] = cos(theta*DEGTORAD);

matrixAy[0][0] = cos(theta*DEGTORAD);
matrixAy[0][2] = sin(theta*DEGTORAD);
matrixAy[2][0] = -sin(theta*DEGTORAD);
matrixAy[2][2] = cos(theta*DEGTORAD);

matrixAz[0][0] = cos(theta*DEGTORAD);
matrixAz[0][1] = -sin(theta*DEGTORAD);
matrixAz[1][0] = sin(theta*DEGTORAD);
matrixAz[1][1] = cos(theta*DEGTORAD);

// y' = (a,b,c) = (- sin theta, cos theta, 0 )
float a = -sin(theta*DEGTORAD);
float b = cos(theta*DEGTORAD);
float c = 0;
// Standard matrix for a counterclockwise rotation through an angle
    phi about an arbitrary unit vector u=(a,b,c)
matrixAy2[0][0] = pow(a,2)*(1-cos(phi*DEGTORAD)) + cos(phi*DEGTORAD
    );
matrixAy2[0][1] = a*b*(1-cos(phi*DEGTORAD)) - c*sin(phi*DEGTORAD);
matrixAy2[0][2] = a*c*(1-cos(phi*DEGTORAD)) + b*sin(phi*DEGTORAD);
matrixAy2[1][0] = a*b*(1-cos(phi*DEGTORAD)) + c*sin(phi*DEGTORAD);
matrixAy2[1][1] = pow(b,2)*(1-cos(phi*DEGTORAD)) + cos(phi*DEGTORAD
    );
matrixAy2[1][2] = b*c*(1-cos(phi*DEGTORAD)) - a*sin(phi*DEGTORAD);
matrixAy2[2][0] = a*c*(1-cos(phi*DEGTORAD)) - b*sin(phi*DEGTORAD);
matrixAy2[2][1] = b*c*(1-cos(phi*DEGTORAD)) + a*sin(phi*DEGTORAD);
```

```
matrixAy2[2][2] = pow(c,2)*(1-cos(phi*DEGTORAD)) + cos(phi*DEGTORAD
    );

arma::mat ArmaAx(N,N,fill::zeros); // Declare matrix ArmaAx of size
    N X N with Armadillo
arma::mat ArmaAy(N,N,fill::zeros); // Declare matrix ArmaAy of size
    N X N with Armadillo
arma::mat ArmaAy2(N,N,fill::zeros); // Declare matrix ArmaAy of size
     N X N with Armadillo
arma::mat ArmaAz(N,N,fill::zeros); // Declare matrix ArmaAz of size
    N X N with Armadillo
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              ArmaAx[i+j*N] = matrixAx[i][j] ; // Standard Matrix
                  for rotation about x-axis
              ArmaAy[i+j*N] = matrixAy[i][j] ; // Standard Matrix
                  for rotation about y-axis
              ArmaAy2[i+j*N] = matrixAy2[i][j] ; // Standard Matrix
                  for rotation about y'-axis
              ArmaAz[i+j*N] = matrixAz[i][j] ; // Standard Matrix
                  for rotation about z-axis
       }
}
// Standard Matrix for rotation about z-axis = Transition matrix
    from B' to B
arma::mat ArmaAz_Transpose = ArmaAz.t();
cout <<"P_{z} (Transition matrix from B' to B):" << "\n" << ArmaAz
    <<endl;
cout <<"P_{z}^{T} (Transition matrix from B to B'):" << "\n" <<
    ArmaAz_Transpose <<endl;

arma::mat Xrotatez = ArmaAz*X ;
arma::mat Yrotatez = ArmaAz*Y ;
cout <<"The rotated vector u1 about z-axis:" << "\n" << Xrotatez <<
    endl;
cout <<"The rotated vector u2 about z-axis:" << "\n" << Yrotatez <<
    endl;

arma::mat ArmaAy2_Transpose = ArmaAy2.t();
cout <<"P_{y'} (Transition matrix from B'' to B'):" << "\n" <<
    ArmaAy2 <<endl;
cout <<"P_{y'}^{T} (Transition matrix from B' to B''):" << "\n" <<
    ArmaAy2_Transpose <<endl;

arma::mat Xrotatey2 = ArmaAy2*Xrotatez ;
arma::mat Zrotatey2 = ArmaAy2*Z ;
```

```
cout <<"The rotated vector u1' about y'-axis:" << "\n" << Xrotatey2
     <<endl;
cout <<"The rotated vector u3' about y'-axis:" << "\n" << Zrotatey2
     <<endl;

arma::mat PTQ;
PTQ = ArmaAz_Transpose*Q;
arma::mat PTQ2;
PTQ2 = ArmaAy2_Transpose*ArmaAz_Transpose*Q;

cout <<"Q relative to the xyz-system:" << endl;
cout <<"(x, y,z) = " << "( " << Q[0] << ", " << Q[1] << ", " << Q[2]
     << " )"<<endl;
cout <<"Q relative to the x'y'z'-system:" << endl;
cout <<"(x', y', z') = " << "( " << PTQ[0] << ", " << PTQ[1] << ", "
     << PTQ[2] << " )"<<endl;
cout <<"Q relative to the x''y''z''-system:" << endl;
cout <<"(x'', y'', z'') = " << "( " << PTQ2[0] << ", " << PTQ2[1] <<
     ", " << PTQ2[2] << " )"<<endl;

// We use a separate container for each column, like so:
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
std::vector<double> pts_E_dz;
std::vector<double> pts_F_x;
std::vector<double> pts_F_y;
std::vector<double> pts_F_z;
std::vector<double> pts_F_dx;
std::vector<double> pts_F_dy;
std::vector<double> pts_F_dz;
std::vector<double> pts_G_x;
std::vector<double> pts_G_y;
std::vector<double> pts_G_z;
std::vector<double> pts_G_dx;
std::vector<double> pts_G_dy;
std::vector<double> pts_G_dz;
std::vector<double> pts_H_x;
std::vector<double> pts_H_y;
std::vector<double> pts_H_z;
```

```
            std::vector<double> pts_H_dx;
            std::vector<double> pts_H_dy;
            std::vector<double> pts_H_dz;
            std::vector<double> pts_I_x;
            std::vector<double> pts_I_y;
            std::vector<double> pts_I_z;
            std::vector<double> pts_I_dx;
            std::vector<double> pts_I_dy;
            std::vector<double> pts_I_dz;
            std::vector<double> pts_J_x;
            std::vector<double> pts_J_y;
            std::vector<double> pts_J_z;
            std::vector<double> pts_J_dx;
            std::vector<double> pts_J_dy;
            std::vector<double> pts_J_dz;
            float o = 0;

            // Create a vector u1
            pts_D_x .push_back(o);
            pts_D_y .push_back(o);
            pts_D_z .push_back(o);
            pts_D_dx.push_back(X[0]);
            pts_D_dy.push_back(X[1]);
            pts_D_dz.push_back(X[2]);
            // Create a vector u2
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(Y[0]);
            pts_E_dy.push_back(Y[1]);
            pts_E_dz.push_back(Y[2]);
            // Create a vector u3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(Z[0]);
            pts_F_dy.push_back(Z[1]);
            pts_F_dz.push_back(Z[2]);
            // Rotated vector u1 about z—axis
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(Xrotatez[0]);
            pts_G_dy.push_back(Xrotatez[1]);
            pts_G_dz.push_back(Xrotatez[2]);
            // Rotated vector u2 about z—axis
            pts_H_x .push_back(o);
            pts_H_y .push_back(o);
```

```
            pts_H_z .push_back(o);
            pts_H_dx.push_back(Yrotatez[0]);
            pts_H_dy.push_back(Yrotatez[1]);
            pts_H_dz.push_back(Yrotatez[2]);
            // Rotated vector u1' about y'—axis
            pts_I_x .push_back(o);
            pts_I_y .push_back(o);
            pts_I_z .push_back(o);
            pts_I_dx.push_back(Xrotatey2[0]);
            pts_I_dy.push_back(Xrotatey2[1]);
            pts_I_dz.push_back(Xrotatey2[2]);
            // Rotated vector u3' about y'—axis
            pts_J_x .push_back(o);
            pts_J_y .push_back(o);
            pts_J_z .push_back(o);
            pts_J_dx.push_back(Zrotatey2[0]);
            pts_J_dy.push_back(Zrotatey2[1]);
            pts_J_dz.push_back(Zrotatey2[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [—0.5:1]\nset yrange [—0.5:1]\nset zrange
                [—1:1]\n";
            gp << "set xlabel 'x—axis'\n set ylabel 'y—axis'\n set zlabel 'z—
                axis'\n";
            gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
            // '—' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            // If you want to plot xy—plane (u,v,0) use this: splot '++' using
                1:2:(0) with lines
            gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray80'
                title 'xy—plane',\
            '—' with vectors lw 1 lc rgb 'dark—green' title '{/:Bold {u}_
                {1}}',\
            '—' with vectors lw 1 lc rgb 'dark—turquoise' title '{/:Bold {u}_
                {2}}',\
            '—' with vectors lw 1 lc rgb 'dark—red' title '{/:Bold {u}_{3}}',\
            '—' with vectors lt 4 lw 3 lc rgb 'dark—green' title 'rotated {/:
                Bold {u}_{1}}',\
            '—' with vectors lt 5 lw 3 lc rgb 'dark—turquoise' title 'rotated
                {/:Bold {u}_{2}}',\
            '—' with vectors lw 5 lc rgb 'dark—green' title 'rotated {/:Bold {u
                `}_{1}}',\
            '—' with vectors lw 5 lc rgb 'dark—red' title 'rotated {/:Bold {u`}
                _{3}}',\
            'vectorQ.dat' with points pt 7 ps 2 title 'Q'\n";
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
```

```
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
            gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx,
                pts_H_dy, pts_H_dz));
            gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_z, pts_I_dx,
                pts_I_dy, pts_I_dz));
            gp.send1d(boost::make_tuple(pts_J_x, pts_J_y, pts_J_z, pts_J_dx,
                pts_J_dy, pts_J_dz));
    }
```

**C++ Code 206:** *main.cpp "Rotation of axes in 3 dimensional space around z-axis then around y′-axis"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We define the **matrixAy2** here to be the standard matrix for a counterclockwise rotation through an angle $\phi$ about an axis in $\mathbb{R}^3$, which is determined by the unit vector $\boldsymbol{y'} = (a, b, c) = (-\sin\ \theta, \cos\ \theta, 0)$ that has its′ initial point at the origin.

```
            ...
            ...

            // y' = (a,b,c) = (— sin theta, cos theta, 0 )
            float a = —sin(theta*DEGTORAD);
            float b = cos(theta*DEGTORAD);
            float c = 0;
            // Standard matrix for a counterclockwise rotation through an
                angle phi about an arbitrary unit vector u=(a,b,c)
            matrixAy2[0][0] = pow(a,2)*(1—cos(phi*DEGTORAD)) + cos(phi*
                DEGTORAD);
            matrixAy2[0][1] = a*b*(1—cos(phi*DEGTORAD)) — c*sin(phi*
                DEGTORAD);
            matrixAy2[0][2] = a*c*(1—cos(phi*DEGTORAD)) + b*sin(phi*
                DEGTORAD);
            matrixAy2[1][0] = a*b*(1—cos(phi*DEGTORAD)) + c*sin(phi*
                DEGTORAD);
            matrixAy2[1][1] = pow(b,2)*(1—cos(phi*DEGTORAD)) + cos(phi*
                DEGTORAD);
```

```
            matrixAy2[1][2] = b*c*(1-cos(phi*DEGTORAD)) - a*sin(phi*
                DEGTORAD);
            matrixAy2[2][0] = a*c*(1-cos(phi*DEGTORAD)) - b*sin(phi*
                DEGTORAD);
            matrixAy2[2][1] = b*c*(1-cos(phi*DEGTORAD)) + a*sin(phi*
                DEGTORAD);
            matrixAy2[2][2] = pow(c,2)*(1-cos(phi*DEGTORAD)) + cos(phi*
                DEGTORAD);

            ...
            ...
```

```
P_{z} (Transition matrix from B' to B):
   0.5000  -0.8660        0
   0.8660   0.5000        0
        0        0   1.0000

P_{z}^{T} (Transition matrix from B to B'):
   0.5000   0.8660        0
  -0.8660   0.5000        0
        0        0   1.0000

The rotated vector u1 about z-axis:
   0.5000
   0.8660
        0

The rotated vector u2 about z-axis:
  -0.8660
   0.5000
        0

P_{y'} (Transition matrix from B'' to B'):
   0.9268  -0.1268   0.3536
  -0.1268   0.7803   0.6124
  -0.3536  -0.6124   0.7071

P_{y'}^{T} (Transition matrix from B' to B''):
   0.9268  -0.1268  -0.3536
  -0.1268   0.7803  -0.6124
   0.3536   0.6124   0.7071

The rotated vector u1' about y'-axis:
   0.3536
   0.6124
  -0.7071

The rotated vector u3' about y'-axis:
   0.3536
   0.6124
   0.7071

Q relative to the xyz-system:
(x, y,z) = ( 0.6, 0.2, 0.5 )
Q relative to the x'y'z'-system:
(x', y', z') = ( 0.473205, -0.419615, 0.5 )
Q relative to the x''y''z''-system:
(x'', y'', z'') = ( 0.314997, -0.69364, 0.263896 )
```

**Figure 23.173:** *The computation to obtain the new basis $\{\boldsymbol{u}_1'', \boldsymbol{u}_2'', \boldsymbol{u}_3''\}$ from rotating the old basis counterclockwise around the z-axis (from positive z-axis looking toward the origin) through an angle $\theta = \frac{\pi}{3}$, then rotating counterclockwise again around the $y'$-axis (from positive $y'$-axis looking toward the origin) through an angle $\phi = \frac{\pi}{4}$ and obtain the new coordinates of the point Q relative to the $x''y''z''$-system. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation of Axes About z-Axis then y'-axis/main.cpp**).*

**Figure 23.174:** *The plot to obtain the new basis* $\{\boldsymbol{u}_1'', \boldsymbol{u}_2'', \boldsymbol{u}_3''\}$ *from rotating the old basis counterclockwise around the z-axis (from positive z-axis looking toward the origin) through an angle* $\theta = \frac{\pi}{3}$*, then rotating counterclockwise again around the* $y'$*-axis (from positive* $y'$*-axis looking toward the origin) through an angle* $\phi = \frac{\pi}{4}$*. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Rotation of Axes About z-Axis then y'-axis/main.cpp**).*

## iv. C++ Plot and Computation: Orthogonally Diagonalizing a Symmetric Matrix

Find an orthogonal matrix $P$ that diagonalizes

$$A = \begin{bmatrix} 4 & 2 & 2 \\ 2 & 4 & 2 \\ 2 & 2 & 4 \end{bmatrix}$$

**Solution:**

The characteristic equation of $A$ is

$$\det(\lambda I - A) = \det \begin{bmatrix} \lambda - 4 & -2 & -2 \\ -2 & \lambda - 4 & -2 \\ -2 & -2 & \lambda - 4 \end{bmatrix} = (\lambda - 2)^2(\lambda - 8) = 0$$

Thus, the distrinct eigenvalues of $A$ are

$$\lambda = 2, \quad \lambda = 8$$

Thus for $\lambda = 2$, we will have

$$\lambda I - A = \begin{bmatrix} -2 & -2 & -2 \\ -2 & -2 & -2 \\ -2 & -2 & -2 \end{bmatrix}$$

the general solution for $(\lambda I - A)\boldsymbol{x} = \mathbf{0}$ is

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ x_1 &= -x_2 - x_3 \\ x_2 &= s \\ x_3 &= t \end{aligned}$$

in vector form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = s \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

so

$$\boldsymbol{u}_1 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

form a basis for the eigenspace corresponding to $\lambda = 2$. Applying the Gram-Schmidt process to $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ yields the following orthonormal eigenvectors:

$$\boldsymbol{v}_1 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} -\frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \end{bmatrix}$$

The eigenspace corresponding to $\lambda = 8$ has

$$\boldsymbol{u}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

as a basis. Applying the Gram-Schmidt process to $\{\boldsymbol{u}_3\}$ yields

$$\boldsymbol{v}_3 = \begin{bmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$$

Finally, using $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$ as column vectors, we obtain

$$P = \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{3}} \end{bmatrix}$$

which orthogonally diagonalizes $A$. Finally we will have

$$P^T AP = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} 4 & 2 & 2 \\ 2 & 4 & 2 \\ 2 & 2 & 4 \end{bmatrix} \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{3}} \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 8 \end{bmatrix}$$

The C++ code is using**Armadillo** for the computation part and **Gnuplot** to do the plot of the orthonormal eigenvectors. **Armadillo**' computation for eigenvectors is already in form of orthonormal eigenvectors, so there is no need to do Gram-Schmidt process manually.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(3,3,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix

       cout << "Eigenvalues:" << endl;
```

```
int n = 3;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

cout << endl;
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
mat A3 = I*real(eigval[2]) - A;
cout <<"Matrix l_1I - A:" << "\n" << A1 <<endl;
cout <<"Matrix l_2I - A:" << "\n" << A2 <<endl;

cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
cout << endl;

cout << "Orthonormal Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

// Save each orthonormal eigenvectors as a vector with Armadillo
vec v1 = real(eigvec.col(0));
vec v2 = real(eigvec.col(1));
vec v3 = real(eigvec.col(2));

//cout <<"v1 . v2:" << dot(v1,v2) <<endl;
//cout <<"v2 . v3:" << dot(v2,v3) <<endl;

mat P = real(eigvec);
mat P_transpose = P.t();
mat D = P_transpose*A*P;

cout <<"Matrix P (which orthogonally diagonalizes A):" << "\n" << P
    <<endl;

cout <<"Matrix P^{T} * A * P:" << "\n" << D <<endl;


Gnuplot gp;

// We use a separate container for each column, like so:
```

```
            std::vector<double> pts_D_x;
            std::vector<double> pts_D_y;
            std::vector<double> pts_D_z;
            std::vector<double> pts_D_dx;
            std::vector<double> pts_D_dy;
            std::vector<double> pts_D_dz;
            std::vector<double> pts_E_x;
            std::vector<double> pts_E_y;
            std::vector<double> pts_E_z;
            std::vector<double> pts_E_dx;
            std::vector<double> pts_E_dy;
            std::vector<double> pts_E_dz;
            std::vector<double> pts_F_x;
            std::vector<double> pts_F_y;
            std::vector<double> pts_F_z;
            std::vector<double> pts_F_dx;
            std::vector<double> pts_F_dy;
            std::vector<double> pts_F_dz;

            float o = 0;

            // Create the orthonormal eigenvector v1
            pts_D_x .push_back(o);
            pts_D_y .push_back(o);
            pts_D_z .push_back(o);
            pts_D_dx.push_back(v1[0]);
            pts_D_dy.push_back(v1[1]);
            pts_D_dz.push_back(v1[2]);
            // Create the orthonormal eigenvector v2
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(v2[0]);
            pts_E_dy.push_back(v2[1]);
            pts_E_dz.push_back(v2[2]);
            // Create the orthonormal eigenvector v3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(v3[0]);
            pts_F_dy.push_back(v3[1]);
            pts_F_dz.push_back(v3[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [-1:1]\nset yrange [-1:1]\nset zrange [-1:1]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
```

```
            axis'\n";
        gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray80'
            title 'xy-plane',\
        '-' with vectors lw 2 lc rgb 'dark-green' title '{/:Bold {v}_
            {1}}',\
        '-' with vectors lw 2 lc rgb 'dark-turquoise' title '{/:Bold {v}_
            {2}}',\
        '-' with vectors lw 2 lc rgb 'dark-red' title '{/:Bold {v}_{3}}'\n"
            ;
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
            pts_D_dy, pts_D_dz));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
            pts_E_dy, pts_E_dz));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
            pts_F_dy, pts_F_dz));

        return 0;
    }
```

**C++ Code 207:** *main.cpp "Orthogonally diagonalizing a symmetric matrix"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.175:** *The plot of orthonormal vectors $\{v_1, v_2, v_3\}$ from matrix $A$.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Eigenvectors and Compute Orthogonal Matrix P that Diagonalizes A with Armadillo/main.cpp).***

```
Matrix A:
   4.0000   2.0000   2.0000
   2.0000   4.0000   2.0000
   2.0000   2.0000   4.0000

Eigenvalues:
λ_1 = 2 + 0i
λ_2 = 8 + 0i
λ_3 = 2 + 0i

Matrix λ_1I - A:
  -2.0000  -2.0000  -2.0000
  -2.0000  -2.0000  -2.0000
  -2.0000  -2.0000  -2.0000

Matrix λ_2I - A:
   4.0000  -2.0000  -2.0000
  -2.0000   4.0000  -2.0000
  -2.0000  -2.0000   4.0000


Eigenspaces for λ_1: 2
Eigenspaces for λ_2: 1

Orthonormal Eigenvectors:
x[λ_1] :
   -0.8165
   0.4082
   0.4082

x[λ_2] :
    0.5774
   0.5774
   0.5774

x[λ_3] :
   -0.3244
  -0.4867
   0.8111

Matrix P (which orthogonally diagonalizes A):
  -0.8165   0.5774  -0.3244
   0.4082   0.5774  -0.4867
   0.4082   0.5774   0.8111

Matrix P^{T} * A * P:
   2.0000e+00   7.2164e-16   7.9472e-01
   6.6613e-16   8.0000e+00  -4.4409e-16
   7.9472e-01  -3.3307e-16   2.0000e+00
```

**Figure 23.176:** *The computation to obtain orthonormal vectors $\{v_1, v_2, v_3\}$ from matrix A and the matrix P that orthogonally diagonalizes A. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Eigenvectors and Compute Orthogonal Matrix P that Diagonalizes A with Armadillo/main.cpp**).*

## v. C++ Plot and Computation: A Geometric Interpretation of a Spectral Decomposition

The matrix

$$A = \begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix}$$

has eigenvalues $\lambda_1 = -3$ and $\lambda_2 = 2$ with corresponding eigenvectors

$$\boldsymbol{x}_1 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad \boldsymbol{x}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Normalizing these basis vectors yields

$$\boldsymbol{u}_1 = \frac{\boldsymbol{x}_1}{||\boldsymbol{x}_1||} = \begin{bmatrix} \frac{1}{\sqrt{5}} \\ -\frac{2}{\sqrt{5}} \end{bmatrix}, \quad \boldsymbol{x}_2 = \frac{\boldsymbol{x}_2}{||\boldsymbol{x}_2||} = \begin{bmatrix} \frac{2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{bmatrix}$$

so a spectral decomposition of $A$ is

$$\begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} = \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T = (-3) \begin{bmatrix} \frac{1}{\sqrt{5}} \\ -\frac{2}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{5}} \end{bmatrix} + (2) \begin{bmatrix} \frac{2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \end{bmatrix}$$

$$\lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T = (-3) \begin{bmatrix} \frac{1}{5} & -\frac{2}{5} \\ -\frac{2}{5} & \frac{4}{5} \end{bmatrix} + (2) \begin{bmatrix} \frac{4}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{1}{5} \end{bmatrix}$$

the $2 \times 2$ matrices resulted from the spectral decomposition are the standard matrices for the orthogonal projections onto the eigenspaces corresponding to $\lambda_1 = -3$ and $\lambda_2 = 2$, respectively.

Now, let us see what this spectral decomposition tells us about the image of the vector $\boldsymbol{x} = (1, 1)$ under the multiplication by $A$. Writing $\boldsymbol{x}$ in column form, it follows that

$$A\boldsymbol{x} = \begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

and from spectral decomposition of $A$

$$A\boldsymbol{x} = \begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = (-3) \begin{bmatrix} \frac{1}{5} & -\frac{2}{5} \\ -\frac{2}{5} & \frac{4}{5} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + (2) \begin{bmatrix} \frac{4}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{1}{5} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$A\boldsymbol{x} = (-3) \begin{bmatrix} -\frac{1}{5} \\ \frac{2}{5} \end{bmatrix} + (2) \begin{bmatrix} \frac{6}{5} \\ \frac{3}{5} \end{bmatrix} = \begin{bmatrix} \frac{3}{5} \\ -\frac{6}{5} \end{bmatrix} + \begin{bmatrix} \frac{12}{5} \\ \frac{6}{5} \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

In short:

1. The spectral decomposition of $A$ is

   $$A = \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T$$

   with $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are the orthonormal eigenvectors for $A$ and the eigenvalues for $A$ are $\lambda_1$ and $\lambda_2$.

2. The image of the vector $\boldsymbol{x} = (1, 1)$ under multiplication by $A$ is

   $$A\boldsymbol{x} = \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T \boldsymbol{x} + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T \boldsymbol{x}$$

   we write $\boldsymbol{x}$ in column form.

3. When we project $\boldsymbol{x} = (1, 1)$ onto the eigenspaces corresponding to $\lambda_1 = -3$ it means that

$$\boldsymbol{u}_1 \boldsymbol{u}_1^T \boldsymbol{x}$$

is a scalar multiplication of the eigenvector $\boldsymbol{u}_1$.

When we project $\boldsymbol{x} = (1, 1)$ onto the eigenspaces corresponding to $\lambda_2 = 2$ it means that

$$\boldsymbol{u}_2 \boldsymbol{u}_2^T \boldsymbol{x}$$

is a scalar multiplication of the eigenvector $\boldsymbol{u}_2$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
        mat A;
        A.load("matrixA.txt");
        vec X;
        X.load("vectorX.txt");
        cx_mat eigvec;
        cx_vec eigval;

        mat I(2,2,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
```

```
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
mat A3 = I*real(eigval[2]) - A;
cout <<"Matrix l_1I - A:" << "\n" << A1 <<endl;
cout <<"Matrix l_2I - A:" << "\n" << A2 <<endl;

cout << endl;
cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
cout << endl;

cout << "Orthonormal Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

// Save each orthonormal eigenvectors as a vector with Armadillo
vec v1 = real(eigvec.col(0));
vec v2 = real(eigvec.col(1));

//cout <<"v1 . v2:" << dot(v1,v2) <<endl;

// Spectral Decomposition of A for vector x=(1,1)
vec spectral_decomp = A*X;
vec spectral_decomp1 = v1*v1.t()*X;
vec spectral_decomp2 = v2*v2.t()*X;
vec scaled_spectral_decomp1 = v1*v1.t()*X*real(eigval[0]);
vec scaled_spectral_decomp2 = v2*v2.t()*X*real(eigval[1]);

mat P = real(eigvec);
mat P_transpose = P.t();
mat D = P_transpose*A*P;

cout <<"Matrix P (which orthogonally diagonalizes A):" << "\n" << P
    <<endl;

cout <<"Matrix P^{T} * A * P:" << "\n" << D <<endl;

cout <<"Projection of vector x=(1,1) onto eigenspaces corresponding
    to l_1 = "<< real(eigval[0])<< ":" << "\n" << spectral_decomp1
    <<endl;
cout <<"Projection of vector x=(1,1) onto eigenspaces corresponding
    to l_2 = "<< real(eigval[1])<< ":" << "\n" << spectral_decomp2
    <<endl;

cout <<"Scaled spectral decomposition of x=(1,1) onto eigenspaces
```

```
        corresponding to l_1 = "<< real(eigval[0])<< ":" << "\n" <<
        scaled_spectral_decomp1 <<endl;
    cout <<"Scaled spectral decomposition of x=(1,1) onto eigenspaces
        corresponding to l_2 = "<< real(eigval[1])<< ":" << "\n" <<
        scaled_spectral_decomp2 <<endl;

    Gnuplot gp;

    // We use a separate container for each column, like so:
    std::vector<double> pts_D_x;
    std::vector<double> pts_D_y;
    std::vector<double> pts_D_dx;
    std::vector<double> pts_D_dy;
    std::vector<double> pts_E_x;
    std::vector<double> pts_E_y;
    std::vector<double> pts_E_dx;
    std::vector<double> pts_E_dy;
    std::vector<double> pts_F_x;
    std::vector<double> pts_F_y;
    std::vector<double> pts_F_dx;
    std::vector<double> pts_F_dy;
    std::vector<double> pts_G_x;
    std::vector<double> pts_G_y;
    std::vector<double> pts_G_dx;
    std::vector<double> pts_G_dy;
    std::vector<double> pts_H_x;
    std::vector<double> pts_H_y;
    std::vector<double> pts_H_dx;
    std::vector<double> pts_H_dy;
    std::vector<double> pts_I_x;
    std::vector<double> pts_I_y;
    std::vector<double> pts_I_dx;
    std::vector<double> pts_I_dy;
    std::vector<double> pts_J_x;
    std::vector<double> pts_J_y;
    std::vector<double> pts_J_dx;
    std::vector<double> pts_J_dy;
    std::vector<double> pts_K_x;
    std::vector<double> pts_K_y;
    std::vector<double> pts_K_dx;
    std::vector<double> pts_K_dy;

    float o = 0;

    // Create the orthonormal eigenvector v1
    pts_D_x .push_back(o);
    pts_D_y .push_back(o);
    pts_D_dx.push_back(v1[0]);
```

```
        pts_D_dy.push_back(v1[1]);
        // Create the orthonormal eigenvector v2
        pts_E_x .push_back(o);
        pts_E_y .push_back(o);
        pts_E_dx.push_back(v2[0]);
        pts_E_dy.push_back(v2[1]);
        // Create the vector x = (1,1)
        pts_F_x .push_back(o);
        pts_F_y .push_back(o);
        pts_F_dx.push_back(X[0]);
        pts_F_dy.push_back(X[1]);
        // Projection of vector x=(1,1) onto eigenspaces corresponding to
            l_1
        pts_G_x .push_back(o);
        pts_G_y .push_back(o);
        pts_G_dx.push_back(spectral_decomp1[0]);
        pts_G_dy.push_back(spectral_decomp1[1]);
        // Projection of vector x=(1,1) onto eigenspaces corresponding to
            l_2
        pts_H_x .push_back(o);
        pts_H_y .push_back(o);
        pts_H_dx.push_back(spectral_decomp2[0]);
        pts_H_dy.push_back(spectral_decomp2[1]);
        // Projection of vector x=(1,1) onto eigenspaces corresponding to
            l_1 then scaled with l_1
        pts_I_x .push_back(o);
        pts_I_y .push_back(o);
        pts_I_dx.push_back(scaled_spectral_decomp1[0]);
        pts_I_dy.push_back(scaled_spectral_decomp1[1]);
        // Projection of vector x=(1,1) onto eigenspaces corresponding to
            l_2 then scaled with l_2
        pts_J_x .push_back(o);
        pts_J_y .push_back(o);
        pts_J_dx.push_back(scaled_spectral_decomp2[0]);
        pts_J_dy.push_back(scaled_spectral_decomp2[1]);
        // Image of vector x=(1,1) under multiplication by matrix A
        pts_K_x .push_back(o);
        pts_K_y .push_back(o);
        pts_K_dx.push_back(spectral_decomp[0]);
        pts_K_dy.push_back(spectral_decomp[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-1:4]\nset yrange [-1.5:2]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
```

```
        ///*
        gp << "plot '-' with vectors lw 1 lc rgb 'dark-red' title '{/:Bold
            {x}}',\
        '-' with vectors lw 1 lc rgb 'dark-green' title '{/:Bold {spectral
            decomposition x on l}_{1}}',\
        '-' with vectors lw 1 lc rgb 'dark-turquoise' title '{/:Bold {
            spectral decomposition x on l}_{2}}',\
        '-' with vectors lw 2 lc rgb 'dark-green' dashtype 3 title '{/:Bold
             {scaled spectral decomposition x on l}_{1}}',\
        '-' with vectors lw 2 lc rgb 'dark-turquoise' dashtype 3 title '{/:
            Bold {scaled spectral decomposition x on l}_{2}}',\
        '-' with vectors lw 2 lc rgb 'dark-red' title 'A{/:Bold {x}}'\n";
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
        gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_dx, pts_G_dy));
        gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_dx, pts_H_dy));
        gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_dx, pts_I_dy));
        gp.send1d(boost::make_tuple(pts_J_x, pts_J_y, pts_J_dx, pts_J_dy));
        gp.send1d(boost::make_tuple(pts_K_x, pts_K_y, pts_K_dx, pts_K_dy));
        //*/

        // to plot the orthonormal eigenvectors and vector x only
        /*
        gp << "plot '-' with vectors lw 1 lc rgb 'black' title '{/:Bold {u}
            _{1}}',\
        '-' with vectors lw 1 lc rgb 'gray50' title '{/:Bold {u}_{2}}',\
        '-' with vectors lw 1 lc rgb 'dark-red' title '{/:Bold {x}}'\n";
        gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_dx, pts_D_dy));
        gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_dx, pts_E_dy));
        gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_dx, pts_F_dy));
        */

        return 0;
    }
```

**C++ Code 208:** *main.cpp "A geometric interpretation of a spectral decomposition"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- To compute the spectral decomposition of $A$ to see the image of the vector $\boldsymbol{x} = (1,1)$ :

$$A\boldsymbol{x}$$

$$\boldsymbol{u}_1\boldsymbol{u}_1^T\boldsymbol{x}$$

$$\boldsymbol{u}_2\boldsymbol{u}_2^T\boldsymbol{x}$$

then scaling by the corresponding eigenvalue:

$$\lambda_1\boldsymbol{u}_1\boldsymbol{u}_1^T\boldsymbol{x}$$

$$\lambda_2\boldsymbol{u}_2\boldsymbol{u}_2^T\boldsymbol{x}$$

we save these 5 formulas with **arma::vec** or **vec** if we already use namespace **arma**. These 5 vectors result can then be plotted with **gnuplot**.

```
...

vec spectral_decomp = A*X;
vec spectral_decomp1 = v1*v1.t()*X;
vec spectral_decomp2 = v2*v2.t()*X;
vec scaled_spectral_decomp1 = v1*v1.t()*X*real(eigval[0]);
vec scaled_spectral_decomp2 = v2*v2.t()*X*real(eigval[1]);

...
```

**Figure 23.177:** *The plot of the spectral decomposition of $A$ toward vector $\boldsymbol{x} = (1,1)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Geometric Interpretation of Spectral Decomposition with Armadillo/main.cpp**).*

```
Eigenvalues:
λ_1 = 2 + 0i
λ_2 = -3 + 0i

Matrix λ_1I - A:
   1.0000  -2.0000
  -2.0000   4.0000

Matrix λ_2I - A:
  -4.0000  -2.0000
  -2.0000  -1.0000


Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Orthonormal Eigenvectors:
x[λ_1] :
    0.8944
   0.4472

x[λ_2] :
   -0.4472
   0.8944

Matrix P (which orthogonally diagonalizes A):
   0.8944  -0.4472
   0.4472   0.8944

Matrix P^{T} * A * P:
   2.0000        0
        0  -3.0000

Projection of vector x=(1,1) onto eigenspaces corresponding to λ_1 = 2:
   1.2000
   0.6000

Projection of vector x=(1,1) onto eigenspaces corresponding to λ_2 = -3:
  -0.2000
   0.4000

Scaled spectral decomposition of x=(1,1) onto eigenspaces corresponding to λ_1 = 2:
   2.4000
   1.2000

Scaled spectral decomposition of x=(1,1) onto eigenspaces corresponding to λ_2 = -3:
   0.6000
  -1.2000
```

**Figure 23.178:** *The computation to obtain the image of the vector* $\boldsymbol{x} = (1,1)$ *under multiplication of* $A$ *and the spectral decomposition of* $A$ *toward* $\boldsymbol{x}$. *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Geometric Interpretation of Spectral Decomposition with Armadillo/main.cpp**).*

## vi. C++ Plot and Computation: Determine Symmetric Matrix from Known Eigenvalues and Eigenvectors

Is there a $3 \times 3$ symmetric matrix with eigenvalues $\lambda_1 = -1, \lambda_2 = 3, \lambda_3 = 7$ and corresponding eigenvectors $\boldsymbol{v}_1 = (0, 1, -1), \boldsymbol{v}_2 = (1, 0, 0), \boldsymbol{v}_3 = (1, 1, 1)$?

**Solution:**

First, we will perform Gram-Schmidt process to convert eigenvectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ into orthogonal basis / eigenvectors $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$

$$\begin{aligned}
\boldsymbol{u}_1 &= \boldsymbol{v}_1 \\
\boldsymbol{u}_2 &= \boldsymbol{v}_2 - \frac{< \boldsymbol{v}_2, \boldsymbol{u}_1 >}{||\boldsymbol{u}_1||^2}\boldsymbol{u}_1 \\
\boldsymbol{u}_3 &= \boldsymbol{v}_2 - \frac{< \boldsymbol{v}_3, \boldsymbol{u}_1 >}{||\boldsymbol{u}_1||^2}\boldsymbol{u}_1 - \frac{< \boldsymbol{v}_3, \boldsymbol{u}_2 >}{||\boldsymbol{u}_2||^2}\boldsymbol{u}_2
\end{aligned}$$

Then we will normalize the orthogonal eigenvectors $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$

$$\boldsymbol{u}_1 = \frac{\boldsymbol{u}_1}{||\boldsymbol{u}_1||} = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \quad \boldsymbol{u}_2 = \frac{\boldsymbol{u}_2}{||\boldsymbol{u}_2||} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{u}_3 = \frac{\boldsymbol{u}_3}{||\boldsymbol{u}_3||} = \begin{bmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$$

We will now use the spectral decomposition formula

$$A = \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T + \lambda_3 \boldsymbol{u}_3 \boldsymbol{u}_3^T$$

$$A = (-1) \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} + (3) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} + (7) \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 4 \\ 0 & 4 & 3 \end{bmatrix}$$

The matrix $A$ obtained above is a symmetric matrix, thus we know that there is a symmetric matrix for the given eigenvalues $\lambda_1 = -1, \lambda_2 = 3, \lambda_3 = 7$ and corresponding eigenvectors $\boldsymbol{v}_1 = (0, 1, -1), \boldsymbol{v}_2 = (1, 0, 0), \boldsymbol{v}_3 = (1, 1, 1)$.

The C++ code is using **Armadillo** for the matrix and vector computation, and **gnuplot** to do the plotting of the eigenvectors.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <boost/tuple/tuple.hpp>

#include "gnuplot−iostream.h"

using namespace std;
using namespace arma;

// Driver code
```

```
int main(int argc, char** argv)
{
        vec v1;
        vec v2;
        vec v3;
        v1.load("vector1.txt");
        v2.load("vector2.txt");
        v3.load("vector3.txt");
        cout <<"Eigenvector 1:" << "\n" << v1 <<endl;
        cout <<"Eigenvector 2:" << "\n" << v2 <<endl;
        cout <<"Eigenvector 3:" << "\n" << v3 <<endl;

        float l1 = -1;
        float l2 = 3;
        float l3 = 7;

        // Gram-Schmidt Process
        mat u1 = v1;

        mat u1v2 = (dot(u1,v2)/pow(norm(u1),2) )*u1;
        mat u2 = v2 - u1v2;

        mat u1v3 = (dot(u1,v3)/pow(norm(u1),2) )*u1;
        mat u2v3 = (dot(u2,v3)/pow(norm(u2),2) )*u2;
        mat u3 = v3 - u1v3 - u2v3;

        // Normalize the orthogonal basis vectors
        u1 = u1/norm(u1);
        u2 = u2/norm(u2);
        u3 = u3/norm(u3);

        mat u1_transpose = u1.t();
        mat u2_transpose = u2.t();
        mat u3_transpose = u3.t();

        cout <<"After Gram-Schmidt Process:" << "\n" <<endl;
        cout <<"Unit Eigenvector 1:" << "\n" << u1 <<endl;
        cout <<"Unit Eigenvector 2:" << "\n" << u2 <<endl;
        cout <<"Unit Eigenvector 3:" << "\n" << u3 <<endl;

        // Spectral decomposition computation
        mat A(3,3,fill::eye);
        A = l1*u1*u1_transpose + l2*u2*u2_transpose + l3*u3*u3_transpose;

        cout <<"Matrix A:" << "\n" << A <<endl;
        // save matrix as a text file
        A.save("matrixA.txt", raw_ascii);
```

```
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_z;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;
        std::vector<double> pts_D_dz;
        std::vector<double> pts_E_x;
        std::vector<double> pts_E_y;
        std::vector<double> pts_E_z;
        std::vector<double> pts_E_dx;
        std::vector<double> pts_E_dy;
        std::vector<double> pts_E_dz;
        std::vector<double> pts_F_x;
        std::vector<double> pts_F_y;
        std::vector<double> pts_F_z;
        std::vector<double> pts_F_dx;
        std::vector<double> pts_F_dy;
        std::vector<double> pts_F_dz;
        std::vector<double> pts_G_x;
        std::vector<double> pts_G_y;
        std::vector<double> pts_G_z;
        std::vector<double> pts_G_dx;
        std::vector<double> pts_G_dy;
        std::vector<double> pts_G_dz;
        std::vector<double> pts_H_x;
        std::vector<double> pts_H_y;
        std::vector<double> pts_H_z;
        std::vector<double> pts_H_dx;
        std::vector<double> pts_H_dy;
        std::vector<double> pts_H_dz;
        std::vector<double> pts_I_x;
        std::vector<double> pts_I_y;
        std::vector<double> pts_I_z;
        std::vector<double> pts_I_dx;
        std::vector<double> pts_I_dy;
        std::vector<double> pts_I_dz;

        float o = 0;

        // Create the eigenvector v1
        pts_D_x .push_back(o);
        pts_D_y .push_back(o);
        pts_D_z .push_back(o);
        pts_D_dx.push_back(v1[0]);
        pts_D_dy.push_back(v1[1]);
```

```
            pts_D_dz.push_back(v1[2]);
            // Create the eigenvector v2
            pts_E_x .push_back(o);
            pts_E_y .push_back(o);
            pts_E_z .push_back(o);
            pts_E_dx.push_back(v2[0]);
            pts_E_dy.push_back(v2[1]);
            pts_E_dz.push_back(v2[2]);
            // Create the eigenvector v3
            pts_F_x .push_back(o);
            pts_F_y .push_back(o);
            pts_F_z .push_back(o);
            pts_F_dx.push_back(v3[0]);
            pts_F_dy.push_back(v3[1]);
            pts_F_dz.push_back(v3[2]);

            // Create the orthonormal eigenvector u1
            pts_G_x .push_back(o);
            pts_G_y .push_back(o);
            pts_G_z .push_back(o);
            pts_G_dx.push_back(u1[0]);
            pts_G_dy.push_back(u1[1]);
            pts_G_dz.push_back(u1[2]);
            // Create the orthonormal eigenvector u2
            pts_H_x .push_back(o);
            pts_H_y .push_back(o);
            pts_H_z .push_back(o);
            pts_H_dx.push_back(u2[0]);
            pts_H_dy.push_back(u2[1]);
            pts_H_dz.push_back(u2[2]);
            // Create the orthonormal eigenvector u3
            pts_I_x .push_back(o);
            pts_I_y .push_back(o);
            pts_I_z .push_back(o);
            pts_I_dx.push_back(u3[0]);
            pts_I_dy.push_back(u3[1]);
            pts_I_dz.push_back(u3[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [-1:1]\nset yrange [-1:1]\nset zrange [-1:1]\n";
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";
            gp << "set view 80,90,1\n"; // pitch,yaw,zoom for y axis on front
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "splot '++' using 1:2:(0) with lines lw 2 lc rgb 'gray80'
```

```
                title 'xy-plane',\
            '-' with vectors lw 1 lc rgb 'dark-green' dashtype 4 title '{/:Bold
                 {v}_{1}}',\
            '-' with vectors lw 1 lc rgb 'dark-turquoise' dashtype 4 title '{/:
                Bold {v}_{2}}',\
            '-' with vectors lw 1 lc rgb 'dark-red' dashtype 4 title '{/:Bold {
                v}_{3}}',\
            '-' with vectors lw 2 lc rgb 'dark-green' title '{/:Bold {u}_
                {1}}',\
            '-' with vectors lw 2 lc rgb 'dark-turquoise' title '{/:Bold {u}_
                {2}}',\
            '-' with vectors lw 2 lc rgb 'dark-red' title '{/:Bold {u}_{3}}'\n"
                ;
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));
            gp.send1d(boost::make_tuple(pts_F_x, pts_F_y, pts_F_z, pts_F_dx,
                pts_F_dy, pts_F_dz));
            gp.send1d(boost::make_tuple(pts_G_x, pts_G_y, pts_G_z, pts_G_dx,
                pts_G_dy, pts_G_dz));
            gp.send1d(boost::make_tuple(pts_H_x, pts_H_y, pts_H_z, pts_H_dx,
                pts_H_dy, pts_H_dz));
            gp.send1d(boost::make_tuple(pts_I_x, pts_I_y, pts_I_z, pts_I_dx,
                pts_I_dy, pts_I_dz));

            return 0;
    }
```

**C++ Code 209:** *main.cpp "Determine symmetric matrix from known eigenvalues and eigenvectors"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.179:** *The plot of the eigenvectors* $\boldsymbol{v}_1 = (0, 1, -1), \boldsymbol{v}_2 = (1, 0, 0), \boldsymbol{v}_3 = (1, 1, 1)$ *and the converted eigenvectors with Gram-Schmidt process into orthonormal basis / eigenvectors* $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$. (***DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Eigenvectors and Determine Matrix A from Known Eigenvalues and Eigenvectors with Armadillo/main.cpp***).

```
Eigenvector 1:
        0
   1.0000
  -1.0000

Eigenvector 2:
   1.0000
        0
        0

Eigenvector 3:
   1.0000
   1.0000
   1.0000

After Gram-Schmidt Process:

Unit Eigenvector 1:
        0
   0.7071
  -0.7071

Unit Eigenvector 2:
   1.0000
        0
        0

Unit Eigenvector 3:
        0
   0.7071
   0.7071

Matrix A:
   3.0000        0        0
        0   3.0000   4.0000
        0   4.0000   3.0000
```

**Figure 23.180:** *The computation to obtain matrix A from known eigenvalues $\lambda_1 = -1, \lambda_2 = 3, \lambda_3 = 7$ and corresponding eigenvectors $\boldsymbol{v}_1 = (0, 1, -1), \boldsymbol{v}_2 = (1, 0, 0), \boldsymbol{v}_3 = (1, 1, 1)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Eigenvectors and Determine Matrix A from Known Eigenvalues and Eigenvectors with Armadillo/-main.cpp**).*

# XIII. Quadratic Forms

[DF*] Expressions of the form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n$$

occurred in our study of linear equations and linear systems. If $a_1, a_2, \ldots, a_n$ are treated as fixed constants, then this expression is a real-valued function of the $n$ variables $x_1, x_2, \ldots, x_n$ and is called a linear form on $\mathbb{R}^n$.

All variables in a linear form occur to the first power and there are no products of variables.

[DF*] Quadratic forms on $\mathbb{R}^n$ are functions of the form

$$a_1x_1^2 + a_2x_2^2 + \cdots + a_nx_n^2 + \sum a_kx_ix_j$$

The notation of $\sum a_kx_ix_j$ represents all possible terms of $a_kx_ix_j$ in which $x_i \neq x_j$.

The terms of the form $a_kx_ix_j$ are called cross product terms. It is common to combine the cross product terms involving $x_ix_j$ with those involving $x_jx_i$ to avoid duplication.

[DF*] A general quadratic form on $\mathbb{R}^2$ would typically be expressed as

$$a_1x_1^2 + a_2x_2^2 + 2a_3x_1x_2 \tag{23.192}$$

in matrix form it can be expressed as

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a_1 & a_3 \\ a_3 & a_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \boldsymbol{x}^T A\boldsymbol{x}$$

and a general quadratic form on $\mathbb{R}^3$ can be expressed as

$$a_1x_1^2 + a_2x_2^2 + a_3x_3^2 + 2a_4x_1x_2 + 2a_5x_1x_3 + 2a_6x_2x_3 \tag{23.193}$$

or, in matrix form

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} a_1 & a_4 & a_5 \\ a_4 & a_2 & a_6 \\ a_5 & a_6 & a_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \boldsymbol{x}^T A\boldsymbol{x}$$

The matrix $A$ in these formulas is symmetric, that its diagonal entries are the coefficients of the squared terms, and its off-diagonal entries are half the coefficients of the cross product terms.

[DF*] In general, if $A$ is a symmetric $n \times n$ matrix and $\boldsymbol{x}$ is an $n \times 1$ column vector of variables, then we call the function

$$Q_A(\boldsymbol{x}) = \boldsymbol{x}^T A\boldsymbol{x} \tag{23.194}$$

the quadratic form associated with $A$. It can be expressed in dot product notation as

$$\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{x} \cdot A\boldsymbol{x} = A\boldsymbol{x} \cdot \boldsymbol{x} \tag{23.195}$$

In the case where $A$ is a diagonal matrix, the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ has no cross product terms; for example, if $A$ has diagonal entries $\lambda_1, \lambda_2, \ldots, \lambda_n$, then

$$\boldsymbol{x}^T A\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$\boldsymbol{x}^T A\boldsymbol{x} = \lambda_1x_1^2 + \lambda_2x_2^2 + \cdots + \lambda_nx_n^2$$

[DF*] There are three kinds of problems that occur in applications of quadratic forms:

1. If $\boldsymbol{x}^T A\boldsymbol{x}$ is a quadratic form on $\mathbb{R}^2$ or $\mathbb{R}^3$, what kind of curve or surface is represented by the equation $\boldsymbol{x}^T A\boldsymbol{x} = k$ ?
2. If $\boldsymbol{x}^T A\boldsymbol{x}$ is a quadratic form on $\mathbb{R}^n$, what conditions must $A$ satisfy for $\boldsymbol{x}^T A\boldsymbol{x}$ to have positive values for $\boldsymbol{x} \neq \mathbf{0}$ ?
3. If $\boldsymbol{x}^T A\boldsymbol{x}$ is a quadratic form on $\mathbb{R}^n$, what are its maximum and minimum values if $\boldsymbol{x}$ is constrained to satisfy $||\boldsymbol{x}|| = 1$ ?

[DF*] Many of the techniques for solving the problems above are based on simplifying the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ by making a substitution

$$\boldsymbol{x} = P\boldsymbol{y} \tag{23.196}$$

that expresses the variables $x_1, x_2, \ldots, x_n$ in terms of new variables $y_1, y_2, \ldots, y_n$. If $P$ is invertible, then we call (23.196) a change of variable, and if $P$ is orthogonal, then we call (23.196) an orthogonal change of variable.

If we change the variable $\boldsymbol{x} = P\boldsymbol{y}$ in the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$, then we obtain

$$\boldsymbol{x}^T A\boldsymbol{x} = (P\boldsymbol{y})^T A(P\boldsymbol{y}) = \boldsymbol{y}^T P^T AP\boldsymbol{y} = \boldsymbol{y}^T (P^T AP)\boldsymbol{y} \tag{23.197}$$

Since the matrix $B = P^T AP$ is symmetric, the effect of the change of variable is to produce a new quadratic form $\boldsymbol{y}^T B\boldsymbol{y}$ in the variables $y_1, y_2, \ldots, y_n$.

In particular, if we choose $P$ to orthogonally diagonalize $A$, then the new quadratic form will be $\boldsymbol{y}^T D\boldsymbol{y}$, where $D$ is a diagonal matrix with the eigenvalues of $A$ on the main diagonal; that is,

$$\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{y}^T D\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & \ldots & y_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{y}^T D\boldsymbol{y} = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2$$

**Theorem 23.125: The Principal Axes Theorem**

If $A$ is a symmetric $n \times n$ matrix, then there is an orthogonal change of variable that transforms the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ into a quadratic form $\boldsymbol{y}^T D\boldsymbol{y}$ with no cross product terms. Specifically, if $P$ orthogonally diagonalizes $A$, then making the change of variable $\boldsymbol{x} = P\boldsymbol{y}$ in the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ yields the quadratic form

$$\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{y}^T D\boldsymbol{y} = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \tag{23.198}$$

in which $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $A$ corresponding to the eigenvectors that form the successive columns of $P$.

[DF*] If $A$ is a symmetric $n \times n$ matrix, then the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ is a real-valued function whose range is the set of all possible values for $\boldsymbol{x}^T A\boldsymbol{x}$ as $\boldsymbol{x}$ varies over $\mathbb{R}^n$.

It can be shown that an orthogonal change of variable $\boldsymbol{x} = P\boldsymbol{y}$ does not alter the range of a quadratic form; that is, the set of all values for $\boldsymbol{x}^T A\boldsymbol{x}$ as $\boldsymbol{x}$ varies over $\mathbb{R}^n$ is the same as the set of all values for $\boldsymbol{y}^T (P^T AP)\boldsymbol{y}$ as $\boldsymbol{y}$ varies over $\mathbb{R}^n$.

[DF*] Recall that a conic section or conic is a curve that results by cutting a double-napped cone with a plane. The most important conic sections are ellipses, hyperbolas, and parabolas, which result when the cutting plane does not pass through the vertex.

Circles are special cases of ellipses that results when the cutting plane is perpendicular to the axis of symmetry of the cone. If the cutting plane passes through the vertex, then the resulting intersection is called a degenerate conic. The possibilities are a point, a pair of intersecting lines or a single line.

[DF*] Quadratic forms in $\mathbb{R}^2$ arise naturally in the study of conic sections. For example it is shown in analytic geometry that an equation of the form

$$ax^2 + 2bxy + cy^2 + dx + ey + f = 0 \tag{23.199}$$

in which $a, b$, and $c$ are not all zero, represents a conic section. If $d = e = 0$ in () then there are no linear terms, so the equation becomes

$$ax^2 + 2bxy + cy^2 + f = 0 \tag{23.200}$$

and is said to represent central conic. These include circles, ellipses, and hyperbolas, but not parabolas. Furthermore, if $b =$ in, then there is no cross product term, and the equation

$$ax^2 + cy^2 + f = 0 \tag{23.201}$$

is said to represent a central conic in standard position.

If we take the constant $f$ in Equations (23.200) and (23.201) to the right side and let $k = -f$, then we can rewrite these equations in matrix form as

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = k \tag{23.202}$$

this one corresponds to Equation (23.200) in which there is a cross product term $2bxy$.

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & 0 \\ 0 & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = k \tag{23.203}$$

This one corresponds to Equation (23.201) in which there is no cross product term. Geometrically, the existence of a cross product term signals that the graph of the quadratic form is rotated about the origin.

The three-dimensional analogs of the Equation (23.202) is

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} a & d & e \\ d & b & f \\ e & f & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = k \tag{23.204}$$

The three-dimensional analogs of the Equation (23.202) is

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = k \tag{23.205}$$

If $a, b$, and $c$ are not all zero, then the graphs of these equations in $\mathbb{R}^3$ are called central quadrics in standard position.

[DF*] An equation of the form

$$ax^2 + 2bxy + cy^2 + f = 0 \tag{23.206}$$

represents a central conic. If $b = 0$, then the conic is in standard position, and if $b \neq 0$, it is rotated. It is an easy matter to identify central conics in standard position by matching the equation with one of the standard forms:

**Ellipse**

$$\frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} = 1 \tag{23.207}$$

with $\alpha > 0, \beta > 0$.

**Hyperbola type 1**

$$\frac{x^2}{\alpha^2} - \frac{y^2}{\beta^2} = 1 \tag{23.208}$$

with $\alpha > 0, \beta > 0$.

**Hyperbola type 2**

$$\frac{y^2}{\beta^2} - \frac{x^2}{\alpha^2} = 1 \tag{23.209}$$

with $\alpha > 0, \beta > 0$.

[DF*] The equation

$$9x^2 + 16y^2 - 144 = 0$$

can be rewritten as

$$\frac{x^2}{16} + \frac{y^2}{9} = 1$$

which is the ellipse with $\alpha = 4, \beta = 3$.

[DF*] If a central conic is rotated out of standard position, then it can be identified by first rotating the coordinate axes to put it in standard position and then matching the resulting equation with one of standard forms.

To find a rotation that eliminates the cross product term in the equation

$$ax^2 + 2bxy + cy^2 = k \tag{23.210}$$

it will be convenient to express the equation in the matrix form

$$\boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = k \tag{23.211}$$

and look for a change of variable

$$\boldsymbol{x} = P\boldsymbol{x}'$$

that diagonalizes $A$ and for which $\det(P) = 1$. The transition matrix

$$P = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix} \tag{23.212}$$

has the effect of rotating the $xy$-axes of a rectangular coordinate system through an angle $\theta$, our problem reduces to finding $\theta$ that diagonalizes $A$, thereby eliminating the cross product

term in Equation (23.210). If we make this change of variable, then in the $x'y'$-coordinate system. Equation (23.211) will become

$$\boldsymbol{x}^T A\boldsymbol{x} = (P\boldsymbol{x}')^T AP\boldsymbol{x}' = \boldsymbol{x}'^T P^T AP\boldsymbol{x}' = \boldsymbol{x}'^T D\boldsymbol{x}'$$

$$\boldsymbol{x}'^T D\boldsymbol{x}' = \begin{bmatrix} x' & y' \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = k \tag{23.213}$$

where $\lambda_1$ and $\lambda_2$ are the eigenvalues of $A$. The conic can now be identified by writing Equation (23.213) in the form

$$\lambda_1 x'^2 + \lambda_2 y'^2 = k \tag{23.214}$$

and performing the necessary algebra to match it with one of the standard forms.

For example, if $\lambda_1, \lambda_2$, and $k$ are positive, then Equation (23.214) represents an ellipse with an axis of length $2\sqrt{\frac{k}{\lambda_1}}$ in the $x'$-direction and $2\sqrt{\frac{k}{\lambda_2}}$ in the $y'$-direction.

The first column vector of $P$, which is a unit eigenvector corresponding to $\lambda_1$, is along the positive $x'$-axis; and the second column vector of $P$, which is a unit eigenvector corresponding to $\lambda_2$, is a unit vector along the $y'$-axis. These are called the principal axes of the ellipse.

**Figure 23.181:** *The principal axes of the ellipse, the conic above is of form $\lambda_1 x'^2 + \lambda_2 y'^2 = k$, if $\lambda_1, \lambda_2$, and $k$ are positive, then it is an ellipse with $\alpha = \sqrt{\frac{k}{\lambda_1}}$ and $\beta = \sqrt{\frac{k}{\lambda_2}}$.*

[DF*] If we have a conic with equation

$$ax^2 + 2bxy + cy^2 = k$$

with $b \neq 0$, the cross product term from the equation above ($2bxy$) can be eliminated by a rotation through an angle $\theta$ that satisfies

$$\cot\ 2\theta = \frac{a-c}{2b} \tag{23.215}$$

**Definition 23.58: Terminology in Quadratic Form**

A quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ is said to be

- **Positive definite** if $\boldsymbol{x}^T A\boldsymbol{x} > 0$ for $\boldsymbol{x} \neq 0$.
- **Negative definite** if $\boldsymbol{x}^T A\boldsymbol{x} < 0$ for $\boldsymbol{x} \neq 0$.
- **Indefinite** if $\boldsymbol{x}^T A\boldsymbol{x}$has both positive and negative values.

**Theorem 23.126: Terminology for Known Symmetric Matrix and Eigenvalues**

If $A$ is a symmetric matrix, then:

(a) $\boldsymbol{x}^T A\boldsymbol{x}$ is positive definite if and only if all eigenvalues of $A$ are positive.

(b) $\boldsymbol{x}^T A\boldsymbol{x}$ is negative definite if and only if all eigenvalues of $A$ are negative.

(c) $\boldsymbol{x}^T A\boldsymbol{x}$ is indefinite if and only if $A$ has at least one positive eigenvalue and at least one negative eigenvalue.

[DF*] A quadratic form for which $\boldsymbol{x}^T A\boldsymbol{x} \geq 0$ if $\boldsymbol{x} \neq \mathbf{0}$ is called positive semidefinite, and one for which $\boldsymbol{x}^T A\boldsymbol{x} \leq 0$ if $\boldsymbol{x} \neq \mathbf{0}$ is called negative semidefinite.

$\boldsymbol{x}^T A\boldsymbol{x}$ is positive semidefinite if and only if all eigenvalues of $A$ are nonnegative and is negative semidefinite if and only if all eigenvalues of $A$ are nonpositive.

[DF*] **Example:**

It is not usually possible to tell from the signs of the entries in a symmetric matrix $A$ whether that matrix is positive definite, negative definite, or indefinite. For example, the entries of the matrix

$$A = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix}$$

are nonnegative, but the matrix is indefinite since its eigenvalues are

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 4 \\ \lambda_3 &= -2 \end{aligned}$$

If we write the quadratic form, it will be

$$\boldsymbol{x}^T A\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 3 & 1 & 1 \\ 1 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 3x_1^2 + 2x_1x_2 + 2x_1x_3 + 4x_2x_3$$

now, if we use different input for $\boldsymbol{x}$

$$\boldsymbol{x}^T A\boldsymbol{x} = 4$$

for $x_1 = 0, x_2 = 1$, and $x_3 = 1$.

$$\boldsymbol{x}^T A\boldsymbol{x} = -4$$

for $x_1 = 0, x_2 = 1$, and $x_3 = -1$. Thus the quadratic form is indefinite.

[DF*] If $\boldsymbol{x}^T B\boldsymbol{x} = k$ is the equation of a conic, and if $k \neq 0$, then we can divide through by $k$ and rewrite the equation in the form

$$\boldsymbol{x}^T A\boldsymbol{x} = 1 \tag{23.216}$$

where $A = (1/k)B$. If we now rotate the coordinate axes to eliminate the cross product term (if any) in this equation, then the equation of the conic in the new coordinate system will be of the form

$$\lambda_1 x'^2 + \lambda_2 y'^2 = 1 \tag{23.217}$$

in which $\lambda_1$ and $\lambda_2$ are the eigenvalues of $A$. The particular type of conic represented by this equation will depend on the signs of the eigenvalues $\lambda_1$ and $\lambda_2$

- $\boldsymbol{x}^T A\boldsymbol{x} = 1$ represents an ellipse if $\lambda_1 > 0$ and $\lambda_2 > 0$.
- $\boldsymbol{x}^T A\boldsymbol{x} = 1$ has no graph if $\lambda_1 < 0$ and $\lambda_2 < 0$.
- $\boldsymbol{x}^T A\boldsymbol{x} = 1$ represents a hyperbola if $\lambda_1$ and $\lambda_2$ have opposite signs.

In the case of the ellipse, Equation (23.) can be rewritten as

$$\frac{x'^2}{\left(\frac{1}{\sqrt{\lambda_1}}\right)^2} + \frac{y'^2}{\left(\frac{1}{\sqrt{\lambda_2}}\right)^2} = 1 \tag{23.218}$$

so the axes of the ellipse have lengths $2/\sqrt{\lambda_1}$ and $2/\sqrt{\lambda_2}$.

**Theorem 23.127: Classifying Conic Sections**

If $A$ is a symmetric $2 \times 2$ matrix, then:

(a) $\boldsymbol{x}^T A\boldsymbol{x} = 1$ represents an ellipse if $A$ is positive definite.

(b) $\boldsymbol{x}^T A\boldsymbol{x} = 1$ has no graph if $A$ is negative definite.

(c) $\boldsymbol{x}^T A\boldsymbol{x} = 1$ represents an hyperbola if $A$ is indefinite.

[DF*] Positive definite matrices are the most important symmetric matrices in applications, so it will be useful to learn a little more about them. We already know that a symmetric matrix is positive definite if and only if its eigenvalues are all positive.

[DF*] We define the $k$th principal submatrix of an $n \times n$ matrix $A$ to be the $k \times k$ submatrix consisting of the first $k$ rows and columns of $A$.

**Theorem 23.128: Determinant Test for Symmetric Matrix**

A symmetric matrix $A$ is positive definite if and only if the determinant of every principal submatrix is positive.

[DF*] **Example:**
The matrix

$$A = \begin{bmatrix} 2 & -1 & -3 \\ -1 & 2 & 4 \\ -3 & 4 & 9 \end{bmatrix}$$

is positive definite since the determinants

$$|2| = 2, \quad \begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} = 3, \quad \begin{vmatrix} 2 & -1 & -3 \\ -1 & 2 & 4 \\ -3 & 4 & 9 \end{vmatrix} = 1$$

are all positive. Thus, we are guaranteed that all eigenvalues of $A$ are positive and $\boldsymbol{x}^T A\boldsymbol{x} > 0$ for $\boldsymbol{x} \neq \mathbf{0}$.

```
Matrix A:
    (+2.000e+00,+0.000e+00)    (-1.000e+00,+0.000e+00)    (-3.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)    (+2.000e+00,+0.000e+00)    (+4.000e+00,+0.000e+00)
    (-3.000e+00,+0.000e+00)    (+4.000e+00,+0.000e+00)    (+9.000e+00,+0.000e+00)


Submatrix A1 :
    (+2.000e+00,+0.000e+00)

det(Submatrix A1) = (2,0)

Submatrix A2 :
    (+2.000e+00,+0.000e+00)    (-1.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)    (+2.000e+00,+0.000e+00)

det(Submatrix A2) = (3,0)

Submatrix A3 :
    (+2.000e+00,+0.000e+00)    (-1.000e+00,+0.000e+00)    (-3.000e+00,+0.000e+00)
    (-1.000e+00,+0.000e+00)    (+2.000e+00,+0.000e+00)    (+4.000e+00,+0.000e+00)
    (-3.000e+00,+0.000e+00)    (+4.000e+00,+0.000e+00)    (+9.000e+00,+0.000e+00)

det(Submatrix A3) = (1,-0)
```

**Figure 23.182:** *The computation to obtain the determinant for all principal submatrices of matrix A. (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Determinant of Principal Submatrices with Armadillo/main.cpp**).*

[DF*] Positive definite quadratic form:

$$x_1^2 + x_2^2$$

for $(x_1, x_2) \neq 0$ we will have $Q = x_1^2 + x_2^2 > 0$ thus $Q$ is positive definite.

Positive semidefinite quadratic form:

$$(x_1 - x_2)^2$$

for $x_1 \neq x_2$ we will have $Q = (x_1 - x_2)^2 > 0$, and $Q = 0$ for $x_1 = x_2$, thus $Q$ is positive semidefinite.

Negative definite quadratic form:

$$-x_1^2 - 3x_2^2$$

for $(x_1, x_2) \neq 0$ we will have $Q = -x_1^2 - 3x_2^2 < 0$ thus $Q$ is negative definite.

Negative semidefinite quadratic form:

$$-(x_1 - x_2)^2$$

for $(x_1, x_2) \neq 0$ we will have $Q = -(x_1 - x_2)^2 < 0$, and $Q = 0$ for $x_1 = x_2$, thus $Q$ is negative semidefinite.

Indefinite quadratic form:

$$x_1^2 - x_2^2$$

for $x_1 \neq 0, x_2 = 0$ we will have $Q = x_1^2 - x_2^2 > 0$, and for $x_1 = 0, x_2 \neq 0$ we will have $Q = x_1^2 - x_2^2 < 0$ thus $Q$ is indefinite.

[DF*] **Example:**
Let $\boldsymbol{x}^T A\boldsymbol{x}$ be a quadratic form in the variables $x_1, x_2, \ldots, x_n$, and define $T : \mathbb{R}^n \to \mathbb{R}$ by $T(\boldsymbol{x}) = \boldsymbol{x}^T A\boldsymbol{x}$.

(a) Show that $T(\boldsymbol{x}+\boldsymbol{y}) = T(\boldsymbol{x}) + 2\boldsymbol{x}^T A\boldsymbol{y} + T(\boldsymbol{y})$.

(b) Show that $T(c\boldsymbol{x}) = c^2 T(\boldsymbol{x})$.

**Solution:**

(a) We will start with $n = 2$

$$\boldsymbol{x}^T A\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = ax_1^2 + 2bx_1x_2 + cx_2^2$$

now for $T(\boldsymbol{x}+\boldsymbol{y})$

$$T(\boldsymbol{x}+\boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y}^T A\boldsymbol{x} + \boldsymbol{y}$$

$$T(\boldsymbol{x}+\boldsymbol{y}) = \begin{bmatrix} x_1+y_1 & x_2+y_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \end{bmatrix} = a(x_1+y_1)^2 + 2b(x_1+y_1)(x_2+y_2) + c(x_2+y_2)^2$$

and

$$2\boldsymbol{x}^T A\boldsymbol{y} = 2\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = 2ax_1y_1 + 2bx_1y_2 + 2bx_2y_1 + 2cx_2y_2$$

thus

$$\begin{aligned} T(\boldsymbol{x}+\boldsymbol{y}) &= a(x_1+y_1)^2 + 2b(x_1+y_1)(x_2+y_2) + c(x_2+y_2)^2 \\ &= ax_1^2 + 2ax_1y_1 + ay_1^2 + 2bx_1x_2 + 2by_1y_2 + 2bx_1y_2 + 2bx_2y_1 + cx_2^2 + 2cx_2y_2 + cy_2^2 \\ &= (ax_1^2 + 2bx_1x_2 + cx_2^2) + (2ax_1y_1 + 2bx_1y_2 + 2bx_2y_1 + 2cx_2y_2) + (ay1^2 + 2by_1y_2 + cy_2^2) \\ &= T(\boldsymbol{x}) + 2\boldsymbol{x}^T A\boldsymbol{y} + T(\boldsymbol{y}) \end{aligned}$$

If we expand for $n = 3, 4, \ldots$ we will obtain the same general formula / result, thus it is proven that $T(\boldsymbol{x}+\boldsymbol{y}) = T(\boldsymbol{x}) + 2\boldsymbol{x}^T A\boldsymbol{y} + T(\boldsymbol{y})$

(b) We will start with $n = 2$

$$T(c\boldsymbol{x}) = c\boldsymbol{x}^T Ac\boldsymbol{x} = \begin{bmatrix} cx_1 & cx_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & d \end{bmatrix} \begin{bmatrix} cx_1 \\ cx_2 \end{bmatrix} = c^2(ax_1^2 + 2bx_1x_2 + dx_2^2) = c^2 T(\boldsymbol{x})$$

now for $n = 3$

$$T(c\boldsymbol{x}) = c\boldsymbol{x}^T Ac\boldsymbol{x} = \begin{bmatrix} cx_1 & cx_2 & cx_3 \end{bmatrix} \begin{bmatrix} a & b & e \\ b & d & f \\ e & f & g \end{bmatrix} \begin{bmatrix} cx_1 \\ cx_2 \\ cx_3 \end{bmatrix} = c^2(ax_1^2 + 2bx_1x_2 + 2ex_1x_3 + 2fx_2x_3 + dx_2^2 + gx_3^2)$$

$$c^2(ax_1^2 + 2bx_1x_2 + 2ex_1x_3 + 2fx_2x_3 + dx_2^2 + gx_3^2) = c^2 T(\boldsymbol{x})$$

it is proven then, when we expand for $n = 4, 5, 6 \ldots$ that $T(c\boldsymbol{x}) = c^2 T(\boldsymbol{x})$

[DF*] **Example:**
In statistics, the quantities

$$\bar{x} = \frac{1}{n}(x_1 + x_2 + \cdots + x_n)$$

and

$$s_x^2 = \frac{1}{n-1}\left[(x_1-\bar{x})^2 + (x_2-\bar{x})^2 + \cdots + (x_n-\bar{x})^2\right]$$

are called, respectively, the sample mean and sample variance of $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$.

(a) Express the quadratic form $s_x^2$ in the matrix notation $\boldsymbol{x}^T A\boldsymbol{x}$, where $A$ is symmetric.

(b) Is $s_x^2$ a positive definite quadratic form? Explain.

**Solution:**

(a) For each $i = 1, \ldots, n$ we have

$$\begin{aligned}(x_i - \bar{x})^2 &= x_i^2 - 2x_i\bar{x} + \bar{x}^2 \\ &= x_i^2 - 2x_i\frac{1}{n}\sum_{j=1}^{n} x_j + \frac{1}{n^2}\left(\sum_{j=1}^{n} x_j\right)^2 \\ &= x_i^2 - \frac{2}{n}\sum_{j=1}^{n} x_i x_j + \frac{1}{n^2}\left(\sum_{j=1}^{n} x_j^2 + 2\sum_{j=1}^{n}\sum_{k=j+1}^{n} x_j x_k\right)\end{aligned}$$

Thus in the quadratic form

$$s_x^2 = \frac{1}{n-1}\left[(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2\right]$$

the coefficient of $x_i^2$ is

$$\frac{1}{n-1}\left[1 - \frac{2}{n} + \frac{1}{n^2}n\right] = \frac{1}{n}$$

and the coefficient of $x_i x_j$ for $i \neq j$ is

$$\frac{1}{n-1}\left[-\frac{2}{n} - \frac{2}{n} + \frac{2}{n^2}n\right] = \frac{2}{n(n-1)}$$

It follows that

$$s_x^2 = \boldsymbol{x}^T A\boldsymbol{x}$$

where

$$A = \begin{bmatrix} \frac{1}{n} & -\frac{1}{n(n-1)} & \cdots & -\frac{1}{n(n-1)} \\ -\frac{1}{n(n-1)} & \frac{1}{n} & \cdots & -\frac{1}{n(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{1}{n(n-1)} & -\frac{1}{n(n-1)} & \cdots & \frac{1}{n} \end{bmatrix}$$

(b) We have

$$s_x^2 = \frac{1}{n-1}\left[(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2\right] \geq 0$$

and $s_x^2 = 0$ if and only if $x_1 = \bar{x}, x_2 = \bar{x}, \ldots, x_n = \bar{x}$, for example, if and only if $x_1 = x_2 = \cdots = x_n$.

Thus, $s_x^2$ is a positive semidefinite form.

[DF*] **Example:**

The graph in an $xyz$-coordinate system of an equation of form $ax^2 + by^2 + cz^2 = 1$ in which $a, b$, and $c$ are positive is a surface called a central ellipsoid in standard position. This is the three-dimensional generalization of the ellipse $ax^2 + by^2 = 1$ in the $xy$-plane. The coordinate axes determine three line segments called the axes of the ellipsoid. If a central ellipsoid is rotated about the origin so two or more of its axes do not coincide with any of the coordinate axes, then the resulting equation will have one or more cross product terms.

(a) Show that the equation

$$\frac{4}{3}x^2 + \frac{4}{3}y^2 + \frac{4}{3}z^2 + \frac{4}{3}xy + \frac{4}{3}xz + \frac{4}{3}yz = 1$$

represents an ellipsoid, and find the lengths of its axes. Write the equation in the form of $\boldsymbol{x}^T A\boldsymbol{x} = 1$ and make an orthogonal change of variable to eliminate the cross product terms.

(b) What property must a symmetric $3 \times 3$ matrix have in order for the equation $\boldsymbol{x}^T A\boldsymbol{x} = 1$ to represent an ellipsoid?

**Solution:**

(a) Let's write out the quadratic form of $\frac{4}{3}x^2 + \frac{4}{3}y^2 + \frac{4}{3}z^2 + \frac{4}{3}xy + \frac{4}{3}xz + \frac{4}{3}yz = 1$

$$Q = \boldsymbol{x}^T A\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} \frac{4}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{4}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & \frac{4}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Since the equation represent the central ellipsoid in standard position then the major $x$-axis has the length of $2\left(\frac{4}{3}\right) = \frac{8}{3}$, the major $y$-axis has the length of $2\left(\frac{4}{3}\right) = \frac{8}{3}$, and the major $z$-axis has the length of $2\left(\frac{4}{3}\right) = \frac{8}{3}$.

Now we want to eliminate the cross product terms. First, we compute the eigenvalues of $A$

$$\begin{vmatrix} \lambda - \frac{4}{3} & -\frac{2}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \lambda - \frac{4}{3} & -\frac{2}{3} \\ -\frac{2}{3} & -\frac{2}{3} & \lambda - \frac{4}{3} \end{vmatrix} = \left(\lambda - \frac{2}{3}\right)^2 \left(\lambda - \frac{8}{3}\right) = 0$$

so the eigenvalues are

$$\lambda_1 = \frac{2}{3}$$
$$\lambda_2 = \frac{8}{3}$$
$$\lambda_3 = \frac{2}{3}$$

The orthonormal bases for the three eigenspaces are

$$\lambda = \frac{2}{3} : \begin{bmatrix} -0.8165 \\ 0.4082 \\ 0.4082 \end{bmatrix}, \quad \begin{bmatrix} 0.2007 \\ -0.7858 \\ 0.5850 \end{bmatrix}, \quad \lambda = \frac{8}{3} : \begin{bmatrix} 0.5774 \\ 0.5774 \\ 0.5774 \end{bmatrix}$$

we will construct matrix $P$ from the orthonormal bases, but we will arrange the column vector so that $\det(P) = 1$, thus

$$P = [\boldsymbol{v}_2 | \boldsymbol{v}_1 | \boldsymbol{v}_3] \begin{bmatrix} 0.5774 & -0.8165 & 0.2007 \\ 0.5774 & 0.4082 & -0.7858 \\ 0.5774 & 0.4082 & 0.5850 \end{bmatrix}$$

If the standard form of matrix $P = [\boldsymbol{v}_1 | \boldsymbol{v}_2 | \boldsymbol{v}_3]$ has $\det(P) = -1$, then we only need to interchange two column vector in matrix $P$ to make $\det(P) = 1$, either interchange $\boldsymbol{v}_1$

and $v_2$, $v_1$ and $v_3$ or $v_2$ and $v_3$.

This produces the new quadratic form

$$Q = \boldsymbol{y}^T(P^TAP)\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} \frac{2}{3} & 0 & 0 \\ 0 & \frac{8}{3} & 0 \\ 0 & 0 & \frac{2}{3} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \frac{2}{3}y_1^2 + \frac{8}{3}y_2^2 + \frac{2}{3}y_3^2$$

in which there are no cross product terms.

(b) A symmetric $3 \times 3$ matrix $A$ will have the property of $\det(A) \neq 0$ in order for the equation $\boldsymbol{x}^T A\boldsymbol{x} = 1$ to represent an ellipsoid. Furthermore, the eigenvalues of $A$ must be positive.

```
Matrix A:
   1.3333   0.6667   0.6667
   0.6667   1.3333   0.6667
   0.6667   0.6667   1.3333

Eigenvalues:
λ_1 = 0.666667 + 0i
λ_2 = 2.66667 + 0i
λ_3 = 0.666667 + 0i

Orthonormal Eigenvectors:
x[λ_1] :
   -0.8165
   0.4082
   0.4082

x[λ_2] :
    0.5774
   0.5774
   0.5774

x[λ_3] :
    0.2007
  -0.7858
   0.5850

Matrix P (which orthogonally diagonalizes A):
   0.5774  -0.8165   0.2007
   0.5774   0.4082  -0.7858
   0.5774   0.4082   0.5850

det(P):
0.969304
Matrix P^{T} * A * P:
   6.6667e-01   3.8858e-16  -1.6391e-01
   4.4409e-16   2.6667e+00  -2.2204e-16
  -1.6391e-01  -3.0531e-16   6.6667e-01

y =
[y1]
[y2]
[y3]

y^{T} =
[y1 y2 y3]

D =
[0.666667     0        0   ]
[    0     2.66667     0   ]
[    0        0    0.666667]

y^{T} * D * y = 0.666667*y1^(2)+2.66667*y2^(2)+0.666667*y3^(2)
```

**Figure 23.183:** *The computation to find the orthogonal change of variable to eliminate the cross product terms in the equation* $\frac{4}{3}x^2 + \frac{4}{3}y^2 + \frac{4}{3}z^2 + \frac{4}{3}xy + \frac{4}{3}xz + \frac{4}{3}yz = 1$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Principal Axes Theorem Fraction Entries with Armadillo and SymbolicC++/)****.*

[DF*] **Example:**
What property must a symmetric $2 \times 2$ matrix $A$ have for $\boldsymbol{x}^T A\boldsymbol{x} = 1$ to represent a circle?

**Solution:**
The eigenvalues of $A$ must be positive and equal to each other. That is, $A$ must have a positive eigenvalue of multiplicity 2.

$$\lambda_1 = \lambda_2$$
$$\lambda_1 > 0$$
$$\lambda_2 > 0$$

[DF*] **Example:**
Prove: If $b \neq 0$, then the cross product term can be eliminated from the quadratic form $ax^2 + 2bxy + cy^2$ by rotating the coordinate axes through an angle $\theta$ that satisfies the equation

$$\cot\ 2\theta = \frac{a-c}{2b}$$

**Solution:**

We will use the formula for the new quadratic form to eliminate the cross product terms

$$Q = \boldsymbol{y}^T(P^T AP)\boldsymbol{y} = \lambda_1 y_1^2 + \lambda_2 y_2^2$$

with

$$D = P^T AP = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

$$D = \begin{bmatrix} \cos\ \theta & \sin\ \theta \\ -\sin\ \theta & \cos\ \theta \end{bmatrix} \begin{bmatrix} a\cos\ \theta + b\sin\ \theta & -a\sin\ \theta + b\cos\ \theta \\ b\cos\ \theta + c\sin\ \theta & -b\sin\ \theta + c\cos\ \theta \end{bmatrix}$$

$$D = \begin{bmatrix} a\cos^2\ \theta + b\cos\ \theta\ \sin\ \theta + b\cos\ \theta\ \sin\ \theta + c\sin^2\ \theta & -a\cos\ \theta\ \sin\ \theta + b\cos^2\ \theta - b\sin^2\ \theta + c\cos\ \theta\ \sin\ \theta \\ -a\cos\ \theta\ \sin\ \theta + b\cos^2\ \theta - b\sin^2\ \theta + c\cos\ \theta\ \sin\ \theta & a\sin^2\ \theta - b\cos\ \theta\ \sin\ \theta - b\cos\ \theta\ \sin\ \theta + c\cos^2\theta \end{bmatrix}$$

Now we have the equation

$$\begin{aligned} -a\cos\ \theta\ \sin\ \theta + b\cos^2\ \theta - b\sin^2\ \theta + c\cos\ \theta\ \sin\ \theta &= 0 \\ (c-a)\cos\ \theta\ \sin\ \theta + b(\cos^2\ \theta - \sin^2\ \theta) &= 0 \\ (c-a)\cos\ \theta\ \sin\ \theta &= -b(\cos^2\ \theta - \sin^2\ \theta) \\ (c-a)\frac{\sin\ 2\theta}{2} &= -b\cos\ 2\theta \\ \frac{\cos\ 2\theta}{\sin\ 2\theta} &= \frac{(a-c)}{2b} \\ \cot\ 2\theta &= \frac{a-c}{2b} \end{aligned}$$

it is proven then with the double angle identities in trigonometry and the formula of $D = P^T AP$.

[DF*] **Example:**
Prove that if $A$ is an $n \times n$ symmetric matrix all of whose eigenvalues are nonnegative, then $\boldsymbol{x}^T A\boldsymbol{x} \geq 0$ for all nonzero $\boldsymbol{x}$ in $\mathbb{R}^n$.

**Solution:**
If $A$ is a symmetrix matrix, then by the principal axes theorem we know that

$$\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{y}^T D\boldsymbol{y}$$

with $D$ is a diagonal matrix with the eigenvalues of $A$ as the diagonal entries.

Thus

$$\boldsymbol{y}^T D\boldsymbol{y} = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2$$

we know that $y_i^2 \geq 0$ for $i = 1, 2, \ldots, n$ and that the eigenvalues are nonnegative, so

$$\lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \geq 0$$

consequently

$$\boldsymbol{y}^T D\boldsymbol{y} \geq 0 \rightarrow \boldsymbol{x}^T A\boldsymbol{x} \geq 0$$

for all nonzero $\boldsymbol{x}$ in $\mathbb{R}^n$.

[DF*] Quadratic forms arise in various problems in which the maximum or minimum value of some quantity is required.

Consider the problem of finding the maximum and minimum values of quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ subject of the constraint $||\boldsymbol{x}|| = 1$.

To visualize this problem geometrically in the case where $\boldsymbol{x}^T A\boldsymbol{x}$ is a quadratic form on $\mathbb{R}^2$, view $z = \boldsymbol{x}^T A\boldsymbol{x}$ as the equation of some surface in a rectangular $xyz$-coordinate system and view $||\boldsymbol{x}|| = 1$ as the unit circle centered at the origin of the $xy$-plane.

Geometrically, the problem of finding the maximum and minimum values of $\boldsymbol{x}^T A\boldsymbol{x}$ subject to the requirement $||\boldsymbol{x}|| = 1$ amounts to finding the highest and lowest points on the intersection of the surface with the right circular cylinder determined by the circle.

**Theorem 23.129: Constrained Extremum Theorem**

Let $A$ be a symmetric $n \times n$ matrix whose eigenvalues in order of decreasing size are $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Then:

(a) the quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ attains a maximum value and a minimum value on the set of vectors for which $||\boldsymbol{x}|| = 1$;

(b) the maximum value attained in part (a) occurs at a unit vector corresponding to the eigenvalue $\lambda_1$;

(c) the minimum value attained in part (a) occurs at a unit vector corresponding to the eigenvalue $\lambda_n$;

[DF*] **Remark**
The condition $||\boldsymbol{x}|| = 1$ in the theorem above is called a constraint, and the maximum or minimum value of $\boldsymbol{x}^T A\boldsymbol{x}$ subject to the constraint is called a constrained extremum. This constraint can also be expressed as $\boldsymbol{x}^T\boldsymbol{x} = 1$ or as $x_1^2 + x_2^2 + \cdots + x_n^2 = 1$, when convenient.

[DF*] A useful way of visualizing the behavior of a function $f(x, y)$ of two variables is to consider the curves in the $xy$-plane along which $f(x, y)$ is constant. These curves have equations of the form

$$f(x, y) = k$$

and are called the level curves of $f$. In particular, the level curves of a quadratic form $\boldsymbol{x}^T A\boldsymbol{x}$ on $\mathbb{R}^2$ have equations of the form

$$\boldsymbol{x}^T A\boldsymbol{x} = k \tag{23.219}$$

so the maximum and minimum values of $\boldsymbol{x}^T A\boldsymbol{x}$ subject to the constraint $||\boldsymbol{x}|| = 1$ are the largest and smallest values of $k$ for which the graph of (23.219) intersects the unit circle.

Typically, such values of $k$ produce level curves that just touch the unit circle, and the coordinates of the points where the level curves just touch produce the vectors that maximize and minimize $\boldsymbol{x}^T A\boldsymbol{x}$ subject to the constraint $||\boldsymbol{x}|| = 1$.

[DF*] The quadratic forms can be used to study characteristics of real-valued functions of two variables.

If a function $f(x, y)$ has first-order partial derivatives, then its relative maxima and minima, if any, occur at points where

$$f_x(x, y) = 0$$
$$f_y(x, y) = 0$$

These are called critical points of $f$. The specific behavior of $f$ at a critical point $(x_0, y_0)$ is determined by the sign of

$$D(x, y) = f(x, y) - f(x_0, y_0)$$

at points $(x, y)$ that are close to, but different from, $(x_0, y_0)$:

- If $D(x, y) > 0$ at points $(x, y)$ that are sufficiently close to, but different from, $(x_0, y_0)$, then $f(x_0, y_0) < f(x, y)$ at such points and $f$ is said to have a relative minimum at $(x_0, y_0)$.
- If $D(x, y) < 0$ at points $(x, y)$ that are sufficiently close to, but different from, $(x_0, y_0)$, then $f(x_0, y_0) > f(x, y)$ at such points and $f$ is said to have a relative maximum at $(x_0, y_0)$.
- If $D(x, y)$ has both positive and negative values inside every circle centered at $(x_0, y_0)$, then there are points $(x, y)$ that are arbitrarily close to $(x_0, y_0)$ at which $f(x_0, y_0) < f(x, y)$ and points $(x, y)$ that are arbitrarily close to $(x_0, y_0)$ at which $f(x_0, y_0) > f(x, y)$. In this case we say that $f$ has a saddle point at $(x_0, y_0)$.

**Theorem 23.130: Second Derivative Test**

Suppose that $(x_0, y_0)$ is a critical point of $f(x, y)$ and that $f$ has continuous second-order partial derivatives in some circular region centered at $(x_0, y_0)$. Then:

(a) $f$ has a relative minimum at $(x_0, y_0)$ if

$$f_{xx}(x_0, y_0)f_{yy}(x_0, y_0) - f_{xy}^2(x_0, y_0) > 0$$

and

$$f_{xx}(x_0, y_0) > 0$$

(b) $f$ has a relative maximum at $(x_0, y_0)$ if

$$f_{xx}(x_0, y_0)f_{yy}(x_0, y_0) - f_{xy}^2(x_0, y_0) > 0$$

and

$$f_{xx}(x_0, y_0) < 0$$

(c) $f$ has a saddle point at $(x_0, y_0)$ if

$$f_{xx}(x_0, y_0)f_{yy}(x_0, y_0) - f_{xy}^2(x_0, y_0) < 0$$

(d) The test is inconclusive if

$$f_{xx}(x_0, y_0)f_{yy}(x_0, y_0) - f_{xy}^2(x_0, y_0) = 0$$

[DF*] We will reformulate the Second Derivative Test theorem using properties of symmetric matrices. For this purpise we consider the symmetric matrix

$$H(x, y) = \begin{bmatrix} f_{xx}(x, y) & f_{xy}(x, y) \\ f_{xy}(x, y) & f_{yy}(x, y) \end{bmatrix}$$

which is called the Hessian or Hessian matrix of $f$ (from German mathematician and scientist Ludwig Otto Hesse). The notation $H(x, y)$ emphasizes that the entries in the matrix depend on $x$ and $y$. The Hessian is of interest because

$$\det[H(x_0, y_0)] = \begin{vmatrix} f_{xx}(x_0, y_0) & f_{xy}(x_0, y_0) \\ f_{xy}(x_0, y_0) & f_{yy}(x_0, y_0) \end{vmatrix} = f_{xx}(x_0, y_0)f_{yy}(x_0, y_0) - f_{xy}^2(x_0, y_0)$$

is the expression that appears in the Second Derivative Test Theorem.

**Theorem 23.131: Hessian Form of the Second Derivative Test**

Suppose that $(x_0, y_0)$ is a critical point of $f(x, y)$ and that $f$ has continuous second-order partial derivatives in some circular region centered at $(x_0, y_0)$. If $H(x_0, y_0)$ is the Hessian of $f$ at $(x_0, y_0)$, then:

(a) $f$ has a relative minimum at $(x_0, y_0)$ if $H(x_0, y_0)$ is positive definite.

(b) $f$ has a relative maximum at $(x_0, y_0)$ if $H(x_0, y_0)$ is negative definite.

(c) $f$ has a saddle point at $(x_0, y_0)$ if $H(x_0, y_0)$ is indefinite.

(d) The test is inconclusive otherwise.

[DF*] **Example:**
Suppose that the Hessian matrix of a certain quadratic form $f(x, y)$ is

$$H = \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix}$$

What can you say about the location and classification of the critical points of $f$?
**Solution:**
First, we will compute the eigenvalues for matrix $H$.

$$\begin{aligned} \lambda_1 &= 6 \\ \lambda_2 &- 2 \end{aligned}$$

Since $H$ is the Hessian of $f$ at $(x_0, y_0)$ (the critical point for this Hessian matrix), then based on the eigenvalues $H(x_0, y_0)$ is indefinite. Thus, $f$ has a saddle point at $(x_0, y_0)$. How do we determine $(x_0, y_0)$ from the Hessian matrix can be done with using a bit of integration

$$\begin{aligned} f_{xx} &= 2 \\ f_x &= 2x + C_1(y) \\ f &= x^2 + C_1 x(y) + C_2 \end{aligned}$$

the term $C_1 x(y)$ means it is a function of $y$, could be power of $n$ of $y$ multiplied by $x$ with power of 1, $C_1$ and $C_2$ are arbitrary constant.

$$\begin{aligned} f_{yy} &= 2 \\ f_y &= 2y + C_3(x) \\ f &= y^2 + C_3 y(x) + C_4 \end{aligned}$$

From the two equations above we can make an assumption that $C_1 x(y) = C_3 y(x)$, but we will see the last integration for $f_{xy}$ before we make a conclusion

$$\begin{aligned} f_{xy} &= 4 \\ f_x &= 4y + C_5(x) \\ f &= 4xy + C_5 y(x) + C_6 \end{aligned}$$

or

$$f = 4xy + C_5(x) + C_6$$

It seems that $C_5 y(x)$ is a function of $x$ with power of 2 for $x$, and possibily multiplied by power of 1 for $y$ or just $C_5(x)$, hence $C_5 x$ could be $x^2$ and $C_1 x(y) = C_3 y(x) = 4xy$. We can combine the arbitrary values of $C_2, C_4, C_6$ into $C$. Thus, $f$ is probably to be

$$f(x, y) = x^2 + y^2 + 4xy + C$$

where $C$ is a constant. The critical point of the function above is $(0, 0)$ which is also a saddle point. We can say that for the Hessian matrix $H$ above, the critical point is $(0, 0)$ and it is a saddle point.

**Figure 23.184:** *The plot of $x^2 + y^2 + 4xy$.*

[DF*] **Example:**
Suppose that $A$ is an $n \times n$ symmetric matrix and

$$q(\boldsymbol{x}) = \boldsymbol{x}^T A \boldsymbol{x}$$

where $\boldsymbol{x}$ is a vector in $\mathbb{R}^n$ that is expressed in column form. What can you say about the value of $q$ if $\boldsymbol{x}$ is a unit eigenvector corresponding to an eigenvalue $\lambda$ of $A$.

**Solution:**

If $\boldsymbol{x}$ is a unit eigenvector of $A$ corresponding to $\lambda$, then

$$q(\boldsymbol{x}) = \boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{x}^T(\lambda\boldsymbol{x}) = \lambda\boldsymbol{x}^T\boldsymbol{x} = \lambda||\boldsymbol{x}||^2 = \lambda$$

so $q(\boldsymbol{x}) = \lambda$.

[DF*] **Example:**
Prove: If $\boldsymbol{x}^T A\boldsymbol{x}$ is a quadratic form whose minimum and maximum values subject to the constraint $||\boldsymbol{x}|| = 1$ are $m$ and $M$, respectively, then for each number $c$ in the interval $m \leq c \leq M$, there is a unit vector $\boldsymbol{x}_c$ such that $\boldsymbol{x}_c A\boldsymbol{x}_c = c$.

In the case where $m < M$, let $\boldsymbol{u}_m$ and $\boldsymbol{u}_M$ be unit eigenvectors of $A$ such that $\boldsymbol{u}_m^T A\boldsymbol{u}_m = m$ and $\boldsymbol{u}_M^T A\boldsymbol{u}_M = M$, and let

$$\boldsymbol{x}_x = \sqrt{\frac{M-c}{M-m}}\boldsymbol{u}_m + \sqrt{\frac{c-m}{M-m}}\boldsymbol{u}_M$$

Show that $\boldsymbol{x}_c^T A\boldsymbol{x}_c = c$.

**Solution:**
We will examine the inequality, to perform a subtraction with $m$

$$\begin{aligned} m \leq c \leq M \\ 0 \leq c - m \leq M - m \\ 0 \leq \frac{c-m}{M-m} \leq 1 \end{aligned}$$

and then another examination for the inequality, in this case we will subtract all of them with $c$

$$\begin{aligned} m \leq c \leq M \\ m - c \leq 0 \leq M - c \end{aligned}$$

From the inequality, we know that $M - c \leq M - m$, thus

$$\begin{aligned} 0 \leq M - c \leq M - m \\ 0 \leq \frac{M-c}{M-m} \leq 1 \end{aligned}$$

The checking of the inequalities above are to determine the scalar for the linear combination of $\boldsymbol{u}_m$ and $\boldsymbol{u}_M$, as $\boldsymbol{x}_c$ is a vector that is formed from the orthonormal eigenvectors of $A$. Orthonormal eigenvectors are orthonormal bases for the eigenspaces of $m$ and $M$.

For any values of $c$ in the interval $m \leq c \leq M$, if we pick any arbitrary value of $c$ in that interval then there is a unit vector $\boldsymbol{x}_c$ such that $\boldsymbol{x}_c^T A\boldsymbol{x}_c = c$.

Now, we are going to check the norm of $\boldsymbol{x}_c$ (to prove that it is really a unit vector)

$$
\begin{aligned}
\boldsymbol{x}_c &= \sqrt{\frac{M-c}{M-m}}\boldsymbol{u}_m + \sqrt{\frac{c-m}{M-m}}\boldsymbol{u}_M \\
||\boldsymbol{x}_c|| &= ||\sqrt{\frac{M-c}{M-m}}\boldsymbol{u}_m + \sqrt{\frac{c-m}{M-m}}\boldsymbol{u}_M|| \\
&= \left(\sqrt{\frac{M-c}{M-m}}\right)^2 ||\boldsymbol{u}_m|| + \left(\sqrt{\frac{c-m}{M-m}}\right)^2 ||\boldsymbol{u}_M|| \\
&= \frac{M-c}{M-m} + \frac{c-m}{M-m} \\
||\boldsymbol{x}_c|| &= \frac{M-m}{M-m} = 1
\end{aligned}
$$

it is proven then that $\boldsymbol{x}_c$ is a unit vector (remember the proof for the theorem of Triangle Inequality for vectors that state $||\boldsymbol{u}+\boldsymbol{v}||^2 = (||\boldsymbol{u}|| + ||\boldsymbol{v}||)^2$). Thus

$$
\begin{aligned}
\boldsymbol{x}_c^T A \boldsymbol{x}_c &= \boldsymbol{x}_c^T c \boldsymbol{x}_c \\
&= c\boldsymbol{x}_c^T \boldsymbol{x}_c \\
&= c||\boldsymbol{x}_c||^2 \\
&= c(1) = c
\end{aligned}
$$

all values of $c = \boldsymbol{x}_c^T A \boldsymbol{x}_c$ for which $||\boldsymbol{x}|| = 1$ lie between the largest eigenvalue of $A$ ($M$) and the smallest eigenvalue of $A$ ($m$).

## i. C++ Computation: Expressing Quadratic Forms in Matrix Notation

Express the quadratic form in the matrix notation $\boldsymbol{x}^T A\boldsymbol{x}$, where $A$ is symmetric.

$$x_1^2 + 7x_2^2 - 3x_3^2 + 4x_1x_2 - 2x_1x_3 + 8x_2x_3$$

**Solution:**
The diagonal entries of $A$ are the coefficients of the squared terms, and the off-diagonal entries are half the coefficients of the cross product terms, so

$$x_1^2 + 7x_2^2 - 3x_3^2 + 4x_1x_2 - 2x_1x_3 + 8x_2x_3 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 2 & 7 & 4 \\ -1 & 4 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The C++ code is using **SymbolicC++** to be able to do symbolic computation.

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

double halving(double x)
{
       double y = 2;
       return x/y;
}

int main(void)
{
       Symbolic x1("x1"), x2("x2"), x3("x3");
       Symbolic xt = ( ( x1 ), (x2), (x3) );
       Symbolic x =xt.transpose();
       float a11 = 1; //the coefficient for x1^2
       float a22 = 7; //the coefficient for x2^2
       float a33 = -3; //the coefficient for x3^2
       float a12 = halving(4); // the coefficient for x12 divide by 2
       float a13 = halving(-2); // the coefficient for x13 divide by 2
       float a23 = halving(8); // the coefficient for x23 divide by 2

       Symbolic A = ( (Symbolic(a11), Symbolic(a12), Symbolic(a13) ),
       ( Symbolic(a12), Symbolic(a22), Symbolic(a23) ),
       ( Symbolic(a13), Symbolic(a23), Symbolic(a33) ) );

       cout << "x = " << x << endl;
       cout << "x^{T} = " << xt << endl;
       cout << "A = " <<A << endl;
       cout << "x^{T} * A* x = " << xt*A*x << endl;

       return 0;
}
```

**C++ Code 210:** *main.cpp "Expressing quadratic forms in matrix notation"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In the main code we assign each coefficients for $x_1^2, x_2^2, x_3^2, x_1x_2, x_2x_3, x_1x_3$, since the matrix $A$ is symmetric the value for $a_{ij} = a_{ji}$ for $i \neq j$.

  We declare the matrix $A$ and vector $\boldsymbol{x}$ as **Symbolic**, even if the input for matrix $A$ will be float, it is necessary so we can do the computation for $\boldsymbol{x}^T A\boldsymbol{x}$.

```
Symbolic x1("x1"), x2("x2"), x3("x3");
Symbolic xt = ( ( x1 ), (x2), (x3) );
Symbolic x =xt.transpose();
float a11 = 1; //the coefficient for x1^2
float a22 = 7; //the coefficient for x2^2
float a33 = -3; //the coefficient for x3^2
float a12 = halving(4); // the coefficient for x12 divide by 2
float a13 = halving(-2); // the coefficient for x13 divide by
    2
float a23 = halving(8); // the coefficient for x23 divide by 2

Symbolic A = ( (Symbolic(a11), Symbolic(a12), Symbolic(a13) ),
( Symbolic(a12), Symbolic(a22), Symbolic(a23) ),
( Symbolic(a13), Symbolic(a23), Symbolic(a33) ) );
```

```
x =
[x1]
[x2]
[x3]

x^{T} =
[x1 x2 x3]

A =
[ 1  2 -1]
[ 2  7  4]
[-1  4 -3]

x^{T} * A* x = x1^(2)+4*x2*x1-2*x3*x1+7*x2^(2)+8*x3*x2-3*x3^(2)
```

**Figure 23.185:** *The computation to obtain the entries for symmetric matrix $A$ from quadratic form $x_1^2 + 7x_2^2 - 3x_3^2 + 4x_1x_2 - 2x_1x_3 + 8x_2x_3$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix A from Quadratic Forms with SymbolicC++/main.cpp**).*

## ii. C++ Computation: Principal Axes Theorem

Find an orthogonal change of variable that eliminates the cross product terms in the quadratic form

$$Q = x_1^2 - x_3^2 - 4x_1x_2 + 4x_2x_3$$

and express $Q$ in terms of the new variables.

**Solution:**

The quadratic form can be expressed in matrix notation as

$$Q = \boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} 1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The characteristic equation of the matrix $A$ is

$$\lambda^3 - 9\lambda = \lambda(\lambda+3)(\lambda-3) = 0$$

so the eigenvalues are

$$\begin{aligned} \lambda_1 &= 0 \\ \lambda_2 &= -3 \\ \lambda_3 &= 3 \end{aligned}$$

The orthonormal bases for the three eigenspaces are

$$\lambda = 0: \begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \\ \frac{2}{3} \end{bmatrix}, \quad \lambda = -3: \begin{bmatrix} -\frac{1}{3} \\ -\frac{2}{3} \\ \frac{2}{3} \end{bmatrix}, \quad \lambda = 3: \begin{bmatrix} -\frac{2}{3} \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$$

Thus, a substitution $\boldsymbol{x} = P\boldsymbol{y}$ that eliminates the cross product terms is

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{2}{3} \\ \frac{1}{3} & -\frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

This produces the new quadratic form

$$Q = \boldsymbol{y}^T (P^T A P) \boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = -3y_2^2 + 3y_3^2$$

in which there are no cross product terms.

The C++ code is using the library **Armadillo** for the numerical matrix computation and **SymbolicC++** for the symbolic matrix computation to obtain the quadratic form $-3y_2^2 + 3y_3^2$, with the code since the column of the eigenvectors are interchanged thus we obtain $3y_1^2 - 3y_3^2$

```
#include <iostream>
#include <fstream>
```

```
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

#include "symbolicc++.h"

using namespace std;
using namespace arma;

double halving(double x)
{
       double y = 2;
       return x/y;
}

// Driver code
int main(int argc, char** argv)
{
       mat A(3,3, fill::zeros);
       cx_mat eigvec;
       cx_vec eigval;

       mat I(3,3,fill::eye);
       // We create symmetric matrix A from quadratic form Q
       int N = 3;
       float matrixA[N][N] = {};

       matrixA[0][0] = 1;
       matrixA[0][1] = halving(-4);
       matrixA[0][2] = 0;
       matrixA[1][0] = matrixA[0][1];
       matrixA[1][1] = 0;
       matrixA[1][2] = halving(4);
       matrixA[2][0] = matrixA[0][2];
       matrixA[2][1] = matrixA[1][2];
       matrixA[2][2] = -1;

       for (int i = 0; i < N; ++i)
       {
              for(int j = 0; j<N; ++j)
              {
                     A[i+j*N] = matrixA[i][j];
              }
       }

       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix
```

```
cout << "Eigenvalues:" << endl;

int n = 3;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

cout << endl;
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;
mat A3 = I*real(eigval[2]) - A;

cout << "Orthonormal Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

// Save each orthonormal eigenvectors as a vector with Armadillo
arma::vec v1 = real(eigvec.col(0));
arma::vec v2 = real(eigvec.col(1));
arma::vec v3 = real(eigvec.col(2));

mat P = real(eigvec);
mat P_transpose = P.t();
mat D = P_transpose*A*P;

float matrixD[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              matrixD[i][j] = D[i+j*N];
       }
}

cout <<"Matrix P (which orthogonally diagonalizes A):" << "\n" << P
    <<endl;
cout <<"Matrix P^{T} * A * P:" << "\n" << D <<endl;

Symbolic y1("y1"), y2("y2"), y3("y3");

Symbolic yt = ( ( y1 ), (y2), (y3) );
Symbolic y =yt.transpose();
```

```
        float d11 = lround(matrixD[0][0]);
        float d22 = lround(matrixD[1][1]);
        float d33 = lround(matrixD[2][2]);
        float d12 = lround(matrixD[0][1]);
        float d13 = lround(matrixD[0][2]);
        float d23 = lround(matrixD[1][2]);

        Symbolic MatrixD = ( (Symbolic(d11), Symbolic(d12), Symbolic(d13) ),
        ( Symbolic(d12), Symbolic(d22), Symbolic(d23) ),
        ( Symbolic(d13), Symbolic(d23), Symbolic(d33) ) );

        cout << "y = " << y << endl;
        cout << "y^{T} = " << yt << endl;
        cout << "D = " << MatrixD << endl;
        cout << "y^{T} * D * y = " << yt*MatrixD*y << endl;

        return 0;
    }
```

**C++ Code 211:** *main.cpp "Principal axes theorem"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We are using the standard C++ function, **lround()** to round the **float** into **integer** nicely when we create **Symbolic MatrixD** with **SymbolicC++**, since the new quadratic form needs to be in symbolic computation, this is computed after we are done with numerical matrix computation with **Armadillo** at the beginning of the code to compute the eigenvalues, eigenvectors, and matrix $P$.

  Some explanation for rounding functions that available:

  1. lround( ) – The lround( ) function in C++ rounds the integer value that is nearest to the argument, with halfway cases rounded away from zero. The value returned is of type long int. It is similar to the round() function, but returns a long int whereas round returns the same data type as the input. It takes the same parameters as that of round( ) function.
  2. llround( ) – The llround( ) function in C++ rounds the integer value that is nearest to the argument, with halfway cases rounded away from zero. The value returned is of type long long int. It is similar to the lround( ) function, but returns a long long int whereas lround( ) returns long int. It also takes the same parameters as that of round( ) function.

3. round(), std::round() can also be used . Header files used -> cmath, ctgmath

```
...
...

cout <<"Matrix P (which orthogonally diagonalizes A):" << "\n"
    << P <<endl;
cout <<"Matrix P^{T} * A * P:" << "\n" << D <<endl;

Symbolic y1("y1"), y2("y2"), y3("y3");

Symbolic yt = ( ( y1 ), (y2), (y3) );
Symbolic y =yt.transpose();

float d11 = lround(matrixD[0][0]);
float d22 = lround(matrixD[1][1]);
float d33 = lround(matrixD[2][2]);
float d12 = lround(matrixD[0][1]);
float d13 = lround(matrixD[0][2]);
float d23 = lround(matrixD[1][2]);

Symbolic MatrixD = ( (Symbolic(d11), Symbolic(d12), Symbolic(
    d13) ),
( Symbolic(d12), Symbolic(d22), Symbolic(d23) ),
( Symbolic(d13), Symbolic(d23), Symbolic(d33) ) );

cout << "y = " << y << endl;
cout << "y^{T} = " << yt << endl;
cout << "D = " << MatrixD << endl;
cout << "y^{T} * D * y = " << yt*MatrixD*y << endl;

...
```

```
Matrix A:
   1.0000  -2.0000        0
  -2.0000        0   2.0000
        0   2.0000  -1.0000

Eigenvalues:
λ_1 = 3 + 0i
λ_2 = -7.62194e-17 + 0i
λ_3 = -3 + 0i

Orthonormal Eigenvectors:
x[λ_1] :
   -0.6667
   0.6667
   0.3333

x[λ_2] :
   -0.6667
  -0.3333
  -0.6667

x[λ_3] :
   -0.3333
  -0.6667
   0.6667

Matrix P (which orthogonally diagonalizes A):
  -0.6667  -0.6667  -0.3333
   0.6667  -0.3333  -0.6667
   0.3333  -0.6667   0.6667

Matrix P^{T} * A * P:
   3.0000e+00   1.1102e-16   5.5511e-16
   1.1102e-16  -4.9304e-32   4.4409e-16
   6.6613e-16   2.2204e-16  -3.0000e+00

y =
[y1]
[y2]
[y3]

y^{T} =
[y1 y2 y3]

D =
[ 3  0  0]
[ 0  0  0]
[ 0  0 -3]

y^{T} * D * y = 3*y1^(2)-3*y3^(2)
```

**Figure 23.186:** *The computation to obtain the new quadratic form* $Q = \boldsymbol{y}^T(P^TAP)\boldsymbol{y}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Principal Axes Theorem with Armadillo and SymbolicC++/**).*

### iii. C++ Simulation: Ellipse from Conic section

In this simulation we want to draw an ellipse. The general formula for an ellipse is

$$\frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} = 1 \tag{23.220}$$

with $\alpha > 0$ and $\beta > 0$. The parametric equation for ellipse is

$$(x, y) = (\alpha \ \cos \ t, \beta \ \sin \ t) \tag{23.221}$$

with $t \in \mathbb{R}$.

The C++ code is using **Imgui** as the widget tool. The directory has structure of

- include
  - stdafx.hpp (headers only)
- src
  - imgui folder
    (this folder can also be obtained after you install Imgui manually, the Imgui version here is **Dear Imgui, v1.80 WIP**, if your installed Imgui in **/usr/include/imgui** has different version, then no need to worry or install new Imgui, since the source code here will be compiled with CMake specifically will look for Imgui from the current working directory **../src/imgui/**)
  - ball.h
  - ellipse.cpp (can be treated like main.cpp, you edit and modify the source code mainly from here)
  - imgui_helper.hpp

```
#include <stdafx.hpp>
#include "imgui_helper.hpp"
#include "ball.h"
#include <tuple>
#include <thread>
#include <chrono>
#include <vector>
#include <math.h>
#include <list>
#include <numeric>

using namespace std;

int main(int argc, char const *argv[])
{

        int w = 1280;
        int h = 720;
        std::string title = "Conic Simulation";
        initImgui(w, h, title);
```

```
        // simulation state
        bool show_demo_window = true;
        bool show_another_window = false;
        ImVec4 clear_color = ImVec4(89.0f / 255.0, 88.0f / 255.0, 87.0f /
            255.0, 1.00f);
        ImVec4 ray_color = ImVec4(1.0f, 1.0f, 1.0f, 1.0f);
        ImVec4 ray_color2 = ImVec4(0.5f, 0.5f, 0.5f, 1.0f);

        // Main loop
        int ii = 0;
        bool flag = true;
        std::list<ImVec2> wave_tail;
        std::list<int> test;


        while (!glfwWindowShouldClose(window) && flag == true)
        {
               glfwPollEvents();

               ImGui_ImplOpenGL3_NewFrame();
               ImGui_ImplGlfw_NewFrame();

               ImGui::NewFrame();

               ImGuiWindowFlags window_flags = 0;
               ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_FirstUseEver)
                   ;
               ImGui::SetNextWindowSize(ImVec2(w, h), ImGuiCond_FirstUseEver
                   );
               window_flags |= ImGuiWindowFlags_NoMove;
               window_flags |= ImGuiWindowFlags_NoResize;
               window_flags |= ImGuiWindowFlags_NoCollapse;
               window_flags |= ImGuiWindowFlags_NoBackground;
               window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus;

               ImGui::Begin("Simulation", nullptr, window_flags);
               ImDrawList *draw_list = ImGui::GetWindowDrawList();

               float strokex = 600.0; // the beginning of the drawing for
                   ellipse simulation
               float strokey = 410.0; // the beginning of the drawing for
                   ellipse simulation

               static int alpha = 51;
               static int beta = 60;

               ImVec2 wave_N = {std::cos((float)ii)*alpha + strokex, std::
```

```
                    sin((float)ii)*beta + strokey};

                wave_tail.push_front(wave_N);

                // This for loop is the one running the dots that will make
                    the shape of an ellipse
                for (auto &it : wave_tail)
                {
                       draw_list->AddCircleFilled(it, 1.0, IM_COL32(45, 0,
                           255, 255));
                }
                ii++;
                // the if statement to restart the drawing after variable ii
                    reach 801
                if (ii > 800)
                {
                       wave_tail.erase(wave_tail.begin(), wave_tail.end());
                       ii = 0;
                }
                ImGui::PushItemWidth(200);
                ImGui::SliderInt("Alpha", &alpha, 1, 300);
                ImGui::SliderInt("Beta", &beta, 1, 300);
                ImGui::Text("Countdown: %d ", 800-ii);
                ImGui::PopItemWidth();
                ImGui::End();

                // Rendering
                ImGui::Render();
                int display_w, display_h;
                glfwGetFramebufferSize(window, &display_w, &display_h);
                glViewport(0, 0, display_w, display_h);
                glClearColor(clear_color.x, clear_color.y, clear_color.z,
                    clear_color.w);
                glClear(GL_COLOR_BUFFER_BIT);
                ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

                glfwSwapBuffers(window);
            }

            termImgui();
            return 0;
        }
```

**C++ Code 212:** *src/ellipse.cpp "2D Ellipse Simulation"*

To compile it, go to the directory **DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/**, then type:

**mkdir build**

**cd build**
**cmake ..**
**make**
**./2D_Ellipse_Simulation**

Explanation for the codes:

- We declare **float** variables **strokex** and **strokey** to define the center of the ellipse that will be drawn.

  We define the parameters $\alpha$ and $\beta$ as **static int**, and it can be changed when the simulation is running.

  The function from **ImGui**, which is **ImVec2**, to create a vector is used to draw the ellipse, we put the parametric equation for ellipse here.

```
...
...

float strokex = 600.0; // the beginning of the drawing for
    ellipse simulation
float strokey = 410.0; // the beginning of the drawing for
    ellipse simulation

static int alpha = 51;
static int beta = 60;

ImVec2 wave_N = {std::cos((float)ii)*alpha + strokex, std::sin
    ((float)ii)*beta + strokey};

...
...
```

```
xterm
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation ]# mkdir build
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation ]# cd build/
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/build ]# cmake ..
-- The C compiler identification is GNU 11.2.0
-- The CXX compiler identification is GNU 11.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenGL: /usr/lib/libGL.so
-- Configuring done
-- Generating done
-- Build files have been written to: /root/latex/DianFreya Math Physics Simulato
r/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Ell
ipse Simulation/build
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/build ]# make
[ 11%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui.cpp.o
[ 22%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui_demo.cpp.o
[ 33%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui_draw.cpp.o
[ 44%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui_widgets.cpp.o
[ 55%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui_impl_glfw.cpp.o
[ 66%] Building CXX object CMakeFiles/imgui.dir/src/imgui/imgui_impl_opengl3.cpp
.o
[ 77%] Linking CXX static library libimgui.a
[ 77%] Built target imgui
[ 88%] Building CXX object CMakeFiles/2D_Ellipse_Simulation.dir/src/ellipse.cpp.
o
[100%] Linking CXX executable 2D_Ellipse_Simulation
[100%] Built target 2D_Ellipse_Simulation
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/build ]# ./2D_Elli
pse_Simulation
```

**Figure 23.187:** *How to build the simulation with* ***Cmake*** *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/**).*

**Figure 23.188:** *The ellipse simulation that is being drawn from the parametric equation* $(x, y) = (\alpha \cos t, \beta \sin t)$ *based on the* $\alpha$ *and* $\beta$ *that we can change during the simulation (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Ellipse Simulation/**).*

## iv. C++ Simulation: Hyperbola from Conic section

In this simulation we want to draw a hyperbola. There are two types of hyperbola, the first one has the general formula of

$$\frac{x^2}{\alpha^2} - \frac{y^2}{\beta^2} = 1 \tag{23.222}$$

with $\alpha > 0$ and $\beta > 0$. The parametric equation for hyperbola type 1 is

$$(x, y) = (\alpha \ \cosh \ t, \beta \ \sinh \ t) \tag{23.223}$$

with $t \in \mathbb{R}$. This hyperbola has an extreme point with $y' = \infty$ for the graph on positive $x$-axis at point $(\alpha, 0)$, and an extreme point with $y' = \infty$ for the graph on negative $x$-axis at point $(-\alpha, 0)$.

The second type of hyperbola has the general formula of

$$\frac{y^2}{\beta^2} - \frac{x^2}{\alpha^2} = 1 \tag{23.224}$$

with $\alpha > 0$ and $\beta > 0$. The parametric equation for hyperbola type 2 is

$$(x, y) = (\alpha \ \sinh \ t, \beta \ \cosh \ t) \tag{23.225}$$

with $t \in \mathbb{R}$. This hyperbola has an extreme point with $y' = 0$ for the graph on positive $y$-axis at point $(0, \beta)$, and an extreme point with $y' = 0$ for the graph on negative $y$-axis at point $(0, -\beta)$.

The C++ code is using **Imgui** as the widget tool. The directory has structure of

- include
  - stdafx.hpp (headers only)
- src
  - imgui folder
    (this folder can also be obtained after you install Imgui manually, the Imgui version here is **Dear Imgui, v1.80 WIP**, if your installed Imgui in **/usr/include/imgui** has different version, then no need to worry or install new Imgui, since the source code here will be compiled with CMake specifically will look for Imgui from the current working directory **../src/imgui/**)
  - ball.h
  - hyperbola.cpp (can be treated like main.cpp, you edit and modify the source code mainly from here)
  - imgui_helper.hpp

We have two source codes, the first one is to show hyperbola type 1 $\left(\frac{x^2}{\alpha^2} - \frac{y^2}{\beta^2} = 1\right)$, and the second source code is to show hyperbola type 2 $\left(\frac{y^2}{\beta^2} - \frac{x^2}{\alpha^2} = 1\right)$.

```
#include <stdafx.hpp>
#include "imgui_helper.hpp"
#include "ball.h"
#include <tuple>
#include <thread>
```

```
#include <chrono>
#include <vector>
#include <math.h>
#include <list>
#include <numeric>

using namespace std;

int main(int argc, char const *argv[])
{

        int w = 1280;
        int h = 720;
        std::string title = "Hyperbola Type 1 Simulation";
        initImgui(w, h, title);

        // simulation state
        bool show_demo_window = true;
        bool show_another_window = false;
        ImVec4 clear_color = ImVec4(89.0f / 255.0, 88.0f / 255.0, 87.0f /
            255.0, 1.00f);
        ImVec4 ray_color = ImVec4(1.0f, 1.0f, 1.0f, 1.0f);
        ImVec4 ray_color2 = ImVec4(0.5f, 0.5f, 0.5f, 1.0f);

        // Main loop
        int i = 0;
        bool flag = true;
        std::list<ImVec2> wave_tail;
        std::list<int> test;


        while (!glfwWindowShouldClose(window) && flag == true)
        {
                glfwPollEvents();

                ImGui_ImplOpenGL3_NewFrame();
                ImGui_ImplGlfw_NewFrame();

                ImGui::NewFrame();

                ImGuiWindowFlags window_flags = 0;
                ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_FirstUseEver)
                    ;
                ImGui::SetNextWindowSize(ImVec2(w, h), ImGuiCond_FirstUseEver
                    );
                window_flags |= ImGuiWindowFlags_NoMove;
                window_flags |= ImGuiWindowFlags_NoResize;
                window_flags |= ImGuiWindowFlags_NoCollapse;
```

```
window_flags |= ImGuiWindowFlags_NoBackground;
window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus;

ImGui::Begin("Simulation", nullptr, window_flags);
ImDrawList *draw_list = ImGui::GetWindowDrawList();

float strokex = 600.0; // the hyperbola' center x-coordinate
float strokey = 410.0; // the hyperbola' center y-coordinate

static int alpha = 5;
static int beta = 3;

ImVec2 wave_N1 = {coshf((float)i)*alpha + strokex, sinhf((
    float)i)*beta + strokey};
ImVec2 wave_N2 = {coshf((float)i)*alpha + strokex, -sinhf((
    float)i)*beta + strokey};
ImVec2 wave_N3 = {-coshf((float)i)*alpha + strokex, sinhf((
    float)i)*beta + strokey};
ImVec2 wave_N4 = {-coshf((float)i)*alpha + strokex, -sinhf((
    float)i)*beta + strokey};

wave_tail.push_front(wave_N1);
wave_tail.push_front(wave_N2);
wave_tail.push_front(wave_N3);
wave_tail.push_front(wave_N4);

// This for loop is the one drawing the hyperbola that is
    being drawn
for (auto &it : wave_tail)
{
       draw_list->AddCircleFilled(it, 6.0, IM_COL32(65, 0,
           255, 255));
}

i++;
// the if statement to restart the drawing after variable i
    reach 1001
if (i > 100)
{
       wave_tail.erase(wave_tail.begin(), wave_tail.end());
       i = 0;
}
// To draw the hyperbola with explicit function
struct Funcs
{
       static float Hyperbola1(void*, int i) { return sqrt(
           pow(beta,2) * ((pow(-i,2) / pow(alpha,2)) - 1)) ;
            }
```

```
                        static float Hyperbola2(void*, int i) { return - sqrt(
                             pow(beta,2) * ((pow(-i,2) / pow(alpha,2)) - 1))
                            ; }
                    };
                    static int func_type = 0, display_count = 50;
                    ImGui::Combo("func", &func_type, "Hyperbola1\0Hyperbola2");
                    float (*func)(void*, int) = (func_type == 0) ? Funcs::
                        Hyperbola1 : Funcs::Hyperbola2;

                    // PlotLines (const char* label, float(*values_getter)(void*
                        data, int idx), void* data, int values_count,
                    // int values_offset=0, const char* overlay_text = NULL,
                    // float scale_min = FLT_MAX, float scale_max = FLT_MAX,
                        ImVec2 graph_size = (ImVec(0,100));
                    ImGui::PlotLines("", func, NULL, display_count, 0, NULL,
                        -5.0f, 5.0f, ImVec2(0, 100));

                    ImGui::PushItemWidth(200);
                    ImGui::SliderInt("Alpha", &alpha, 1, 30);
                    ImGui::SliderInt("Beta", &beta, 1, 30);
                    ImGui::Text("Countdown: %d ", 100-i);
                    ImGui::PopItemWidth();
                    ImGui::End();

                    // Rendering
                    ImGui::Render();
                    int display_w, display_h;
                    glfwGetFramebufferSize(window, &display_w, &display_h);
                    glViewport(0, 0, display_w, display_h);
                    glClearColor(clear_color.x, clear_color.y, clear_color.z,
                        clear_color.w);
                    glClear(GL_COLOR_BUFFER_BIT);
                    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

                    glfwSwapBuffers(window);
            }

            termImgui();
            return 0;
    }
```

**C++ Code 213:** *src/hyperbola.cpp "2D hyperbola type 1 simulation"*

To compile it, go to the directory **DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Hyperbola Type 1 Simulation/**, then type:

**mkdir build**
**cd build**
**cmake ..**

**make**
**./2D_Hyperbola_Type_1_Simulation**

Explanation for the codes:

- The codes to create the drawing of circle filled for hyperbola type 1 with parametric equation are divided into 4
  1. **ImVec2 wave_N1** will create the circle filled for the first quadrant (top right).
  2. **ImVec2 wave_N2** will create the circle filled for the first quadrant (top left).
  3. **ImVec2 wave_N3** will create the circle filled for the first quadrant (bottom left).
  4. **ImVec2 wave_N4** will create the circle filled for the first quadrant (bottom right).

  We join all of these 4 into 1 **wave_tail**.

```
...
...

ImVec2 wave_N1 = {coshf((float)i)*alpha + strokex, sinhf((float
    )i)*beta + strokey};
ImVec2 wave_N2 = {coshf((float)i)*alpha + strokex, -sinhf((
    float)i)*beta + strokey};
ImVec2 wave_N3 = {-coshf((float)i)*alpha + strokex, sinhf((
    float)i)*beta + strokey};
ImVec2 wave_N4 = {-coshf((float)i)*alpha + strokex, -sinhf((
    float)i)*beta + strokey};

wave_tail.push_front(wave_N1);
wave_tail.push_front(wave_N2);
wave_tail.push_front(wave_N3);
wave_tail.push_front(wave_N4);

// This for loop is the one drawing the hyperbola that is being
     drawn
for (auto &it : wave_tail)
{
       draw_list->AddCircleFilled(it, 6.0, IM_COL32(65, 0, 255,
            255));
}

i++;
// the if statement to restart the drawing after variable i
    reach 1001
if (i > 100)
{
       wave_tail.erase(wave_tail.begin(), wave_tail.end());
       i = 0;
}

...
```

```
...
```

**Figure 23.189:** *The simulation for hyperbola type 1 ($\frac{x^2}{\alpha^2} - \frac{y^2}{\beta^2} = 1$), on the top left, we can see the drawing of the hyperbola for positive $y = \sqrt{\left(\frac{x^2}{\alpha^2} - 1\right)\beta^2}$ (using explicit equation), while the main scene is showing cicles filled / that represent all the points for the parametric equation for hyperbola type 1, $(x, y) = (\alpha \cosh t, \beta \sinh t)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Hyperbola Type 1 Simulation/**).*

```
#include <stdafx.hpp>
#include "imgui_helper.hpp"
#include "ball.h"
#include <tuple>
#include <thread>
#include <chrono>
#include <vector>
#include <math.h>
#include <list>
#include <numeric>

using namespace std;

int main(int argc, char const *argv[])
{

        int w = 1280;
        int h = 720;
        std::string title = "Hyperbola Type 1 Simulation";
        initImgui(w, h, title);

        // simulation state
        bool show_demo_window = true;
        bool show_another_window = false;
        ImVec4 clear_color = ImVec4(89.0f / 255.0, 88.0f / 255.0, 87.0f /
            255.0, 1.00f);
        ImVec4 ray_color = ImVec4(1.0f, 1.0f, 1.0f, 1.0f);
        ImVec4 ray_color2 = ImVec4(0.5f, 0.5f, 0.5f, 1.0f);

        // Main loop
        int i = 0;
        bool flag = true;
        std::list<ImVec2> wave_tail;
        std::list<int> test;


        while (!glfwWindowShouldClose(window) && flag == true)
        {
                glfwPollEvents();

                ImGui_ImplOpenGL3_NewFrame();
                ImGui_ImplGlfw_NewFrame();

                ImGui::NewFrame();

                ImGuiWindowFlags window_flags = 0;
                ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_FirstUseEver)
```

```
    ;
ImGui::SetNextWindowSize(ImVec2(w, h), ImGuiCond_FirstUseEver
    );
window_flags |= ImGuiWindowFlags_NoMove;
window_flags |= ImGuiWindowFlags_NoResize;
window_flags |= ImGuiWindowFlags_NoCollapse;
window_flags |= ImGuiWindowFlags_NoBackground;
window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus;

ImGui::Begin("Simulation", nullptr, window_flags);
ImDrawList *draw_list = ImGui::GetWindowDrawList();

float strokex = 600.0; // the hyperbola' center x-coordinate
float strokey = 410.0; // the hyperbola' center y-coordinate

static int alpha = 5;
static int beta = 3;

ImVec2 wave_N1 = {sinhf((float)i)*alpha + strokex, coshf((
    float)i)*beta + strokey};
ImVec2 wave_N2 = {sinhf((float)i)*alpha + strokex, -coshf((
    float)i)*beta + strokey};
ImVec2 wave_N3 = {-sinhf((float)i)*alpha + strokex, coshf((
    float)i)*beta + strokey};
ImVec2 wave_N4 = {-sinhf((float)i)*alpha + strokex, -coshf((
    float)i)*beta + strokey};

wave_tail.push_front(wave_N1);
wave_tail.push_front(wave_N2);
wave_tail.push_front(wave_N3);
wave_tail.push_front(wave_N4);

// This for loop is the one drawing the hyperbola that is
    being drawn
for (auto &it : wave_tail)
{
       draw_list->AddCircleFilled(it, 6.0, IM_COL32(65, 0,
           255, 255));
}

i++;
// the if statement to restart the drawing after variable i
    reach 1001
if (i > 100)
{
       wave_tail.erase(wave_tail.begin(), wave_tail.end());
       i = 0;
}
```

```
                // To draw the hyperbola with explicit function
                struct Funcs
                {
                        static float Hyperbola1(void*, int i) { return sqrt(
                            pow(beta,2) * ((pow(-i,2) / pow(alpha,2)) + 1)) ;
                             }
                        static float Hyperbola2(void*, int i) { return - sqrt(
                             pow(beta,2) * ((pow(-i,2) / pow(alpha,2)) + 1))
                            ; }
                };
                static int func_type = 0, display_count = 50;
                ImGui::Combo("func", &func_type, "Hyperbola1\0Hyperbola2");
                float (*func)(void*, int) = (func_type == 0) ? Funcs::
                    Hyperbola1 : Funcs::Hyperbola2;

                // PlotLines (const char* label, float(*values_getter)(void*
                    data, int idx), void* data, int values_count,
                // int values_offset=0, const char* overlay_text = NULL,
                // float scale_min = FLT_MAX, float scale_max = FLT_MAX,
                    ImVec2 graph_size = (ImVec(0,100));
                ImGui::PlotLines("", func, NULL, display_count, -25, NULL,
                    -25.0f, 50.0f, ImVec2(0, 100));

                ImGui::PushItemWidth(200);
                ImGui::SliderInt("Alpha", &alpha, 1, 30);
                ImGui::SliderInt("Beta", &beta, 1, 30);
                ImGui::Text("Countdown: %d ", 100-i);
                ImGui::PopItemWidth();
                ImGui::End();

                // Rendering
                ImGui::Render();
                int display_w, display_h;
                glfwGetFramebufferSize(window, &display_w, &display_h);
                glViewport(0, 0, display_w, display_h);
                glClearColor(clear_color.x, clear_color.y, clear_color.z,
                    clear_color.w);
                glClear(GL_COLOR_BUFFER_BIT);
                ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

                glfwSwapBuffers(window);
        }

        termImgui();
        return 0;
}
```

**C++ Code 214:** *src/hyperbola.cpp "2D hyperbola type 2 simulation"*

To compile it, go to the directory **DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Hyperbola Type 2 Simulation/**, then type:

**mkdir build**
**cd build**
**cmake ..**
**make**
**./2D_Hyperbola_Type_2_Simulation**

**Figure 23.190:** *The simulation for hyperbola type 2 ($\frac{y^2}{\beta^2} - \frac{x^2}{\alpha^2} = 1$), on the top left, we can see the drawing of the hyperbola for positive $y = \sqrt{\left(\frac{x^2}{\alpha^2} + 1\right)\beta^2}$ (using explicit equation), while the main scene is showing cicles filled / that represent all the points for the parametric equation for hyperbola type 1, $(x, y) = (\alpha \sinh t, \beta \cosh t)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Hyperbola Type 2 Simulation/**).*

## v. C++ Plot and Computation: Identifying a Conic

(a) Identify the conic whose equation is $5x^2 - 4xy + 8y^2 - 36 = 0$ by rotating the $xy$-axes to put the conic in standard position.

(b) Find the angle $\theta$ through which you rotated the $xy$-axes in part(a)

**Solution:**

(a) The given equation can be written in the matrix form

$$\boldsymbol{x}^T A \boldsymbol{x} = 36$$

where

$$A = \begin{bmatrix} 5 & -2 \\ -2 & 8 \end{bmatrix}$$

The characteristic polynomial of $A$ is

$$\begin{vmatrix} \lambda - 5 & 2 \\ 2 & \lambda - 8 \end{vmatrix} = (\lambda - 4)(\lambda - 9)$$

so the eigenvalues are

$$\lambda_1 = 4$$
$$\lambda_2 = 9$$

The orthonormal bases for the eigenspaces are

$$\lambda_1 = 4 : \begin{bmatrix} \frac{2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{bmatrix}, \quad \lambda_2 = 9 : \begin{bmatrix} -\frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{bmatrix}$$

Thus, $A$ is orthogonally diagonalized by

$$P = \begin{bmatrix} \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix}$$

We need to make sure that $\det(P) = 1$, if it turns out that $\det(P) = -1$ then we need to interchange the columns to reverse the sign.

If $\det(P) = 1$ then the substitution $\boldsymbol{x} = P\boldsymbol{x'}$ performs a rotation of axes.

It follows that the equation of the conic in the $x'y'$-coordinate system is

$$\begin{bmatrix} x' & y' \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 9 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = 36$$

which we can write as

$$4x'^2 + 9y'^2 = 36$$
$$\frac{x'^2}{9} + \frac{y'^2}{4} = 1$$

this conic is an ellipse with $\alpha = 3$ and $\beta = 2$ in the $x'y'$-coordinate system.

(b) It follows that the transition matrix $P$ is

$$P = \begin{bmatrix} \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} \end{bmatrix} = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

which implies that

$$\begin{aligned} \cos\ \theta &= \frac{2}{\sqrt{5}} \\ \sin\ \theta &= \frac{1}{\sqrt{5}} \\ \tan\ \theta &= \frac{\sin\ \theta}{\cos\ \theta} = \frac{1}{2} \end{aligned}$$

Thus,

$$\theta = \tan^{-1}\left(\frac{1}{2}\right) \approx 26.6^0$$

In summary the processeses are:

1. Make $2 \times 2$ matrix $A$ from the quadratic form $ax^2 + bxy + cy^2 + d$, with $b \neq 0$.
2. Compute the eigenvalues for matrix $A$ ($\lambda_1$ and $\lambda_2$).
3. Compute the orthonormal eigenvectors ($\boldsymbol{v}_1$ and $\boldsymbol{v}_2$) for each eigenspaces of matrix $A$.
4. Create an orthogonal matrix $P$ consisting of the eigenvectors of matrix $A$ as the column vectors.
5. If $\det(P) = 1$ then we can proceed to the next stage, but if $\det(P) = -1$ then we need to interchange the column to make the determinant to have value of 1.
6. With the formula

   $$\boldsymbol{x}^{\boldsymbol{T}} A\boldsymbol{x} = k$$

   that can be written in quadratic form as $ax^2 + bxy + cy^2 + d$, we can determine the equation of conic in the $x'y'$-coordinate system with this formula

   $$\boldsymbol{x'}^{\boldsymbol{T}} D\boldsymbol{x'} = k$$

   or in quadratic form:

   $$\lambda_1 x'^2 + \lambda_2 y'^2 = k$$

7. We can determine the angle $\theta$ that rotate the $xy$-axes by using the transition matrix $P$

   $$P = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix} = \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

8. After we obtain $\theta$, the new $x'y'$-coordinate system can be obtained after we rotate $xy$-axes (the standard axes from Euclidean coordinate system) counterclockwise through an angle of $\theta$.

**Figure 23.191:** *The plot of a conic with implicit equation of* $5x^2 - 4xy + 8y^2 - 36 = 0$ *and the second conic with equation of* $4x'^2 + 9y'^2 = 36$*. The first conic is the counterclockwise rotation of the second conic with angle of* $\theta$*, with* $\alpha = 3$ *and* $\beta = 2$ *(**DFSimulatorC/Source Codes/Julia/ch23-2dplotimplicitequation2ellipses.jl**).*

For the C++ code, we are using **Armadillo** to compute the eigenvalues and the orthonormal eigenvectors that will become the column vectors for matrix $P$. Then we use **SymbolicC++** to compute the symbolic equation of the conic in the $x'y'$-coordinate system. The plotting will use **Gnuplot**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include <boost/tuple/tuple.hpp>
#include "symbolicc++.h"

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

#include "gnuplot−iostream.h"

using namespace std;
using namespace arma;

double halving(double x)
{
```

```
        double y = 2;
        return x/y;
}

// Driver code
int main(int argc, char** argv)
{
        arma::mat A(2,2, fill::zeros);
        cx_mat eigvec;
        cx_vec eigval;

        mat I(2,2,fill::eye);
        // A conic with equation 5x^2 - 4xy + 8y^2 - 36 = 0
        // We create symmetric matrix A from quadratic form Q
        int N = 2;
        float matrixA[N][N] = {};

        matrixA[0][0] = 5;
        matrixA[0][1] = halving(-4);
        matrixA[1][0] = halving(-4);
        matrixA[1][1] = 8;

        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<N; ++j)
                {
                        A[i+j*N] = matrixA[i][j];
                }
        }

        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        mat A1 = I*real(eigval[0]) - A;
        mat A2 = I*real(eigval[1]) - A;
```

```
cout << "Orthonormal Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}

// Save each orthonormal eigenvectors as a vector with Armadillo
arma::vec v1 = real(eigvec.col(0));
arma::vec v2 = real(eigvec.col(1));

mat P = real(eigvec);
mat P_transpose = P.t();
mat D = P_transpose*A*P;

float matrixD[N][N] = {};
for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              matrixD[i][j] = D[i+j*N];
       }
}

cout <<"Matrix P (which orthogonally diagonalizes A):" << "\n" << P
    <<endl;
cout <<"det(P):" << "\n" << det(P) <<endl;
cout <<"\nMatrix P^{T} * A * P:" << "\n" << D <<endl;

Symbolic x2("x2"), y2("y2");

Symbolic yt = ( ( x2 ), (y2) );
Symbolic y =yt.transpose();

float d11 = lround(matrixD[0][0]);
float d22 = lround(matrixD[1][1]);
float d12 = lround(matrixD[0][1]);

Symbolic MatrixD = ( (Symbolic(d11), Symbolic(d12) ),
( Symbolic(d12), Symbolic(d22) ) );

cout << "x' = " << y << endl;
cout << "x'^{T} = " << yt << endl;
cout << "D = " << MatrixD << endl;
cout << "x'^{T} * D * x' = " << yt*MatrixD*y << " = 36 " << endl;

cout <<"\nThe angle which rotate the xy-axes" <<endl;
cout <<"theta = " << atan(P[1]/P[0])*RADTODEG <<endl;
```

```
        // Plotting time
        Gnuplot gp;

        // We use a separate container for each column, like so:
        std::vector<double> pts_B_x;
        std::vector<double> pts_B_y;
        std::vector<double> pts_B_dx;
        std::vector<double> pts_B_dy;
        std::vector<double> pts_C_x;
        std::vector<double> pts_C_y;
        std::vector<double> pts_C_dx;
        std::vector<double> pts_C_dy;
        std::vector<double> pts_D_x;
        std::vector<double> pts_D_y;
        std::vector<double> pts_D_dx;
        std::vector<double> pts_D_dy;

        float o = 0;

        // Create eigenvector v1 with origin at (0,0)
        pts_B_x .push_back(o);
        pts_B_y .push_back(o);
        pts_B_dx.push_back(-v1[0]);
        pts_B_dy.push_back(-v1[1]);
        // Create eigenvector v2 with origin at (0,0)
        pts_C_x .push_back(o);
        pts_C_y .push_back(o);
        pts_C_dx.push_back(-v2[0]);
        pts_C_dy.push_back(-v2[1]);

        // Don't forget to put "\n" at the end of each line!
        gp << "set parametric\n";
        //gp << "unset key\n";
        gp << "set zeroaxis lw 2\n";
        gp << "set xrange [-3.3:3.3]\nset yrange [-2.3:2.3]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "plot 3*cos(t),2*sin(t) title '(3 cos t, 2 sin t)' lt 3 lc rgb
             'dark-turquoise',\
        '-' with vectors title 'v_{1}' lw 3, '-' with vectors title 'v_{2}'
             lw 3\n";
        gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx, pts_B_dy));
        gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx, pts_C_dy));

        return 0;
    }
```

**C++ Code 215:** *main.cpp "Identifying a conic"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- For the plotting part, to draw an implcit function, we have to use a parametric representation, or the function is once defined as a 3-D surface, and find an intersection curve with z=0 plane.

  If the function can be defined by using parameters, it is easy to make a figure. For example, an equation for circle is

  $$x^2 + y^2 = r^2$$

  and can be re-defined with parametric representation by

  $$(x, y) = (r\ \cos\ t, r\ \sin\ t)$$

  In **gnuplot**, the dummy variable is **t** for curves, and **u / v** for surfaces.

```
gp << "set parametric\n";
//gp << "unset key\n";
gp << "set zeroaxis lw 2\n";
gp << "set xrange [-3.3:3.3]\nset yrange [-2.3:2.3]\n";
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
// '-' means read from stdin. The send1d() function sends data
     to gnuplot's stdin.
gp << "plot 3*cos(t),2*sin(t) title '(3 cos t, 2 sin t)' lt 3
    lc rgb 'dark-turquoise',\
'-' with vectors title 'v_{1}' lw 3, '-' with vectors title '
    v_{2}' lw 3\n";
gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_dx,
    pts_B_dy));
gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_dx,
    pts_C_dy));
```

```
Matrix A:
   5.0000  -2.0000
  -2.0000   8.0000

Eigenvalues:
λ_1 = 4 + 0i
λ_2 = 9 + 0i

Orthonormal Eigenvectors:
x[λ_1] :
   -0.8944
  -0.4472

x[λ_2] :
    0.4472
  -0.8944

Matrix P (which orthogonally diagonalizes A):
  -0.8944   0.4472
  -0.4472  -0.8944

det(P):
1

Matrix P^{T} * A * P:
   4.0000e+00  -2.2204e-16
            0   9.0000e+00

x' =
[x2]
[y2]

x'^{T} =
[x2 y2]

D =
[4 0]
[0 9]

x'^{T} * D * x' = 4*x2^(2)+9*y2^(2) = 36

The angle which rotate the xy-axes
θ = 26.5651
```

**Figure 23.192:** *The computation to obtain the equation of the conic in the $x'y'$-coordinate system and the angle through which we rotate the xy-axes. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Principal Axes of the Ellipse with Armadillo/**).*

**Figure 23.193:** *The plot of the conic that is an ellipse with $\alpha = 3$ and $\beta = 2$, or in parametric form $(3\cos\ t, 2\ \sin\ t)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Principal Axes of the Ellipse with Armadillo/**).*

## vi. C++ Plot and Computation: Ellipsoid in $\mathbb{R}^3$

In $\mathbb{R}^2$ the shape that represents the equation

$$\frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} = 1$$

is called an ellipse.

In $\mathbb{R}^3$ the shape that represents the equation

$$\frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} + \frac{z^2}{\gamma^2} = 1$$

or

$$ax^2 + by^2 + cx^2 = 1$$

with $a = \frac{1}{\alpha}, b = \frac{1}{\beta}, c = \frac{1}{\gamma}$, and $\alpha, \beta, \gamma$ are the length of the semi-axes, thus the equation above is called an ellipsoid.

Ellipsoid is a surface that can be obtained from a sphere by deforming it by means of directional scalings, or more generally, of an affine transformation.

The parametric equation for ellipsoid (measuring from the equator):

$$\begin{aligned} x &= a\ \cos(\theta)\ \cos(\phi) \\ y &= b\ \cos(\theta)\ \sin(\phi) \\ z &= c\ \sin(\theta) \end{aligned} \tag{23.226}$$

with $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ and $0 \leq \phi \leq 2\pi$. $\theta$ is the reduced latitude, parametric latitude, or eccentric anomaly and $\phi$ is azimuth or longitude. A sphere can also be obtained from the parametric equation above with $a = b = c$.

In computer graphics, it is easier to create a 3 dimensional shape with parametric equation, just like for 2 dimensional, we can draw / plot a circle easier with $(\cos\ t, \sin\ t)$.

For the C++ code, we are going to use **Gnuplot** and **OpenGL**, it is very straightforward and easy to use **Gnuplot**, the terminal **wxt** from **gnuplot** can really help us to look around the 3 dimensional space. While the **OpenGL** code will be very hard and has steeper learning curve, at least you will have to memorize tons of terminologies like Vertex Buffer Object (VBO), Index Buffer Object (IBO), **GLM** library, vertex shader, fragment shader, and so on. But with steeper learning curve, the result is fantastic, more interactive, more details, if the result can be projected into this reality it can be modified by our hands directly like Jarvis simulation.

Both the **Gnuplot** and **OpenGL** codes are using the parametric equation to be able to draw the beautiful ellipsoid, and even make a simulation of the ellipsoid.

In this section we are only talking about the **gnuplot** code. The next section will be for **OpenGL**.

We are going to plot multiple ellipsoid / ellipsoids, with centerfor each ellipsoid is defined in **ellipses.dat**. To make a very good ellipsoid, we use parametric funciton for ellipsoid, instead of explicit or implicit function.

```
#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "datafile ='ellipses.dat' \n";
        gp << "stats datafile nooutput \n";
        gp << "num_rows = STATS_records \n";
        gp << "array centers_x[num_rows] \n";
        gp << "array centers_y[num_rows] \n";
        gp << "array centers_z[num_rows] \n";

        gp << "row = 1 \n";
        gp << "stats datafile using (centers_x[row]=$1,\
        centers_y[row]=$2,\
        centers_z[row]=$3,\
        row = row + 1) nooutput\n";

        gp << "Rx = 0.2 \n";
        gp << "Ry = 0.3 \n";
        gp << "Rz = 0.7 \n";

        gp << "set parametric\n"; // To make the plot interactive
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        //gp << "set hidden3d\n"; // to apply the right setting for
            intersecting surfaces
        gp << "set urange[-pi/2:pi/2]\n set vrange[0:2*pi]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
            axis'\n";
        gp << "set view 100,5,1\n"; // pitch,yaw,zoom
        gp << "rgb(r,g,b) = int(r)*65536 + int(g)*256 + int(b)\n";
        gp << "splot datafile using 1:2:3 title 'Ellipses' with points ps 2
            pt 7,\
        for [i=1:num_rows] centers_x[i] + Rx*cos(u)*cos(v),\
        centers_y[i] + Ry*cos(u)*sin(v),\
        centers_z[i] + Rz*sin(u) notitle\n";

        return 0;
}
```

**C++ Code 216:** *main.cpp "Ellipsoids in 3 dimensional space with gnuplot"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.194:** *The plot of the ellipsoids with center of each ellipsoid defined in* ***ellipsoids.dat*** *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot of Ellipsoids Parametric Function with Gnuplot from Textfile/**).*

## vii. C++ Simulation: Sphere and Ellipsoid in $\mathbb{R}^3$

In 3 dimensional simulation we need to choose a suitable model-view-projection (MVP) matrix to present the 3 dimensional data in a clear way. We can even play with thte lightning, this C++ code takes a very long time to comprehend if you are really learning C++ from zero and just reach this stage. We need multiple headers (**.h**) files and multiple C++ (**.cpp**) files.

**OpenGL** can really help to tackle the task along with **GLU** and **GLUT** for the keyboard and mouse click events.

The C++ code is originally obtained from:
**https://www.songho.ca/opengl/gl_sphere.html**
(Read the website, to comprehend more in details, to be able to create a sphere or even an ellipsoid with OpenGL needs more than 1000 lines of C++ codes)

With a minor modification, adding three parameters for the parametric equations, we can create ellipsoid from sphere.

The current working directory for this section/project has structure of

- bin
  - earth2048.bmp moon1024.bmp (this folder is used to store the texture image for the ellipsoid, and the end result / the bin file / executable file)
- src
  - **freeglut** folder
    (this folder is used to store freeglut library)
  - **glut** folder
    (this folder is used to store glut library)
  - Bmp.cpp (To handle the texture that will wrap the ellipsoid)
  - Bmp.h
  - main.cpp
  - Makefile
  - Sphere.cbp
  - Sphere.cpp (To create the ellipsoid with parameters defined here)
  - Sphere.h

```
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <iostream>
#include <sstream>
#include <iomanip>
#include <fstream>
#include "Bmp.h"
#include "Sphere.h"
```

```
// GLUT CALLBACK functions
void displayCB();
void reshapeCB(int w, int h);
void timerCB(int millisec);
void keyboardCB(unsigned char key, int x, int y);
void mouseCB(int button, int stat, int x, int y);
void mouseMotionCB(int x, int y);

void initGL();
int initGLUT(int argc, char **argv);
bool initSharedMem();
void clearSharedMem();
void initLights();
void setCamera(float posX, float posY, float posZ, float targetX, float
    targetY, float targetZ);
void drawString(const char *str, int x, int y, float color[4], void *font);
void drawString3D(const char *str, float pos[3], float color[4], void *font
    );
void toOrtho();
void toPerspective();
GLuint loadTexture(const char* fileName, bool wrap=true);


// constants
const int SCREEN_WIDTH = 1600;
const int SCREEN_HEIGHT = 700;
const float CAMERA_DISTANCE = 6.0f;
const int TEXT_WIDTH = 8;
const int TEXT_HEIGHT = 13;


// global variables
void *font = GLUT_BITMAP_8_BY_13;
int screenWidth;
int screenHeight;
bool mouseLeftDown;
bool mouseRightDown;
bool mouseMiddleDown;
float mouseX, mouseY;
float cameraAngleX;
float cameraAngleY;
float cameraDistance;
int drawMode;
GLuint texId;
int imageWidth;
int imageHeight;
```

```
// sphere: min sector = 2, min stack = 2
Sphere sphere1(1.0f, 36, 18, false, 2); // radius, sectors, stacks, non-
    smooth (flat) shading, Y-up
Sphere sphere2(1.0f, 36, 18, true, 2); // radius, sectors, stacks, smooth(
    default), Y-up



///////////////////////////////////////
int main(int argc, char **argv)
{
       // init global vars
       initSharedMem();

       // init GLUT and GL
       initGLUT(argc, argv);
       initGL();

       // load BMP image
       texId = loadTexture("earth2048.bmp", true);
       //texId = loadTexture("ps2imageFreya.bmp", true);

       // the last GLUT call (LOOP)
       // window will be shown and display callback is triggered by events
       // NOTE: this call never return main().
       glutMainLoop(); /* Start GLUT event-processing loop */

       return 0;
}



////////////////////////////////////////////
// initialize GLUT for windowing
////////////////////////////////////////////
int initGLUT(int argc, char **argv)
{
       // GLUT stuff for windowing
       // initialization openGL window.
       // it is called before any other GLUT routine
       glutInit(&argc, argv);

       glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH |
           GLUT_STENCIL); // display mode

       glutInitWindowSize(screenWidth, screenHeight); // window size
```

```
        glutInitWindowPosition(100, 100); // window location

        // finally, create a window with openGL context
        // Window will not displayed until glutMainLoop() is called
        // it returns a unique ID
        int handle = glutCreateWindow(argv[0]); // param is the title of
            window

        // register GLUT callback functions
        glutDisplayFunc(displayCB);
        glutTimerFunc(33, timerCB, 33); // redraw only every given millisec
        glutReshapeFunc(reshapeCB);
        glutKeyboardFunc(keyboardCB);
        glutMouseFunc(mouseCB);
        glutMotionFunc(mouseMotionCB);

        return handle;
}



////////////////////////////////////
// initialize OpenGL
// disable unused features
////////////////////////////////////
void initGL()
{
        glShadeModel(GL_SMOOTH); // shading mathod: GL_SMOOTH or GL_FLAT
        glPixelStorei(GL_UNPACK_ALIGNMENT, 4); // 4-byte pixel alignment

        // enable /disable features
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
        glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
        //glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_TEXTURE_2D);
        glEnable(GL_CULL_FACE);

        // track material ambient and diffuse from surface color, call it
            before glEnable(GL_COLOR_MATERIAL)
        //glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
        //glEnable(GL_COLOR_MATERIAL);

        glClearColor(0, 0, 0, 0); // background color
        glClearStencil(0); // clear stencil buffer
        glClearDepth(1.0f); // 0 is near, 1 is far
        glDepthFunc(GL_LEQUAL);
```

```
        initLights();
}



///////////////////////////////////
// write 2d text using GLUT
// The projection matrix must be set to orthogonal before call this
    function.
///////////////////////////////////
void drawString(const char *str, int x, int y, float color[4], void *font)
{
        glPushAttrib(GL_LIGHTING_BIT | GL_CURRENT_BIT); // lighting and
            color mask
        glDisable(GL_LIGHTING); // need to disable lighting for proper text
            color
        glDisable(GL_TEXTURE_2D);

        glColor4fv(color); // set text color
        glRasterPos2i(x, y); // place text position

        // loop all characters in the string
        while(*str)
        {
                glutBitmapCharacter(font, *str);
                ++str;
        }

        glEnable(GL_TEXTURE_2D);
        glEnable(GL_LIGHTING);
        glPopAttrib();
}



////////////////////////////////
// draw a string in 3D space
////////////////////////////////
void drawString3D(const char *str, float pos[3], float color[4], void *font
    )
{
        glPushAttrib(GL_LIGHTING_BIT | GL_CURRENT_BIT); // lighting and
            color mask
        glDisable(GL_LIGHTING); // need to disable lighting for proper text
            color
        glDisable(GL_TEXTURE_2D);
```

```
        glColor4fv(color); // set text color
        glRasterPos3fv(pos); // place text position

        // loop all characters in the string
        while(*str)
        {
            glutBitmapCharacter(font, *str);
            ++str;
        }

        glEnable(GL_TEXTURE_2D);
        glEnable(GL_LIGHTING);
        glPopAttrib();
    }



    ///////////////////////////////////
    // initialize global variables
    ///////////////////////////////////
    bool initSharedMem()
    {
        screenWidth = SCREEN_WIDTH;
        screenHeight = SCREEN_HEIGHT;

        mouseLeftDown = mouseRightDown = mouseMiddleDown = false;
        mouseX = mouseY = 0;

        cameraAngleX = cameraAngleY = 0.0f;
        cameraDistance = CAMERA_DISTANCE;

        drawMode = 0; // 0:fill, 1: wireframe, 2:points

        // change up axis to +Y
        //sphere1.setUpAxis(2);
        //sphere2.setUpAxis(2);

        // debug
        sphere2.printSelf();

        return true;
    }



    ////////////////////////////////////////
    // clean up global vars
    ////////////////////////////////////////
```

```
void clearSharedMem()
{
}



///////////////////////////////////////
// initialize lights
///////////////////////////////////////
void initLights()
{
       // set up light colors (ambient, diffuse, specular)
       GLfloat lightKa[] = {.3f, .3f, .3f, 1.0f}; // ambient light
       GLfloat lightKd[] = {.7f, .7f, .7f, 1.0f}; // diffuse light
       GLfloat lightKs[] = {1, 1, 1, 1}; // specular light
       glLightfv(GL_LIGHT0, GL_AMBIENT, lightKa);
       glLightfv(GL_LIGHT0, GL_DIFFUSE, lightKd);
       glLightfv(GL_LIGHT0, GL_SPECULAR, lightKs);

       // position the light
       float lightPos[4] = {0, 0, 1, 0}; // directional light
       glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

       glEnable(GL_LIGHT0); // MUST enable each light source after
           configuration
}



///////////////////////////////////////
// set camera position and lookat direction
///////////////////////////////////////
void setCamera(float posX, float posY, float posZ, float targetX, float
    targetY, float targetZ)
{
       glMatrixMode(GL_MODELVIEW);
       glLoadIdentity();
       gluLookAt(posX, posY, posZ, targetX, targetY, targetZ, 0, 1, 0); //
           eye(x,y,z), focal(x,y,z), up(x,y,z)
}



///////////////////////////////////////
// load raw image as a texture
///////////////////////////////////////
GLuint loadTexture(const char* fileName, bool wrap)
{
```

```
Image::Bmp bmp;
if(!bmp.read(fileName))
return 0; // exit if failed load image

// get bmp info
int width = bmp.getWidth();
int height = bmp.getHeight();
const unsigned char* data = bmp.getDataRGB();
GLenum type = GL_UNSIGNED_BYTE; // only allow BMP with 8-bit per
    channel

// We assume the image is 8-bit, 24-bit or 32-bit BMP
GLenum format;
int bpp = bmp.getBitCount();
if(bpp == 8)
format = GL_LUMINANCE;
else if(bpp == 24)
format = GL_RGB;
else if(bpp == 32)
format = GL_RGBA;
else
return 0; // NOT supported, exit

// gen texture ID
GLuint texture;
glGenTextures(1, &texture);

// set active texture and configure it
glBindTexture(GL_TEXTURE_2D, texture);

// select modulate to mix texture with color for shading
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
//glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);

// if wrap is true, the texture wraps over at the edges (repeat)
// ... false, the texture ends at the edges (clamp)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap ? GL_REPEAT :
     GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap ? GL_REPEAT :
     GL_CLAMP);
//glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE
    );
//glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE
    );
```

```
        // copy texture data
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
            type, data);
        //glGenerateMipmap(GL_TEXTURE_2D);

        // build our texture mipmaps
        switch(bpp)
        {
                case 8:
                gluBuild2DMipmaps(GL_TEXTURE_2D, 1, width, height,
                    GL_LUMINANCE, type, data);
                break;
                case 24:
                gluBuild2DMipmaps(GL_TEXTURE_2D, 3, width, height, GL_RGB,
                    type, data);
                break;
                case 32:
                gluBuild2DMipmaps(GL_TEXTURE_2D, 4, width, height, GL_RGBA,
                    type, data);
                break;
        }

        bmp.printSelf();
        return texture;
}



////////////////////////////////////
// display info messages
////////////////////////////////////
void showInfo()
{
        // backup current model-view matrix
        glPushMatrix(); // save current modelview matrix
        glLoadIdentity(); // reset modelview matrix

        // set to 2D orthogonal projection
        glMatrixMode(GL_PROJECTION); // switch to projection matrix
        glPushMatrix(); // save current projection matrix
        glLoadIdentity(); // reset projection matrix
        //gluOrtho2D(0, screenWidth, 0, screenHeight); // set to orthogonal
            projection
        glOrtho(0, screenWidth, 0, screenHeight, -1, 1); // set to
            orthogonal projection

        float color[4] = {1, 1, 1, 1};
```

```
        std::stringstream ss;
        ss << std::fixed << std::setprecision(3);

        // parameter for the ellipsoid,
        float a = 1.2; // the ellipsoid' x axis has length of 2a
        float b = 0.6; // the ellipsoid' y axis has length of 2b
        float c = 0.5; // the ellipsoid' z axis has length of 2c

        ss << "Sphere Radius: " << sphere2.getRadius() << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-TEXT_HEIGHT, color,
            font);
        ss.str("");

        ss << "Ellipsoid' parameter a: " << a << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(2*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Ellipsoid' parameter b: " << b << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(3*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Ellipsoid' parameter c: " << c << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(4*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Sector Count: " << sphere2.getSectorCount() << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(5*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Stack Count: " << sphere2.getStackCount() << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(6*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Vertex Count: " << sphere2.getVertexCount() << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(7*TEXT_HEIGHT), color
            , font);
        ss.str("");

        ss << "Index Count: " << sphere2.getIndexCount() << std::ends;
        drawString(ss.str().c_str(), 1, screenHeight-(8*TEXT_HEIGHT), color
            , font);
        ss.str("");
```

```
        // unset floating format
        ss << std::resetiosflags(std::ios_base::fixed | std::ios_base::
            floatfield);

        // restore projection matrix
        glPopMatrix(); // restore to previous projection matrix

        // restore modelview matrix
        glMatrixMode(GL_MODELVIEW); // switch to modelview matrix
        glPopMatrix(); // restore to previous modelview matrix
    }



    ////////////////////////////////////
    // set projection matrix as orthogonal
    ////////////////////////////////////
    void toOrtho()
    {
        // set viewport to be the entire window
        glViewport(0, 0, (GLsizei)screenWidth, (GLsizei)screenHeight);

        // set orthographic viewing frustum
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0, screenWidth, 0, screenHeight, -1, 1);

        // switch to modelview matrix in order to set scene
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }



    ////////////////////////////////////////
    // set the projection matrix as perspective
    ////////////////////////////////////////
    void toPerspective()
    {
        // set viewport to be the entire window
        glViewport(0, 0, (GLsizei)screenWidth, (GLsizei)screenHeight);

        // set perspective viewing frustum
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(40.0f, (float)(screenWidth)/screenHeight, 1.0f,
            1000.0f); // FOV, AspectRatio, NearClip, FarClip
```

```
        // switch to modelview matrix in order to set scene
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}









//=====================================
// CALLBACKS
//=====================================

void displayCB()
{
        // clear buffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
            GL_STENCIL_BUFFER_BIT);

        // save the initial ModelView matrix before modifying ModelView
            matrix
        glPushMatrix();

        // tramsform modelview matrix
        glTranslatef(0, 0, -cameraDistance);

        // set material
        float ambient[] = {0.5f, 0.5f, 0.5f, 1};
        float diffuse[] = {0.7f, 0.7f, 0.7f, 1};
        float specular[] = {1.0f, 1.0f, 1.0f, 1};
        float shininess = 128;
        glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
        glMaterialf(GL_FRONT, GL_SHININESS, shininess);

        // line color
        float lineColor[] = {0.2f, 0.2f, 0.2f, 1};

        // draw left flat sphere with lines
        glPushMatrix();
        glTranslatef(-2.5f, 0, 0);
        glRotatef(cameraAngleX, 1, 0, 0); // pitch
```

```
        glRotatef(cameraAngleY, 0, 1, 0); // heading
        glBindTexture(GL_TEXTURE_2D, 0);
        sphere1.drawWithLines(lineColor);
        //sphere1.drawLines(lineColor);
        glPopMatrix();

        // draw centre smooth sphere with line
        glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse); // reset diffuse
        glPushMatrix();
        glRotatef(cameraAngleX, 1, 0, 0);
        glRotatef(cameraAngleY, 0, 1, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
        sphere2.drawWithLines(lineColor);
        glPopMatrix();

        // draw right sphere with texture
        glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse); // reset diffuse
        glPushMatrix();
        glTranslatef(2.5f, 0, 0);
        glRotatef(cameraAngleX, 1, 0, 0);
        glRotatef(cameraAngleY, 0, 1, 0);
        glBindTexture(GL_TEXTURE_2D, texId);
        sphere2.draw();
        glPopMatrix();

        /*
        // using GLU quadric object
        GLUquadricObj* obj = gluNewQuadric();
        gluQuadricDrawStyle(obj, GLU_FILL); // GLU_FILL, GLU_LINE,
            GLU_SILHOUETTE, GLU_POINT
        gluQuadricNormals(obj, GL_SMOOTH);
        gluQuadricTexture(obj, GL_TRUE);
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        gluSphere(obj, 2.0, 50, 50); // radius, slice, stack
        */

        glBindTexture(GL_TEXTURE_2D, 0);

        showInfo(); // print max range of glDrawRangeElements

        glPopMatrix();

        glutSwapBuffers();
    }


    void reshapeCB(int w, int h)
```

```
{
        screenWidth = w;
        screenHeight = h;
        toPerspective();
        std::cout << "window resized: " << w << " x " << h << std::endl;

        #ifdef _WIN32
        HWND handle = ::GetActiveWindow();
        RECT rect;
        ::GetWindowRect(handle, &rect); // with non-client area; border,
            titlebar etc.
        std::cout << "========================" << std::endl;
        std::cout << "full window size with border: " << (rect.right - rect
            .left) << "x" << (rect.bottom - rect.top) << std::endl;
        ::GetClientRect(handle, &rect); // only client dimension
        std::cout << "client window size: " << (rect.right - rect.left) <<
            "x" << (rect.bottom - rect.top) << std::endl;
        std::cout << "========================" << std::endl;
        #endif
}


void timerCB(int millisec)
{
        glutTimerFunc(millisec, timerCB, millisec);
        glutPostRedisplay();
}


void keyboardCB(unsigned char key, int x, int y)
{
        switch(key)
        {
                case 27: // ESCAPE
                clearSharedMem();
                exit(0);
                break;

                case 'd': // switch rendering modes (fill -> wire -> point)
                case 'D':
                ++drawMode;
                drawMode %= 3;
                if(drawMode == 0) // fill mode
                {
                        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                        glEnable(GL_DEPTH_TEST);
                        glEnable(GL_CULL_FACE);
                }
```

```
                else if(drawMode == 1) // wireframe mode
                {
                        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
                        glDisable(GL_DEPTH_TEST);
                        glDisable(GL_CULL_FACE);
                }
                else // point mode
                {
                        glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
                        glDisable(GL_DEPTH_TEST);
                        glDisable(GL_CULL_FACE);
                }
                break;

                case ' ':
                sphere1.reverseNormals();
                sphere2.reverseNormals();
                break;

                default:
                ;
        }
}


void mouseCB(int button, int state, int x, int y)
{
        mouseX = x;
        mouseY = y;

        if(button == GLUT_LEFT_BUTTON)
        {
                if(state == GLUT_DOWN)
                {
                        mouseLeftDown = true;
                }
                else if(state == GLUT_UP)
                mouseLeftDown = false;
        }

        else if(button == GLUT_RIGHT_BUTTON)
        {
                if(state == GLUT_DOWN)
                {
                        mouseRightDown = true;
                }
                else if(state == GLUT_UP)
                mouseRightDown = false;
```

```
        }

        else if(button == GLUT_MIDDLE_BUTTON)
        {
                if(state == GLUT_DOWN)
                {
                        mouseMiddleDown = true;
                }
                else if(state == GLUT_UP)
                mouseMiddleDown = false;
        }
}


void mouseMotionCB(int x, int y)
{
        if(mouseLeftDown)
        {
                cameraAngleY += (x - mouseX);
                cameraAngleX += (y - mouseY);
                mouseX = x;
                mouseY = y;
        }
        if(mouseRightDown)
        {
                cameraDistance -= (y - mouseY) * 0.2f;
                mouseY = y;
        }
}
```

**C++ Code 217:** *src/main.cpp "3D Simulation of ellipsoid and sphere"*

To compile it, go to the source code working directory for this section, then type:
**cd /src**
**make**
**cd ..**
**cd /bin**
**cd ./sphere**

**Figure 23.195:** *The simulation of a sphere (on the left) and ellipsoid (in the middle) and an ellipsoid wrapped with texture image (on the right) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Simulation of Ellipsoid with OpenGL GLUT/**).*

## viii. C++ Computation: Finding Constrained Extrema

Find the maximum and minimum values of the quadratic form

$$z = 5x^2 + 5y^2 + 4xy$$

subject to the constraint $x^2 + y^2 = 1$.

**Solution:**
The quadratic form can be expressed in matrix notation as

$$z = 5x^2 + 5y^2 + 4xy = \boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 5 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The eigenvalues of $A$ and its corresponding orthonormal eigenvectors are

$$\lambda_1 = 7 : \boldsymbol{v}_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad \lambda_2 = 3 : \boldsymbol{v}_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Thus, the constrained extrema are:
**constrained maximum**: $z = 7$ at $(x, y) = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$

**constrained minimum**: $z = 3$ at $(x, y) = \left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$

The C++ code is using **Armadillo** for the computation to obtain the eigenvalues and orthonormal eigenvectors. The eigenvalues will represent the maximum and minimum values for the quadratic form.

```
#include <iostream>
```

```
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

double halving(double x)
{
       double y = 2;
       return x/y;
}

double division(double x, double y)
{
       return x/y;
}


// Driver code
int main(int argc, char** argv)
{
       mat A(2,2, fill::zeros);
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
       // We create symmetric matrix A from quadratic form Q
       int N = 2;
       float matrixA[N][N] = {};

       matrixA[0][0] = 5;
       matrixA[0][1] = halving(4);
       matrixA[1][0] = matrixA[0][1];
       matrixA[1][1] = 5;

       for (int i = 0; i < N; ++i)
       {
              for(int j = 0; j<N; ++j)
              {
                     A[i+j*N] = matrixA[i][j];
              }
       }

       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix
```

```
        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
            cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        mat A1 = I*real(eigval[0]) - A;
        mat A2 = I*real(eigval[1]) - A;

        cout << "Orthonormal Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
            cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                endl;
        }

        // Save each orthonormal eigenvectors as a vector with Armadillo
        arma::vec v1 = real(eigvec.col(0));
        arma::vec v2 = real(eigvec.col(1));

        if ( real(eigval[0]) < real(eigval[1]))
        {
            cout <<"Constrained maximum: z = " << real(eigval[1]) <<endl;
            cout <<"at (x,y) = " << "\n" << real(eigvec.col(1)) << endl;
            cout <<"Constrained minimum: z = " << real(eigval[0]) <<endl;
            cout <<"at (x,y) = " << "\n" << real(eigvec.col(0)) << endl;
        }
        else
        {
            cout <<"Constrained maximum: z = " << real(eigval[0]) <<endl;
            cout <<"at (x,y) = " << "\n" << real(eigvec.col(0)) << endl;
            cout <<"Constrained minimum: z = " << real(eigval[1]) <<endl;
            cout <<"at (x,y) = " << "\n" << real(eigvec.col(1)) << endl;
        }

        return 0;
    }
```

**C++ Code 218:** *main.cpp "Finding constrained extrema"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The **if** conditional statement is used to show constrained maximum at the eigenvalue with highest value compared to other eigenvalues.

  Then to show constrained minimum at the eigenvalue with lowest value compared to other eigenvalues.

```
            if ( real(eigval[0]) < real(eigval[1]))
            {
                   cout <<"Constrained maximum: z = " << real(eigval[1]) <<
                       endl;
                   cout <<"at (x,y) = " << "\n" << real(eigvec.col(1)) <<
                       endl;
                   cout <<"Constrained minimum: z = " << real(eigval[0]) <<
                       endl;
                   cout <<"at (x,y) = " << "\n" << real(eigvec.col(0)) <<
                       endl;
            }
            else
            {
                   cout <<"Constrained maximum: z = " << real(eigval[0]) <<
                       endl;
                   cout <<"at (x,y) = " << "\n" << real(eigvec.col(0)) <<
                       endl;
                   cout <<"Constrained minimum: z = " << real(eigval[1]) <<
                       endl;
                   cout <<"at (x,y) = " << "\n" << real(eigvec.col(1)) <<
                       endl;
            }
```

```
Matrix A:
   5.0000   2.0000
   2.0000   5.0000

Eigenvalues:
λ_1 = 7 + 0i
λ_2 = 3 + 0i

Orthonormal Eigenvectors:
x[λ_1] :
    0.7071
   0.7071

x[λ_2] :
   -0.7071
   0.7071

Constrained maximum: z = 7
at (x,y) =
   0.7071
   0.7071

Constrained minimum: z = 3
at (x,y) =
  -0.7071
   0.7071
```

**Figure 23.196:** *The computation to obtain the constrained extrema from quadratic form $z = 5x^2 + 5y^2 + 4xy$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Finding Constrained Extrema with Armadillo/**).*

## ix. C++ Computation: A Constrained Extremum Problem

A rectangle is to be inscribed in the ellipse $4x^2 + 9y^2 = 36$. Use eigenvalue methods to find nonnegative values of $x$ and $y$ that produce the inscribed rectangle with maximum area.

**Solution:**

The area $z$ of the inscribed rectangle is given by $z = 4xy$, so the problem is to maximize the quadratic form $z = 4xy$ subject to the constraint $4x^2 + 9y^2 = 36$. In this problem, the graph of the constraint equation is an ellipse rather than the unit circle.

We can rewrite the constraint as

$$\left(\frac{x}{3}\right)^2 + \left(\frac{y}{2}\right)^2 = 1$$

and defining new variables, $x_1$ and $y_1$, by the equations

$$x = 3x_1$$
$$y = 2y_1$$

This enables us to reformulate the problem as follows:

$$\text{maximize } z = 4xy = 24x_1y_1$$

subject to the constraint

$$x_1^2 + y_1^2 = 1$$

To solve this problem, we will write the quadratic form $z = 24x_1y_1$ as

$$z = \boldsymbol{x}^T A\boldsymbol{x} = \begin{bmatrix} x_1 & y_1 \end{bmatrix} \begin{bmatrix} 0 & 12 \\ 12 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

The largest eigenvalue of $A$ is $\lambda = 12$ and that the only corresponding unit eigenvector with nonnegative entries is

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Thus, the maximum area is $z = 12$, and this occurs when

$$x = 3x_1 = \frac{3}{\sqrt{2}}$$
$$y = 2y_1 = \frac{2}{\sqrt{2}}$$

The C++ code is using **Armadillo** for the computation to obtain the eigenvalues and orthonormal eigenvectors. The eigenvalues will represent the maximum and minimum values for the quadratic form, we need to give input of the variables $a$ and $b$ manually after we define new variables $x = ax_1$ and $y = by_1$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

double halving(double x)
{
       double y = 2;
       return x/y;
}

double division(double x, double y)
{
       return x/y;
}


// Driver code
// A rectangle inscribed in the ellipse 4x^2 + 9y^2=36
// the area of inscribed rectangle is z=4xy
// 4xy = 24 x1y1
// x = 3x1 = a x1
// y = 2y1 = b y1

// Maximize z = 4xy = 24 x1y1
// subject to the constraint: x1^2 + y1^2 = 1
int main(int argc, char** argv)
{
       float a = 3; // x = 3x1 = a x1
```

```
float b = 2; // y = 2y1 = b y1
mat A(2,2, fill::zeros);
cx_mat eigvec;
cx_vec eigval;

mat I(2,2,fill::eye);
// We create symmetric matrix A from quadratic form Q
int N = 2;
float matrixA[N][N] = {};

matrixA[0][0] = 0;
matrixA[0][1] = halving(24);
matrixA[1][0] = matrixA[0][1];
matrixA[1][1] = 0;

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              A[i+j*N] = matrixA[i][j];
       }
}

cout <<"Matrix A:" << "\n" << A <<endl;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
    square matrix

cout << "Eigenvalues:" << endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

cout << endl;
mat A1 = I*real(eigval[0]) - A;
mat A2 = I*real(eigval[1]) - A;

cout << "Orthonormal Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}
```

```
        // Save each orthonormal eigenvectors as a vector with Armadillo
        arma::vec v1 = real(eigvec.col(0));
        arma::vec v2 = real(eigvec.col(1));

        if ( real(eigval[0]) < real(eigval[1]))
        {
            cout <<"Constrained maximum: z = " << real(eigval[1]) <<endl;
            cout <<"at (x1,y1) = " << "\n" << real(eigvec.col(1)) << endl
                ;
            cout <<"when x= " << v2[0]*a << endl;
            cout <<"and y= " << v2[1]*b << endl;

            cout <<"\nConstrained minimum: z = " << real(eigval[0]) <<
                endl;
            cout <<"at (x1,y1) = " << "\n" << real(eigvec.col(0)) << endl
                ;
            cout <<"when x = " << v1[0]*a << endl;
            cout <<"and y = " << v1[1]*b << endl;
        }
        else
        {
            cout <<"Constrained maximum: z = " << real(eigval[0]) <<endl;
            cout <<"at (x1,y1) = " << "\n" << real(eigvec.col(0)) << endl
                ;
            cout <<"when x = " << v1[0]*a << endl;
            cout <<"and y = " << v1[1]*b << endl;

            cout <<"Constrained minimum: z = " << real(eigval[1]) <<endl;
            cout <<"at (x1,y1) = " << "\n" << real(eigvec.col(1)) << endl
                ;
            cout <<"when x = " << v2[0]*a << endl;
            cout <<"and y = " << v2[1]*b << endl;
        }

        return 0;
    }
```

**C++ Code 219:** *main.cpp "A constrained extremum problem"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A:
         0   12.0000
   12.0000         0

Eigenvalues:
λ_1 = 12 + 0i
λ_2 = -12 + 0i

Orthonormal Eigenvectors:
x[λ_1] :
    0.7071
   0.7071

x[λ_2] :
   -0.7071
   0.7071

Constrained maximum: z = 12
at (x1,y1) =
   0.7071
   0.7071

when x = 2.12132
and y = 1.41421
Constrained minimum: z = -12
at (x1,y1) =
  -0.7071
   0.7071

when x = -2.12132
and y = 1.41421
```

**Figure 23.197:** *The computation to obtain the constrained extremum of a rectangle incsribed in the ellipse $4x^2 + 9y^2 = 36$, the area $z$ of the inscribed rectangle is given by $z = 4xy$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Constrained Extremum Problem Rectangle Inscribed in the Ellipse with Armadillo/**).*

## x. C++ Simulation and Computation: Finding Constrained Extrema with Level Curves

We know that the maximum and minimum values of the quadratic form

$$z = 5x^2 + 5y^2 + 4xy$$

subject to the constraint $x^2 + y^2 = 1$ has the constrained maximum at $z = 7$ and it is attained at the points

$$(x, y) = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$
$$(x, y) = \left(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right)$$

and that the constrained minimum $z = 3$ is attained at the points

$$(x, y) = \left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$
$$(x, y) = \left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right)$$

Geometrically, this means that the level curve $5x^2 + 5y^2 + 4xy = 7$ should just touch the unit circle at the points $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ and $\left(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right)$, and the level curve $5x^2 + 5y^2 + 4xy = 3$ should just touch it at the points in $\left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ and $\left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right)$.

To compute the point of intersection between the ellipse with equation

$$5x^2 + 5y^2 + 4xy = 7$$

and the unit circle of radius 1 with equation

$$x^2 + y^2 = 1$$

we use algebra technique,

$$\begin{aligned} 5x^2 + 5y^2 + 4xy &= 7 \\ 5(x^2 + y^2 &= 1) \end{aligned}$$

Then we subtract the first equation with the second equation

$$\begin{aligned} 5x^2 + 5y^2 + 4xy - 5x^2 - 5y^2 &= 7 - 5 \\ 4xy &= 2 \\ 4xy - 2 &= 0 \end{aligned}$$

Thus all the $x$ and $y$ values that satisfy $4xy - 2 = 0$ are the point of intersections between the first ellipse and the unit circle.

For the second ellipse with equation

$$5x^2 + 5y^2 + 4xy = 3$$

and the unit circle of radius 1 with equation

$$x^2 + y^2 = 1$$

we use the same algebra technique,

$$\begin{aligned} 5x^2 + 5y^2 + 4xy &= 3 \\ 5(x^2 + y^2 &= 1) \end{aligned}$$

Then we subtract the first equation with the second equation

$$\begin{aligned} 5x^2 + 5y^2 + 4xy - 5x^2 - 5y^2 &= 3 - 5 \\ 4xy &= -2 \\ 4xy + 2 &= 0 \end{aligned}$$

Thus all the $x$ and $y$ values that satisfy $4xy + 2 = 0$ are the point of intersections between the second ellipse and the unit circle.

To compute the degree of rotation we need to compute the eigenvalues then the orthonormal eigenvectors of matrix $A$, after that we construct the standard rotation matrix $P$

$$P = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix} \begin{bmatrix} \cos\ \theta & -\sin\ \theta \\ \sin\ \theta & \cos\ \theta \end{bmatrix}$$

with $v_1$ and $v_2$ are the orthonormal eigenvectors of $A$. Afterwards, we can obtain the degree of rotation, $\theta$. But beware, the direction for the rotation is not always counterclockwise, read more below.

**Direction of Rotation for the Conic**

We know that the default quadratic form of

$$ax^2 + bxy + cy^2 + f = 0$$

if $b \neq 0$ then the conic is rotated, and

1. If $b < 0$ then the conic is rotated counterclockwise
2. If $b > 0$ then the conic is rotated clockwise

the degree can then be computed with the trigonometry formula from the standard rotation matrix $P$ that has the orthonormal eigenvectors of matrix $A$ as the column vectors, and $\det(P) = 1$.

**Figure 23.198:** *The plot of the ellipses $5x^2 + 5y^2 + 4xy = 3$, $5x^2 + 5y^2 + 4xy = 7$, and the unit circle $x^2 + y^2 = 1$ (**DFSimulatorC/Source Codes/Julia/ch23-2dplotimplicitequation2ellipses1circle.jl**).*

The C++ code is using **Imgui** as the widget tool. The directory has structure of

- include
  - stdafx.hpp (headers only)
- src
  - imgui folder
    (this folder can also be obtained after you install Imgui manually, the Imgui version here is **Dear Imgui, v1.80 WIP**, if your installed Imgui in **/usr/include/imgui** has different version, then no need to worry or install new Imgui, since the source code here will be compiled with CMake specifically will look for Imgui from the current working directory **../src/imgui/**)
  - ball.h
  - ellipse.cpp (can be treated like main.cpp, you edit and modify the source code mainly from here)
  - imgui_helper.hpp

For the C++ code, after we are able to simulate / plot in real life the ellipses and circle and change their' parameters, we then need to be able to compute the point of intersection between the ellipses and the unit circle, it will be very useful for future application too, with more codes as usual it can be done with **OpenGL** and **Imgui**.

We need to use the right algorithm and apply it with C++ so we can obtain the point of intersection between each ellipse and the unit circle.

```
#include <stdafx.hpp>
#include "imgui_helper.hpp"
#include "ball.h"
#include <tuple>
#include <thread>
#include <chrono>
#include <vector>
#include <math.h>
#include <list>
#include <numeric>
#include <iostream>
// #include "symbolicc++.h"
#include <armadillo>

using namespace std;
using namespace arma;

double halving(double x)
{
       double y = 2;
       return x/y;
}

double division(double x, double y)
{
       return x/y;
}
#define pi 3.1415926535897
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f

int main(int argc, char const *argv[])
{

       int w = 1280;
       int h = 720;
       std::string title = "Level Curves Simulation";
       initImgui(w, h, title);
```

```
// simulation state
bool show_demo_window = true;
bool show_another_window = false;
ImVec4 clear_color = ImVec4(89.0f / 255.0, 88.0f / 255.0, 87.0f /
    255.0, 1.00f);
ImVec4 ray_color = ImVec4(1.0f, 1.0f, 1.0f, 1.0f);
ImVec4 ray_color2 = ImVec4(0.5f, 0.5f, 0.5f, 1.0f);

// Main loop
int ii = 300;
bool flag = true;
std::list<ImVec2> ellipse_tail;
std::list<ImVec2> ellipse_tail2;
std::list<ImVec2> ellipse_tail3;
std::list<ImVec2> point1;
std::list<ImVec2> point2;
std::list<ImVec2> point3;
std::list<ImVec2> point4;
std::list<int> test;

mat A(2,2, fill::zeros);
cx_mat eigvec;
cx_vec eigval;

mat I(2,2,fill::eye);
// We create symmetric matrix A from quadratic form Q
int N = 2;
float matrixA[N][N] = {};

matrixA[0][0] = 5;
matrixA[0][1] = halving(4);
matrixA[1][0] = matrixA[0][1];
matrixA[1][1] = 5;

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<N; ++j)
       {
              A[i+j*N] = matrixA[i][j];
       }
}

cout <<"Matrix A:" << "\n" << A <<endl;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
    square matrix
arma::vec v1 = real(eigvec.col(0));
arma::vec v2 = real(eigvec.col(1));
```

```
while (!glfwWindowShouldClose(window) && flag == true)
{
        glfwPollEvents();

        ImGui_ImplOpenGL3_NewFrame();
        ImGui_ImplGlfw_NewFrame();

        ImGui::NewFrame();

        ImGuiWindowFlags window_flags = 0;
        ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_FirstUseEver)
            ;
        ImGui::SetNextWindowSize(ImVec2(w, h), ImGuiCond_FirstUseEver
            );
        window_flags |= ImGuiWindowFlags_NoMove;
        window_flags |= ImGuiWindowFlags_NoResize;
        window_flags |= ImGuiWindowFlags_NoCollapse;
        window_flags |= ImGuiWindowFlags_NoBackground;
        window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus;

        ImGui::Begin("Simulation", nullptr, window_flags);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();

        float strokex = 600.0; // the ellipse' and circle center x-
            coordinate
        float strokey = 410.0; // the ellipse' and circle center y-
            coordinate

        static int k1 = 7;
        float k2 = 3;
        static int eigen1 = 7;
        static int eigen2 = 3;
        static int r = 100;

        static int deg = acos(v1[0]) * RADTODEG;
        // for creating the ellipses,
        // we multiply by -deg because rotation will be clockwise
            since in the quadratic form ax^2 + bxy + cy2^2, we have b
            >0
        float theta = DEGTORAD*-deg;

        float p1x = cos(theta); // first column first row of
            transition matrix P
        float p1y = sin(theta); // first column second row of
            transition matrix P
        float p2x = -p1x;
        float p2y = -p1y;
```

```
float p3x = -sin(theta); // second column first row of
    transition matrix P
float p3y = cos(theta); // second column second row of
    transition matrix P
float p4x = -p3x;
float p4y = -p3y;

// The rotated ellipse in parametric equation will be:
// P * [x y]
// P is the transition matrix, the standard rotation matrix
// x = alpha * cos t
// y = beta * sin t
ImVec2 wave_N = {std::cos((float)ii)*r*(sqrt(k1/eigen1)) *
    std::cos((float)theta) - std::sin((float)ii)*r*(sqrt(k1/
    eigen2)) * std::sin((float)theta) + strokex, std::sin((
    float)theta) * std::cos((float)ii)*r*(sqrt(k1/eigen1)) +
    std::cos((float)theta) * std::sin((float)ii)*r*(sqrt(k1/
    eigen2)) + strokey}; // the first ellipse
ImVec2 wave_N2 = {std::cos((float)ii)*r*(sqrt(k2/eigen1)) *
    std::cos((float)theta) - std::sin((float)ii)*r*(sqrt(k2/
    eigen2)) * std::sin((float)theta) + strokex, std::sin((
    float)theta) * std::cos((float)ii)*r*(sqrt(k2/eigen1)) +
    std::cos((float)theta) * std::sin((float)ii)*r*(sqrt(k2/
    eigen2)) + strokey}; // the second ellipse
ImVec2 wave_N3 = {std::cos((float)ii)*r + strokex, std::sin((
    float)ii)*r + strokey}; // The unit circle
ImVec2 point_1 = {std::cos((float)ii)*3+strokex + r*p1x, std
    ::sin((float)ii)*3 + strokey + r*p1y}; // The
    intersection point 1
ImVec2 point_2 = {std::cos((float)ii)*3+strokex + r*p2x, std
    ::sin((float)ii)*3 + strokey + r*p2y}; // The
    intersection point 2
ImVec2 point_3 = {std::cos((float)ii)*3+strokex + r*p3x, std
    ::sin((float)ii)*3 + strokey + r*p3y}; // The
    intersection point 3
ImVec2 point_4 = {std::cos((float)ii)*3+strokex + r*p4x, std
    ::sin((float)ii)*3 + strokey + r*p4y}; // The
    intersection point 4

ellipse_tail.push_front(wave_N);
ellipse_tail2.push_front(wave_N2);
ellipse_tail3.push_front(wave_N3);
point1.push_front(point_1);
point2.push_front(point_2);
point3.push_front(point_3);
point4.push_front(point_4);

// This for loop is the one running the dots that will make
```

```
    the shape of an ellipse
for (auto &it : ellipse_tail)
{
       draw_list->AddCircleFilled(it, 1.0, IM_COL32(45, 140,
           25, 255)); // color green
}
for (auto &it2 : ellipse_tail2)
{
       draw_list->AddCircleFilled(it2, 1.0, IM_COL32(45, 90,
           255, 255)); // color blue
}
for (auto &it3 : ellipse_tail3)
{
       draw_list->AddCircleFilled(it3, 1.0, IM_COL32(55, 10,
           25, 155)); // color black
}
for (auto &it4 : point1)
{
       draw_list->AddCircleFilled(it4, 1.0, IM_COL32(55, 230,
            25, 255)); // color metallic green
}
for (auto &it5 : point2)
{
       draw_list->AddCircleFilled(it5, 1.0, IM_COL32(55, 230,
            25, 255)); // color metallic green
}
for (auto &it6 : point3)
{
       draw_list->AddCircleFilled(it6, 1.0, IM_COL32(55, 230,
            25, 255)); // color metallic green
}
for (auto &it7 : point4)
{
       draw_list->AddCircleFilled(it7, 1.0, IM_COL32(55, 230,
            25, 255)); // color metallic green
}
ii++;
// the if statement to restart the drawing after variable ii
    reach 801
if (ii > 800)
{
       ellipse_tail.erase(ellipse_tail.begin(), ellipse_tail.
           end());
       ellipse_tail2.erase(ellipse_tail2.begin(),
           ellipse_tail2.end());
       ellipse_tail3.erase(ellipse_tail3.begin(),
           ellipse_tail3.end());
       point1.erase(point1.begin(), point1.end());
```

```
        point2.erase(point2.begin(), point2.end());
        point3.erase(point3.begin(), point3.end());
        point4.erase(point4.begin(), point4.end());
        ii = 300;
}
// SymbolicC++ codes start here
/*
Symbolic x("x"), y("y");
auto ellipse1 = 5*(x^2) + 5*(y^2) + 4*x*y == 7;
auto ellipse2 = 5*(x^2) + 5*(y^2) + 4*x*y == 3;
auto circle = (x^2) + (y^2) == 1;

Equation e1 = 5*(x^2) + 5*(y^2) + 4*x*y == 7;
Equation e2 = (x^2) + (y^2) == 1;

Equations eqs = {e1, e2};
list<Symbolic> symbs = {x, y};
list<Equations> sols = solve(eqs, symbs);

*/
// SymbolicC++ codes end here
cout << "point 1 : (x,y) = (" << point_1[0] ;
cout << ", " << point_1[1] << ")" << endl;
cout << "point 2 : (x,y) = (" << point_2[0] ;
cout << ", " << point_2[1] << ")" << endl;
cout << "point 3 : (x,y) = (" << point_3[0] ;
cout << ", " << point_3[1] << ")" << endl;
cout << "point 4 : (x,y) = (" << point_4[0] ;
cout << ", " << point_4[1] << ")" << endl;

ImGui::PushItemWidth(200);
ImGui::SliderInt("r", &r, 51, 146);
ImGui::SliderInt("k1", &k1, -51, 108);
ImGui::SliderFloat("k2", &k2, 0.0f, 5.0f); // Edit 1 float
    using a slider from 0.0f to 5.0f
//sImGui::SliderFloat("k2", &k2, -51, 108,"%.1f", 0); //
    still not working for SliderFloat
ImGui::SliderInt("Eigen 1", &eigen1, -51, 108);
ImGui::SliderInt("Eigen 2", &eigen2, -51, 108);

ImGui::SliderInt("Rotation degree", &deg, 1, 180);
ImGui::Text("Countdown: %d ", 800-ii);
ImGui::Text("For circle with radius of 1 : ");
ImGui::Text("Point of intersection 1: ( %.5f , % .5f )", cos(
    theta), sin(theta));
ImGui::Text("Point of intersection 2: ( %.5f , % .5f )", -
    cos(theta), -sin(theta));
ImGui::Text("Point of intersection 3: ( %.5f , % .5f )", -
```

```
                    sin(theta), cos(theta));
                ImGui::Text("Point of intersection 4: ( %.5f , % .5f )", sin(
                    theta), -cos(theta));
                ImGui::Text("cos(theta): % .5f ", cos(theta));
                ImGui::Text("sin(theta): % .5f ", sin(theta));

                ImGui::PopItemWidth();
                ImGui::End();

                // Rendering
                ImGui::Render();
                int display_w, display_h;
                glfwGetFramebufferSize(window, &display_w, &display_h);
                glViewport(0, 0, display_w, display_h);
                glClearColor(clear_color.x, clear_color.y, clear_color.z,
                    clear_color.w);
                glClear(GL_COLOR_BUFFER_BIT);
                ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

                glfwSwapBuffers(window);
            }

            termImgui();
            return 0;
        }
```

**C++ Code 220:** *src/ellipse.cpp "Finding constrained extrema with level curves"*

To compile it, go to the source code working directory for this section, then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./2D_Ellipses_and_Unit_Circle_Simulation_to_Obtain_Constrained_Extrema**

Explanation for the codes:

- We define the degree of rotation to be 45, and then we declare the float variable for theta by multiplying **DEGTORAD** with negative of the degree ($-45$) because based on the quadratic form
$$z = 5x^2 + 5y^2 + 4xy$$
inspect carefully that $bxy = 4xy$, when $b > 0$ the rotation will be clockwise, that is why in the C++ code we multiply by negative of the degree. The point of intersection can easily be obtained by computing the orthonormal eigenvectors, then the pair is the negative of the orhtonormal eigenvectors. With **Armadillo** we can compute this easily, another method / the harder one is to use nonlinear system solver. But for this section, we just put the first and second column of the standard rotation matrix $P$.

```
            ...
            ...
```

```
static int deg = acos(v1[0]) * RADTODEG;
// for creating the ellipses,
// we multiply by -deg because rotation will be clockwise
    since in the quadratic form ax^2 + bxy + cy2^2, we have b>0
float theta = DEGTORAD*-deg;

float p1x = cos(theta); // first column first row of transition
     matrix P
float p1y = sin(theta); // first column second row of
    transition matrix P
float p2x = -p1x;
float p2y = -p1y;
float p3x = -sin(theta); // second column first row of
    transition matrix P
float p3y = cos(theta); // second column second row of
    transition matrix P
float p4x = -p3x;
float p4y = -p3y;

...
...
```

- When we have the quadratic form

$$z = 5x^2 + 5y^2 + 4xy$$

we can construct the symmetric matrix $A$ as

$$\begin{bmatrix} 5 & 2 \\ 2 & 5 \end{bmatrix}$$

from this, we can compute the orthogonal eigenvectors and the eigenvalues for $A$ as $\lambda_1$ and $\lambda_2$.

We draw the 2 ellipses and circle with parameter of $r$ as the circle radius and the ellipse' base radius too. Now, we start with the first ellipse with equation of

$$5x^2 + 5y^2 + 4xy = 7$$

the ellipse above has the parameter $k_1 = 7$. Now for the second ellipse, the equation is

$$5x^2 + 5y^2 + 4xy = 3$$

the ellipse above has the parameter $k_2 = 3$.

For the first ellipse, we can compute the length for the $x$ and $y$ axis for this ellipse with

$$\alpha_1 = \sqrt{\frac{k_1}{\lambda_1}}$$

$$\beta_1 = \sqrt{\frac{k_1}{\lambda_2}}$$

the parameters above will create an ellipse with standard equation of

$$\frac{x^2}{\alpha_1^2} + \frac{x^2}{beta_1^2} = 1$$

For the second ellipse, we can compute the length for the $x$ and $y$ axis for this ellipse with

$$\alpha_2 = \sqrt{\frac{k_2}{\lambda_1}}$$
$$\beta_2 = \sqrt{\frac{k_2}{\lambda_2}}$$

the parameters above will create an ellipse with standard equation of

$$\frac{x^2}{\alpha_2^2} + \frac{x^2}{beta_2^2} = 1$$

To draw the first ellipse (without rotation) is almost the same with drawing circle in C++, we can use the parametric equation of

$$(x, y) = (\alpha_1 \cos\ t, \beta_1 \sin\ t)$$

To draw the second ellipse (without rotation), we can use the parametric equation of

$$(x, y) = (\alpha_2 \cos\ t, \beta_2 \sin\ t)$$

Now, if we want to rotate the ellipse what we need is to multiply a standard rotation matrix $P$ with each of the ellipse, thus for the first ellipse that is being rotated clockwise by $\theta$

$$P \times \text{ellipse}_1 = \begin{bmatrix} \cos\ (-\theta) & -\sin\ (-\theta) \\ \sin\ (-\theta) & \cos\ (-\theta) \end{bmatrix} \begin{bmatrix} \alpha_1 \cos\ t \\ \beta_1 \sin\ t \end{bmatrix}$$

for the second ellipse that is being rotated clockwise by $\theta$

$$P \times \text{ellipse}_2 = \begin{bmatrix} \cos\ (-\theta) & -\sin\ (-\theta) \\ \sin\ (-\theta) & \cos\ (-\theta) \end{bmatrix} \begin{bmatrix} \alpha_2 \cos\ t \\ \beta_2 \sin\ t \end{bmatrix}$$

```
...
...

ImVec2 wave_N = {std::cos((float)ii)*r*(sqrt(k1/eigen1)) * std
    ::cos((float)theta) - std::sin((float)ii)*r*(sqrt(k1/
    eigen2)) * std::sin((float)theta) + strokex, std::sin((
    float)theta) * std::cos((float)ii)*r*(sqrt(k1/eigen1)) +
    std::cos((float)theta) * std::sin((float)ii)*r*(sqrt(k1/
    eigen2)) + strokey}; // the first ellipse
ImVec2 wave_N2 = {std::cos((float)ii)*r*(sqrt(k2/eigen1)) * std
    ::cos((float)theta) - std::sin((float)ii)*r*(sqrt(k2/
    eigen2)) * std::sin((float)theta) + strokex, std::sin((
    float)theta) * std::cos((float)ii)*r*(sqrt(k2/eigen1)) +
    std::cos((float)theta) * std::sin((float)ii)*r*(sqrt(k2/
    eigen2)) + strokey}; // the second ellipse
```

```
            ImVec2 wave_N3 = {std::cos((float)ii)*r + strokex, std::sin((
                float)ii)*r + strokey}; // The unit circle
            ImVec2 point_1 = {std::cos((float)ii)*3+strokex + r*p1x, std::
                sin((float)ii)*3 + strokey + r*p1y}; // The intersection
                point 1
            ImVec2 point_2 = {std::cos((float)ii)*3+strokex + r*p2x, std::
                sin((float)ii)*3 + strokey + r*p2y}; // The intersection
                point 2
            ImVec2 point_3 = {std::cos((float)ii)*3+strokex + r*p3x, std::
                sin((float)ii)*3 + strokey + r*p3y}; // The intersection
                point 3
            ImVec2 point_4 = {std::cos((float)ii)*3+strokex + r*p4x, std::
                sin((float)ii)*3 + strokey + r*p4y}; // The intersection
                point 4

            ...
            ...
```

- Another warning, you might notice it too, when using Imgui for this project, the $y$ axis has reverted value, so from top to bottom the $y$ axis is increasing in value, instead of the usual Euclidean coordinate when the $y$ axis has increasing value from bottom to top. That is why when we are able to compute the point of intersection it is mirrored, reverted by the $y$ axis that has reverted value.

**Figure 23.199:** *The computation and simulation to obtain the constrained extremum (maximum and minimum values) of the quadratic form $z = 5x^2 + 5y^2 + 4xy$ subject to the constraint $x^2 + y^2 = 1$ (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Ellipses and Unit Circle Simulation to Obtain Constrained Extrema/**).*

## xi. C++ Computation: Classify Relative Extrema with Hessian

Find the critical points of the function

$$f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$$

and use the eigenvalues of the Hessian matrix at those points to determine which of them, if any, are relative maxima, relative minima, or saddle points.

**Solution:**

To find both the critical points and the Hessian matrix we will need to calculate the first and second partial derivatives of $f$. These derivatives are

$$\begin{aligned} f_x(x,y) &= x^2 + y^2 - 8y \\ f_y(x,y) &= 2xy - 8x \\ f_{xy}(x,y) &= 2y - 8 \\ f_{xx}(x,y) &= 2x \\ f_{yy}(x,y) &= 2x \end{aligned}$$

Thus, the Hessian matrix is

$$H(x,y) = \begin{bmatrix} f_{xx}(x,y) & f_{xy}(x,y) \\ f_{xy}(x,y) & f_{yy}(x,y) \end{bmatrix} = \begin{bmatrix} 2x & 2y-8 \\ 2y-8 & 2x \end{bmatrix}$$

To find the critical points we set $f_x$ and $f_y$ equal to zero. This yields the equations

$$\begin{aligned} f_x(x,y) &= x^2 + y^2 - 8y = 0 \\ f_y(x,y) &= 2xy - 8x = 0 \end{aligned}$$

Solving the second equation yields $x = 0$ or $y = 4$. Substituting $x = 0$ in the first equation and solving for $y$ yields $y = 0$ or $y = 8$; and substituting $y = 4$ into the first equation and solving for $x$ yields $x = 4$ or $x = -4$. Thus, we have four critical points:

$$(0,0), \quad (0,8), \quad (4,4), \quad (-4,4)$$

Evaluating the Hessian matrix at these points yields

$$H(0,0) = \begin{bmatrix} 0 & -8 \\ -8 & 0 \end{bmatrix}, \quad H(0,8) = \begin{bmatrix} 0 & 8 \\ 8 & 0 \end{bmatrix}$$

$$H(4,4) = \begin{bmatrix} 8 & 0 \\ 0 & 8 \end{bmatrix}, \quad H(-4,4) = \begin{bmatrix} -8 & 0 \\ 0 & -8 \end{bmatrix}$$

For the C++ codes, we will use 2 different C++ codes:

1. **Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL**
   This code is used to determine the four critical points with **GSL** library (GNU Scientific Library).

   The original code is obtained from:
   **GNU Scientific Library, release 2.7 Book, chapter 38**
   (https://www.gnu.org/software/gsl/doc/html/multiroots.html)

| Critical Point $(x_0, y_0)$ | $\lambda_1$ | $\lambda_2$ | Classification |
|---|---|---|---|
| $(0,0)$ | $8$ | $-8$ | Saddle point |
| $(0,8)$ | $8$ | $-8$ | Saddle point |
| $(4,4)$ | $8$ | $8$ | Relative minimum |
| $(0,0)$ | $-8$ | $-8$ | Relative maximum |

**Table 23.4:** *Classifications of the stationary points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$

2. **Classify Relative Extrema using Hessian Matrix with SymbolicC++ and Armadillo**
   This code is used to classify the critical points / the stationary points with **Armadillo** combined with **SymbolicC++**.

   The eigenvalues for each Hessian matrices above can be computed easily with **Armadillo** library.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams
{
        double a;
        double b;
};

// f1(x,y) = x^2 + y^2 - 8y
// f2(x,y) = 2xy - 8x
// the solution of this system is (x,y)=(0,0) , (0,8), (4,4) , (-4,4)
int rosenbrock_f (const gsl_vector * x, void *params, gsl_vector * f)
{

        const double x0 = gsl_vector_get (x, 0);
        const double x1 = gsl_vector_get (x, 1);

        const double y0 = pow(x0,2) + pow(x1,2) - 8*x1;
        const double y1 = 2*x0*x1 - 8*x0;

        gsl_vector_set (f, 0, y0);
        gsl_vector_set (f, 1, y1);

        return GSL_SUCCESS;
}

int print_state (size_t iter, gsl_multiroot_fsolver * s)
{
```

```
        printf ("n = %3u x = % .3f y = % .3f " "\t f1(x,y) = % .3e f2(x,y) =
             % .3e\n",
        iter,
        gsl_vector_get (s->x, 0),
        gsl_vector_get (s->x, 1),
        gsl_vector_get (s->f, 0),
        gsl_vector_get (s->f, 1));
        return 0;
}

int main (void)
{
        const gsl_multiroot_fsolver_type *T;
        gsl_multiroot_fsolver *s;

        int status;
        size_t i, iter = 0;

        const size_t n = 2;
        struct rparams p = {1.0, 10.0};
        gsl_multiroot_function f = {&rosenbrock_f, n, &p};

        //double x_init[2] = {-10.0, 0.0};
        //double x_init[2] = {-3.0, 1.0};
        //double x_init[2] = {0.0, 5.0};
        double x_init[2] = {5.0, 0.0};
        gsl_vector *x = gsl_vector_alloc (n);

        gsl_vector_set (x, 0, x_init[0]);
        gsl_vector_set (x, 1, x_init[1]);

        T = gsl_multiroot_fsolver_hybrids;
        s = gsl_multiroot_fsolver_alloc (T, 2);
        gsl_multiroot_fsolver_set (s, &f, x);

        print_state (iter, s);

        do
        {
                iter++;
                status = gsl_multiroot_fsolver_iterate (s);

                print_state (iter, s);

                if (status) /* check if solver is stuck */
                break;

                status = gsl_multiroot_test_residual (s->f, 1e-7);
```

```
        }
        while (status == GSL_CONTINUE && iter < 1000);

        printf ("status = %s\n", gsl_strerror (status));

        gsl_multiroot_fsolver_free (s);
        gsl_vector_free (x);

        return 0;
    }
```

**C++ Code 221:** *main.cpp "Determine critical points"*

To compile it, type:
**g++ -o main main.cpp -lgsl -lcblas**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The function to input the functions we want to solve for the nonlinear system

$$f_x(x,y) = x^2 + y^2 - 8y = 0$$
$$f_y(x,y) = 2xy - 8x = 0$$

```
int rosenbrock_f (const gsl_vector * x, void *params,
    gsl_vector * f)
{

    const double x0 = gsl_vector_get (x, 0);
    const double x1 = gsl_vector_get (x, 1);

    const double y0 = pow(x0,2) + pow(x1,2) - 8*x1;
    const double y1 = 2*x0*x1 - 8*x0;

    gsl_vector_set (f, 0, y0);
    gsl_vector_set (f, 1, y1);

    return GSL_SUCCESS;
}
```

- The function to print the result.

```
int print_state (size_t iter, gsl_multiroot_fsolver * s)
{
    printf ("n = %3u x = % .3f y = % .3f " "\t f1(x,y) = %
        .3e f2(x,y) = % .3e\n",
```

```
                iter,
                gsl_vector_get (s->x, 0),
                gsl_vector_get (s->x, 1),
                gsl_vector_get (s->f, 0),
                gsl_vector_get (s->f, 1));
                return 0;
        }
```

- This is the part of the code inside the **int main(void)** to change the initial guess for $x$, the left interval is the starting point of the initial guess, whether you put positive number or negative number it will goes toward 0.

```
        ...
        ...

        //double x_init[2] = {-10.0, 0.0};
        //double x_init[2] = {-3.0, 1.0};
        //double x_init[2] = {0.0, 5.0};
        double x_init[2] = {5.0, 0.0};
        gsl_vector *x = gsl_vector_alloc (n);

        ...
        ...
```

```
n =   0 x = -10.000 y = -1.000   f1(x,y) =  1.090e+02 f2(x,y) =  1.000e+02
n =   1 x = -6.067 y =  2.033    f1(x,y) =  2.467e+01 f2(x,y) =  2.386e+01
n =   2 x = -4.964 y =  3.031    f1(x,y) =  9.577e+00 f2(x,y) =  9.618e+00
n =   3 x = -4.275 y =  3.711    f1(x,y) =  2.358e+00 f2(x,y) =  2.470e+00
n =   4 x = -4.051 y =  3.946    f1(x,y) =  4.143e-01 f2(x,y) =  4.415e-01
n =   5 x = -4.003 y =  3.996    f1(x,y) =  2.716e-02 f2(x,y) =  2.892e-02
n =   6 x = -4.000 y =  4.000    f1(x,y) =  3.659e-04 f2(x,y) =  3.598e-04
n =   7 x = -4.000 y =  4.000    f1(x,y) =  1.329e-06 f2(x,y) = -4.519e-06
n =   8 x = -4.000 y =  4.000    f1(x,y) =  2.007e-07 f2(x,y) = -9.619e-07
n =   9 x = -4.000 y =  4.000    f1(x,y) = -2.811e-09 f2(x,y) =  1.347e-08
status = success
```

**Figure 23.200:** *The computation to determine the critical points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$ *with initial guess for* $x$ *starting at* $-10$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL)****.*

```
n =   0 x = -3.000 y =  1.000    f1(x,y) =  2.000e+00 f2(x,y) =  1.800e+01
n =   1 x = -3.000 y =  1.000    f1(x,y) =  2.000e+00 f2(x,y) =  1.800e+01
n =   2 x = -2.152 y =  1.819    f1(x,y) = -6.611e+00 f2(x,y) =  9.388e+00
n =   3 x = -1.997 y =  1.933    f1(x,y) = -7.738e+00 f2(x,y) =  8.257e+00
n =   4 x = -1.943 y =  1.941    f1(x,y) = -7.989e+00 f2(x,y) =  7.998e+00
n =   5 x = -1.517 y =  1.533    f1(x,y) = -7.611e+00 f2(x,y) =  7.487e+00
n =   6 x = -0.723 y =  0.662    f1(x,y) = -4.335e+00 f2(x,y) =  4.828e+00
n =   7 x = -0.723 y =  0.662    f1(x,y) = -4.335e+00 f2(x,y) =  4.828e+00
n =   8 x = -0.106 y =  0.106    f1(x,y) = -8.241e-01 f2(x,y) =  8.221e-01
n =   9 x =  0.023 y = -0.023    f1(x,y) =  1.837e-01 f2(x,y) = -1.830e-01
n =  10 x = -0.001 y =  0.001    f1(x,y) = -4.917e-03 f2(x,y) =  4.917e-03
n =  11 x = -0.000 y =  0.000    f1(x,y) = -2.777e-05 f2(x,y) =  2.777e-05
n =  12 x =  0.000 y = -0.000    f1(x,y) =  4.296e-09 f2(x,y) = -4.294e-09
status = success
```

**Figure 23.201:** *The computation to determine the critical points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$ *with initial guess for* $x$ *starting at* $-3$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL/).***

```
n =   0 x =  0.000 y =  5.000    f1(x,y) = -1.500e+01 f2(x,y) =  0.000e+00
n =   1 x =  0.000 y =  5.000    f1(x,y) = -1.500e+01 f2(x,y) =  0.000e+00
n =   2 x =  0.000 y =  6.579    f1(x,y) = -9.349e+00 f2(x,y) =  0.000e+00
n =   3 x =  0.000 y =  6.579    f1(x,y) = -9.349e+00 f2(x,y) =  0.000e+00
n =   4 x =  0.000 y =  7.782    f1(x,y) = -1.695e+00 f2(x,y) =  0.000e+00
n =   5 x =  0.000 y =  8.049    f1(x,y) =  3.917e-01 f2(x,y) =  0.000e+00
n =   6 x =  0.000 y =  7.999    f1(x,y) = -1.083e-02 f2(x,y) =  0.000e+00
n =   7 x =  0.000 y =  8.000    f1(x,y) = -6.550e-05 f2(x,y) =  0.000e+00
n =   8 x =  0.000 y =  8.000    f1(x,y) =  1.109e-08 f2(x,y) =  0.000e+00
status = success
```

**Figure 23.202:** *The computation to determine the critical points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$ *with initial guess for* $x$ *starting at* $0$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL/).***

```
n =   0 x =  5.000 y =  0.000    f1(x,y) =  2.500e+01 f2(x,y) = -4.000e+01
n =   1 x =  6.944 y =  5.556    f1(x,y) =  3.465e+01 f2(x,y) =  2.160e+01
n =   2 x =  3.344 y =  2.136    f1(x,y) = -1.343e+00 f2(x,y) = -1.247e+01
n =   3 x =  3.624 y =  3.048    f1(x,y) = -1.958e+00 f2(x,y) = -6.899e+00
n =   4 x =  4.315 y =  4.473    f1(x,y) =  2.842e+00 f2(x,y) =  4.078e+00
n =   5 x =  3.961 y =  3.888    f1(x,y) = -2.991e-01 f2(x,y) = -8.834e-01
n =   6 x =  4.001 y =  3.982    f1(x,y) =  1.124e-02 f2(x,y) = -1.419e-01
n =   7 x =  4.003 y =  3.997    f1(x,y) =  2.337e-02 f2(x,y) = -2.607e-02
n =   8 x =  4.001 y =  3.999    f1(x,y) =  9.302e-03 f2(x,y) = -1.175e-02
n =   9 x =  4.000 y =  4.000    f1(x,y) = -6.408e-04 f2(x,y) =  8.182e-04
n =  10 x =  4.000 y =  4.000    f1(x,y) =  1.633e-06 f2(x,y) = -2.152e-06
n =  11 x =  4.000 y =  4.000    f1(x,y) = -1.553e-08 f2(x,y) =  1.976e-08
status = success
```

**Figure 23.203:** *The computation to determine the critical points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$ *with initial guess for* $x$ *starting at* $5$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL/).***

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
#include <armadillo>

using namespace std;
using namespace arma;

#define R 2 // number of rows
#define C 2 // number of columns

double division(double x, double y)
{
       return x/y;
}

// Driver program
int main()
{
       Symbolic x("x"), y("y");
       Symbolic f, fx, fy, fxx, fxy, fyy;

       f = division(1,3)*(x*x*x) + x*(y*y) — 8*x*y + 3;
       fx = df(f, x);
       fy = df(f, y);
       fxx = df(fx, x);
       fxy = df(fx, y);
       fyy = df(fy, y);

       cout << "f : " << f << endl;
       cout << "fx : " << fx << endl;
       cout << "fy : " << fy << endl;
       cout << "fxx : " << fxx << endl;
       cout << "fxy : " << fxy << endl;
       cout << "fyy : " << fyy << endl;

       // Construct a symbolic matrix of size 2 X 2
       Matrix<Symbolic> H_mat(2,2);
       H_mat[0][0] = fxx; H_mat[0][1] = fxy;
       H_mat[1][0] = fxy; H_mat[1][1] = fyy;
       cout << "\nH(x,y):\n" << H_mat <<endl;
       cout << "det(H):\n" << H_mat.determinant() <<endl;
       //cout << "inv(B):\n" << B_mat.inverse() <<endl;
       cout << endl;

       // To find critical point set fx and fy equal to zero and solve the
```

```
    equations
cout << "Computing the critical points: set fx =0 and fy=0, then
    solve the equations.\n" << endl;
cout << "Determine the critical codes with C++ codes with GSL.\n" <<
     endl;
/*
Equation e1 = fx == 0;
Equation e2 = fy == 0;

Equations eqs = {e1, e2};
list<Symbolic> symbs = {x, y};
list<Equations> sols = solve(eqs, symbs);

Symbolic x_sol, y_sol;
int i = 1;
for( auto iter1 = sols.begin(); iter1 != sols.end(); iter1++)
{
       x_sol = x.subst((*(*iter1).begin()));
       y_sol = y.subst((*(——(*iter1).end())));
       cout << "p" << i << " = {" << x_sol << ", " << y_sol << "};"
           << endl;
       cout << "fx (x=" << x_sol << "):" << fx[x==x_sol] << endl;
       cout << "fy (x=" << x_sol << "):" << fy[x==x_sol] << endl;
       cout << "fx (y=" << y_sol << "):" << fx[y==y_sol] << endl;
       cout << "fy (y=" << y_sol << "):" << fy[y==y_sol] << endl;
       cout << endl;
       i++;
}
*/
cout << endl;

// Compute the critical points with GSL first in the first C++ codes
     then you can input the critical points here
mat H1(R,C);
mat H2(R,C);
mat H3(R,C);
mat H4(R,C);

H1[0] = fxx[x==0, y==0];
H1[1] = fxy[x==0, y==0];
H1[2] = fxy[x==0, y==0];
H1[3] = fyy[x==0, y==0];
H2[0] = fxx[x==0, y==8];
H2[1] = fxy[x==0, y==8];
H2[2] = fxy[x==0, y==8];
H2[3] = fyy[x==0, y==8];
H3[0] = fxx[x==4, y==4];
H3[1] = fxy[x==4, y==4];
```

```
H3[2] = fxy[x==4, y==4];
H3[3] = fyy[x==4, y==4];
H4[0] = fxx[x==-4, y==4];
H4[1] = fxy[x==-4, y==4];
H4[2] = fxy[x==-4, y==4];
H4[3] = fyy[x==-4, y==4];

cx_mat eigvec;
cx_vec eigval;

mat I(R,C,fill::eye);
cout <<"Matrix H with critical point (0,0):" << "\n" << H1 <<endl;
cout <<"Matrix H with critical point (0,8):" << "\n" << H2 <<endl;
cout <<"Matrix H with critical point (4,4):" << "\n" << H3 <<endl;
cout <<"Matrix H with critical point (-4,4):" << "\n" << H4 <<endl;

eig_gen(eigval, eigvec, H1); // Eigen decomposition of dense general
     square matrix
cout << "Eigenvalues for matrix H with critical point (0,0):" <<
    endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}
if (real(eigval[0]) < 0 && real(eigval[1]) < 0 )
{
       cout << "This is a relative maximum" << endl;
}
else if (real(eigval[0]) > 0 && real(eigval[1]) > 0 )
{
       cout << "This is a relative minimum" << endl;
}
else if (real(eigval[0]) < 0 && real(eigval[1]) > 0 )
{
       cout << "This is a saddle point" << endl;
}
else if (real(eigval[0]) > 0 && real(eigval[1]) < 0 )
{
       cout << "This is a saddle point" << endl;
}

eig_gen(eigval, eigvec, H2); // Eigen decomposition of dense general
     square matrix
cout << endl;
cout << "\nEigenvalues for matrix H with critical point (0,8):" <<
```

```
                endl;

            for (int i = 1; i <= n; i++)
            {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
            }
            if (real(eigval[0]) < 0 && real(eigval[1]) < 0 )
            {
                cout << "This is a relative maximum" << endl;
            }
            else if (real(eigval[0]) > 0 && real(eigval[1]) > 0 )
            {
                cout << "This is a relative minimum" << endl;
            }
            else if (real(eigval[0]) < 0 && real(eigval[1]) > 0 )
            {
                cout << "This is a saddle point" << endl;
            }
            else if (real(eigval[0]) > 0 && real(eigval[1]) < 0 )
            {
                cout << "This is a saddle point" << endl;
            }

            eig_gen(eigval, eigvec, H3); // Eigen decomposition of dense general
                 square matrix
            cout << endl;
            cout << "\nEigenvalues for matrix H with critical point (4,4):" <<
                endl;

            for (int i = 1; i <= n; i++)
            {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
            }
            if (real(eigval[0]) < 0 && real(eigval[1]) < 0 )
            {
                cout << "This is a relative maximum" << endl;
            }
            else if (real(eigval[0]) > 0 && real(eigval[1]) > 0 )
            {
                cout << "This is a relative minimum" << endl;
            }
            else if (real(eigval[0]) < 0 && real(eigval[1]) > 0 )
            {
                cout << "This is a saddle point" << endl;
            }
            else if (real(eigval[0]) > 0 && real(eigval[1]) < 0 )
```

```
        {
            cout << "This is a saddle point" << endl;
        }

        eig_gen(eigval, eigvec, H4); // Eigen decomposition of dense general
            square matrix
        cout << endl;
        cout << "\nEigenvalues for matrix H with critical point (-4,4):" <<
            endl;

        for (int i = 1; i <= n; i++)
        {
            cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                imag(eigval[i-1]) << "i" << endl;
        }
        if (real(eigval[0]) < 0 && real(eigval[1]) < 0 )
        {
            cout << "This is a relative maximum" << endl;
        }
        else if (real(eigval[0]) > 0 && real(eigval[1]) > 0 )
        {
            cout << "This is a relative minimum" << endl;
        }
        else if (real(eigval[0]) < 0 && real(eigval[1]) > 0 )
        {
            cout << "This is a saddle point" << endl;
        }
        else if (real(eigval[0]) > 0 && real(eigval[1]) < 0 )
        {
            cout << "This is a saddle point" << endl;
        }

        /*
        cout << "\nfx (x=0,y=0):\n" << fx[x==0, y==0] <<endl;
        cout << "fx (x=0,y=8):\n" << fx[x==0, y==8] <<endl;
        cout << "fx (x=4,y=4):\n" << fx[x==4, y==4] <<endl;
        cout << "fx (x=-4,y=4):\n" << fx[x==-4, y==4] <<endl;
        cout << "fy (x=0, y=0):\n" << fy[x==0, y==0] <<endl;
        cout << "fy (x=0, y=8):\n" << fy[x==0, y==8] <<endl;
        cout << "fy (x=4, y=4):\n" << fy[x==4, y==4] <<endl;
        cout << "fy (x=-4, y=4):\n" << fy[x==-4, y==4] <<endl;
        */
        return 0;
    }
```

**C++ Code 222:** *main.cpp "Classify relative extrema with Hessian"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++ -larmadillo**

**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The commented section here is to compute the solution for the nonlinear system $f_x(x, y) = 0$ and $f_y(x, y) = 0$, **SymbolicC++** only able to compute one critical point which is $(0, 0)$, so the first C++ code that is using **GSL** is better in a part where we have to compute all critical points, with different initial guess / starting points we will obtain all critical points.

```
/*
Equation e1 = fx == 0;
Equation e2 = fy == 0;

Equations eqs = {e1, e2};
list<Symbolic> symbs = {x, y};
list<Equations> sols = solve(eqs, symbs);

Symbolic x_sol, y_sol;
int i = 1;
for( auto iter1 = sols.begin(); iter1 != sols.end(); iter1++)
{
       x_sol = x.subst((*(*iter1).begin()));
       y_sol = y.subst((*(--(*iter1).end())));
       cout << "p" << i << " = {" << x_sol << ", " << y_sol <<
           "};" << endl;
       cout << "fx (x=" << x_sol << "):" << fx[x==x_sol] <<
           endl;
       cout << "fy (x=" << x_sol << "):" << fy[x==x_sol] <<
           endl;
       cout << "fx (y=" << y_sol << "):" << fx[y==y_sol] <<
           endl;
       cout << "fy (y=" << y_sol << "):" << fy[y==y_sol] <<
           endl;
       cout << endl;
       i++;
}
*/
```

- The rest of the codes are simple enough to be understood, where we already get used to compute eigenvalues with **Armadillo** and use **SymbolicC++** to make symbolic matrix and then substitute the variables with float numbers.

```
f : 0.333333*x^(3)+x*y^(2)-8*x*y+3
fx : x^(2)+y^(2)-8*y
fy : 2*x*y-8*x
fxx : 2*x
fxy : 2*y-8
fyy : 2*x

H(x,y):
[ 2*x  2*y-8]
[2*y-8  2*x ]

det(H):
4*x^(2)-4*y^(2)+32*y-64

Computing the critical points: set fx =0 and fy=0, then solve the equations.

Determine the critical codes with C++ codes with GSL.


Matrix H with critical point (0,0):
        0  -8.0000
  -8.0000        0

Matrix H with critical point (0,8):
        0   8.0000
   8.0000        0

Matrix H with critical point (4,4):
   8.0000        0
        0   8.0000

Matrix H with critical point (-4,4):
  -8.0000        0
        0  -8.0000

Eigenvalues for matrix H with critical point (0,0):
λ_1 = 8 + 0i
λ_2 = -8 + 0i
This is a saddle point

Eigenvalues for matrix H with critical point (0,8):
λ_1 = 8 + 0i
λ_2 = -8 + 0i
This is a saddle point

Eigenvalues for matrix H with critical point (4,4):
λ_1 = 8 + 0i
λ_2 = 8 + 0i
This is a relative minimum

Eigenvalues for matrix H with critical point (-4,4):
λ_1 = -8 + 0i
λ_2 = -8 + 0i
This is a relative maximum
```

**Figure 23.204:** *The computation to classify the critical points of the function* $f(x,y) = \frac{1}{3}x^3 + xy^2 - 8xy + 3$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Classify Relative Extrema using Hessian Matrix with SymbolicC++ and Armadillo/).***

## xii. C++ Computation: A Constrained Extremum Problem, Temperature on a Metal Plate

Suppose that the temperature at a point $(x, y)$ on a metal plate is $T(x, y) = 4x^2 - 4xy + y^2$. An ant, walking on the plate, traverses a circle of radius 5 centered at the origin. What are the highest and lowest temperatures encountered by the ant?

**Solution:**

The problem is to maximize and minimize the quadratic form $T(x, y) = 4x^2 - 4xy + y^2$ subject to the constraint $x^2 + y^2 = 25$.

We can rewrite the constraint as

$$\left(\frac{x}{5}\right)^2 + \left(\frac{y}{5}\right)^2 = 1$$

and defining new variables, $x_1$ and $y_1$, by the equations

$$x = 5x_1$$
$$y = 5y_1$$

This enables us to reformulate the problem as follows:

$$\text{maximize } z = 4x^2 - 4xy + y^2 = 100x_1^2 - 100x_1y_1 + 25y_1^2$$
$$\text{minimize } z = 4x^2 - 4xy + y^2 = 100x_1^2 - 100x_1y_1 + 25y_1^2$$

subject to the constraint

$$x_1^2 + y_1^2 = 1$$

To solve this problem, we will write the quadratic form $z = 100x_1^2 - 100x_1y_1 + 25y_1^2$ as

$$z = \boldsymbol{x}^T A \boldsymbol{x} = \begin{bmatrix} x_1 & y_1 \end{bmatrix} \begin{bmatrix} 100 & -50 \\ -50 & 25 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

The eigenvalues of $A$ are

$$\lambda_1 = 125$$
$$\lambda_2 = 0$$

The corresponding unit eigenvector for $\lambda_1 = 125$ is

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0.8944 \\ -0.4472 \end{bmatrix}$$

The corresponding unit eigenvector for $\lambda_1 = 0$ is

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0.4472 \\ 0.8944 \end{bmatrix}$$

Thus, the maximum temperature encountered by the ant is $T(x, y) = 125$, and this occurs when

$$x = 5x_1 = 4.47199$$
$$y = 5y_1 = -2.23599$$

The minimum temperature encountered by the ant is $T(x, y) = 0$, and this occurs when

$$x = 5x_1 = 2.23599$$
$$y = 5y_1 = 4.47199$$

You can check it by substituting $x, y$ for each of the maximum and minimum temperature into the equation $T(x, y)$ or the $x_1, y_1$ into $T(x_1, y_1)$.

After we determine matrix $A$, we can compute the eigenvalues and orthonormal eigenvectors easily with **Armadillo**

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix

       cout << "Eigenvalues:" << endl;

       int n = 2;
       for (int i = 1; i <= n; i++)
       {
              cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                  imag(eigval[i-1]) << "i" << endl;
       }

       cout << endl;
       cout <<"Matrix l_1I - A:" << "\n" << I*eigval[0] - A <<endl;
       cout <<"Matrix l_2I - A:" << "\n" << I*eigval[1] - A <<endl;
       mat A1 = I*real(eigval[0]) - A;
       mat A2 = I*real(eigval[1]) - A;

       cout << endl;
```

```
        cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
        cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
        cout << endl;

        cout << "Orthonormal Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
            cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                endl;
        }

        return 0;
    }
```

**C++ Code 223:** *main.cpp "Compute eigenvalues and eigenvectors for 2 dimensional matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A:
   1.0000e+02  -5.0000e+01
  -5.0000e+01   2.5000e+01

Eigenvalues:
λ_1 = 125 + 0i
λ_2 = 0 + 0i

Matrix λ_1I - A:
    (+2.500e+01,+0.000e+00)    (+5.000e+01,+0.000e+00)
    (+5.000e+01,+0.000e+00)    (+1.000e+02,+0.000e+00)

Matrix λ_2I - A:
    (-1.000e+02,+0.000e+00)    (+5.000e+01,+0.000e+00)
    (+5.000e+01,+0.000e+00)    (-2.500e+01,+0.000e+00)


Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Orthonormal Eigenvectors:
x[λ_1] :
    0.8944
  -0.4472

x[λ_2] :
    0.4472
   0.8944
```

**Figure 23.205:** *The computation to obtain the eigenvalues and orthonormal eigenvector of 2 dimensional matrix $A$ from quadratic form $T(x,y) = 4x^2 - 4xy + y^2$ subject to the constraint $x^2 + y^2 = 25$ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Eigenvectors for 2 Dimensional Matrix with Armadillo/**).*

## xiii. C++ Computation: Second Derivative Test for Two Functions

(a) Show that the functions

$$f(x,y) = x^4 + y^4$$
$$g(x,y) = x^4 - y^4$$

have a critical point at $(0,0)$ but the second derivative test is inconclusive at that point.

(b) Give a reasonable argument to show that $f$ has a relative minimum at $(0,0)$ and $g$ has a saddle point at $(0,0)$.

**Solution:**

(a) To find the critical points we set $f_x$ and $f_y$ equal to zero, then to the second function we set $g_x$ and $g_y$ equal to zero.

$$f_x(x,y) = 4x^3 = 0$$
$$f_y(x,y) = 4y^3 = 0$$

Solving the equations above yield the critical point $(0,0)$. Now, to the second function

$$g_x(x,y) = 4x^3 = 0$$
$$g_y(x,y) = -4y^3 = 0$$

Solving the equations above yield the critical point $(0,0)$.

For the second derivative test on the first function we will do the computation below

$$f_{xx}(x,y)f_{yy}(x,y) - f_{xy}^2(x,y) = 12x^2 12y^2 - 0 = 144x^2y^2$$

For the second derivative test on the second function we will do the computation below

$$g_{xx}(x,y)g_{yy}(x,y) - g_{xy}^2(x,y) = 12x^2(-12y^2) - 0 = 144x^2y^2$$

If we input $(0,0)$ on both equations above, we will obtain the second derivative test is inconclusive since $f_{xx}(x_0,y_0)f_{yy}(x_0,y_0) - f_{xy}^2(x_0,y_0) = 0$ and $g_{xx}(x_0,y_0)g_{yy}(x_0,y_0) - g_{xy}^2(x_0,y_0) = 0$.

(b) If we take a look at the second derivative test for $f$ we will see that

$$f_{xx}(x,y)f_{yy}(x,y) - f_{xy}^2(x,y) = 12x^2 12y^2 - 0 = 144x^2y^2$$

when we are close to the critical point $(0,0)$ either from minus $x$ / positive $x$ or from minus $y$ / positive $y$ we will have the value of

$$f_{xx}(x_0,y_0)f_{yy}(x_0,y_0) - f_{xy}^2(x_0,y_0) = 144x_0^2y_0^2$$
$$144x_0^2y_0^2 > 0$$

to be a positive value, since $x_0^2$ and $y_0^2$ will always be positive, for all real numbers when it is near $(0,0)$. Thus we know that when $f_{xx}(x_0,y_0)f_{yy}(x_0,y_0) - f_{xy}^2(x_0,y_0) > 0$, and then we analyze more

$$f_{xx}(x_0,y_0) = 12x^2$$
$$12x^2 > 0$$

we can tell that $12x^2$ will always be positive for all values of $x$ near $(0,0)$, hence the point is a relative minimum for that critical point.

For the second function, we will analyze with same method

$$g_{xx}(x_0,y_0)g_{yy}(x_0,y_0) - g_{xy}^2(x_0,y_0) = -144x_0^2y_0^2$$
$$-144x_0^2y_0^2 < 0$$

the value of $-144x_0^2y_0^2$ will always be negative, for all real numbers when it is near $(0,0)$. Thus we know that when $f_{xx}(x_0,y_0)f_{yy}(x_0,y_0) - f_{xy}^2(x_0,y_0) < 0$ then the point is a saddle point for that critical point.

Given the case like in this example, we have to analyze the surrounding points around that critical point, so we can make sure whether it is a relative minimum, relative maximum, or a saddle point, especially since the second derivative test is inconclusive.

The first code is using **GSL**, almost the same as the previous 2 sections, we only change the system of nonlinear equations, with this code we can compute the critical points to be $(0,0)$.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams
{
       double a;
       double b;
};

// f1(x,y) = x^4 + y^4
// f2(x,y) = x^4 — y^4
// the solution of this system is (x,y)=(0,0)
int rosenbrock_f (const gsl_vector * x, void *params, gsl_vector * f)
{
       const double x0 = gsl_vector_get (x, 0);
       const double x1 = gsl_vector_get (x, 1);

       //const double y0 = 4*pow(x0,3) ;
       //const double y1 = 4*pow(x1,3);
```

```
        const double y0 = 4*pow(x0,3) ;
        const double y1 = -4*pow(x1,3);

        gsl_vector_set (f, 0, y0);
        gsl_vector_set (f, 1, y1);

        return GSL_SUCCESS;
}

int print_state (size_t iter, gsl_multiroot_fsolver * s)
{
        printf ("n = %3u x = % .3f y = % .3f " "\t f1(x,y) = % .3e f2(x,y) =
             % .3e\n",
        iter,
        gsl_vector_get (s->x, 0),
        gsl_vector_get (s->x, 1),
        gsl_vector_get (s->f, 0),
        gsl_vector_get (s->f, 1));
        return 0;
}

int main (void)
{
        const gsl_multiroot_fsolver_type *T;
        gsl_multiroot_fsolver *s;

        int status;
        size_t i, iter = 0;

        const size_t n = 2;
        struct rparams p = {1.0, 10.0};
        gsl_multiroot_function f = {&rosenbrock_f, n, &p};

        //double x_init[2] = {-10.0, 0.0};
        //double x_init[2] = {-3.0, 1.0};
        //double x_init[2] = {0.0, 5.0};
        double x_init[2] = {5.0, 0.0};
        gsl_vector *x = gsl_vector_alloc (n);

        gsl_vector_set (x, 0, x_init[0]);
        gsl_vector_set (x, 1, x_init[1]);

        T = gsl_multiroot_fsolver_hybrids;
        s = gsl_multiroot_fsolver_alloc (T, 2);
        gsl_multiroot_fsolver_set (s, &f, x);

        print_state (iter, s);
```

```
        do
        {
            iter++;
            status = gsl_multiroot_fsolver_iterate (s);

            print_state (iter, s);

            if (status) /* check if solver is stuck */
            break;

            status = gsl_multiroot_test_residual (s->f, 1e-7);
        }
        while (status == GSL_CONTINUE && iter < 1000);

        printf ("status = %s\n", gsl_strerror (status));

        gsl_multiroot_fsolver_free (s);
        gsl_vector_free (x);

        return 0;
    }
```

**C++ Code 224:** *main.cpp "Compute Critical Points for Nonlinear System of Equations"*

To compile it, type:
**g++ -o main main.cpp -lgsl -lcblas**
**./main**

or with Makefile, type:
**make**
**./main**

```
n =   0 x =  5.000 y =  0.000    f1(x,y) =  5.000e+02 f2(x,y) = -0.000e+00
n =   1 x =  3.333 y =  0.000    f1(x,y) =  1.481e+02 f2(x,y) = -0.000e+00
n =   2 x =  2.632 y =  0.000    f1(x,y) =  7.290e+01 f2(x,y) = -0.000e+00
n =   3 x =  1.952 y =  0.000    f1(x,y) =  2.974e+01 f2(x,y) = -0.000e+00
n =   4 x =  1.483 y =  0.000    f1(x,y) =  1.305e+01 f2(x,y) = -0.000e+00
n =   5 x =  1.117 y =  0.000    f1(x,y) =  5.572e+00 f2(x,y) = -0.000e+00
n =   6 x =  0.844 y =  0.000    f1(x,y) =  2.404e+00 f2(x,y) = -0.000e+00
n =   7 x =  0.637 y =  0.000    f1(x,y) =  1.033e+00 f2(x,y) = -0.000e+00
n =   8 x =  0.481 y =  0.000    f1(x,y) =  4.445e-01 f2(x,y) = -0.000e+00
n =   9 x =  0.363 y =  0.000    f1(x,y) =  1.912e-01 f2(x,y) = -0.000e+00
n =  10 x =  0.274 y =  0.000    f1(x,y) =  8.224e-02 f2(x,y) = -0.000e+00
n =  11 x =  0.207 y =  0.000    f1(x,y) =  3.538e-02 f2(x,y) = -0.000e+00
n =  12 x =  0.156 y =  0.000    f1(x,y) =  1.522e-02 f2(x,y) = -0.000e+00
n =  13 x =  0.118 y =  0.000    f1(x,y) =  6.546e-03 f2(x,y) = -0.000e+00
n =  14 x =  0.089 y =  0.000    f1(x,y) =  2.816e-03 f2(x,y) = -0.000e+00
n =  15 x =  0.067 y =  0.000    f1(x,y) =  1.211e-03 f2(x,y) = -0.000e+00
n =  16 x =  0.051 y =  0.000    f1(x,y) =  5.210e-04 f2(x,y) = -0.000e+00
n =  17 x =  0.038 y =  0.000    f1(x,y) =  2.241e-04 f2(x,y) = -0.000e+00
n =  18 x =  0.029 y =  0.000    f1(x,y) =  9.641e-05 f2(x,y) = -0.000e+00
n =  19 x =  0.022 y =  0.000    f1(x,y) =  4.147e-05 f2(x,y) = -0.000e+00
n =  20 x =  0.016 y =  0.000    f1(x,y) =  1.784e-05 f2(x,y) = -0.000e+00
n =  21 x =  0.012 y =  0.000    f1(x,y) =  7.674e-06 f2(x,y) = -0.000e+00
n =  22 x =  0.009 y =  0.000    f1(x,y) =  3.301e-06 f2(x,y) = -0.000e+00
n =  23 x =  0.007 y =  0.000    f1(x,y) =  1.420e-06 f2(x,y) = -0.000e+00
n =  24 x =  0.005 y =  0.000    f1(x,y) =  6.108e-07 f2(x,y) = -0.000e+00
n =  25 x =  0.004 y =  0.000    f1(x,y) =  2.627e-07 f2(x,y) = -0.000e+00
n =  26 x =  0.003 y =  0.000    f1(x,y) =  1.130e-07 f2(x,y) = -0.000e+00
n =  27 x =  0.002 y =  0.000    f1(x,y) =  4.862e-08 f2(x,y) = -0.000e+00
status = success
```

**Figure 23.206:** *The computation to obtain the critical points for nonlinear system of equations $f(x,y) = x^4 + y^4$ and $g(x,y) = x^4 - y^4$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Solution of Nonlinear System of Equations to Determine Critical Points with GSL Exercise 19/**).*

The second code is used to classify the critical points / the stationary points with **SymbolicC++** by using the second derivative test, instead of Hessian matrix.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>

using namespace std;

#define R 2 // number of rows
#define C 2 // number of columns

double division(double x, double y)
{
	return x/y;
}

// Thanks Sentinel for this
Symbolic power(Symbolic x, double n)
{
	Symbolic a = x;
	for (int i = 1; i <n; i++)
	{
```

```
                x = x * a;
        }
        return x;
}

// Driver program
int main()
{
        Symbolic x("x"), y("y");
        Symbolic f, fx, fy, fxx, fxy, fyy;
        Symbolic g, gx, gy, gxx, gxy, gyy;

        f = (x*x*x*x) + (y*y*y*y);
        fx = df(f, x);
        fy = df(f, y);
        fxx = df(fx, x);
        fxy = df(fx, y);
        fyy = df(fy, y);
        g = (x^4) — power(y,4);
        gx = df(g, x);
        gy = df(g, y);
        gxx = df(gx, x);
        gxy = df(gx, y);
        gyy = df(gy, y);

        cout << "f : " << f << endl;
        cout << "fx : " << fx << endl;
        cout << "fy : " << fy << endl;
        cout << "fxx : " << fxx << endl;
        cout << "fxy : " << fxy << endl;
        cout << "fyy : " << fyy << endl;

        cout << "\ng : " << g << endl;
        cout << "gx : " << gx << endl;
        cout << "gy : " << gy << endl;
        cout << "gxx : " << gxx << endl;
        cout << "gxy : " << gxy << endl;
        cout << "gyy : " << gyy << endl;

        // Construct a symbolic matrix of size 2 X 2
        Matrix<Symbolic> Hf_mat(2,2);
        Hf_mat[0][0] = fxx; Hf_mat[0][1] = fxy;
        Hf_mat[1][0] = fxy; Hf_mat[1][1] = fyy;
        cout << "\nH(x,y) for f:\n" << Hf_mat <<endl;
        Matrix<Symbolic> Hg_mat(2,2);
        Hg_mat[0][0] = gxx; Hf_mat[0][1] = gxy;
        Hg_mat[1][0] = gxy; Hf_mat[1][1] = gyy;
        cout << "\nH(x,y) for g:\n" << Hg_mat <<endl;
```

```
        cout << endl;

        // To find critical point set fx and fy equal to zero and solve the
            equations
        cout << "fxx * fyy - fxy^2 = " << fxx * fyy - fxy * fxy << endl;
        cout << "gxx * gyy - gxy^2 = " << gxx * gyy - gxy * gxy << endl;
        cout << "\nThe second derivative test at (0,0).\n" << endl;
        cout << "fxx * fyy - fxy^2 = " << fxx[x==0,y==0] * fyy[x==0,y==0] -
             fxy[x==0,y==0]*fxy[x==0,y==0] << endl;
        cout << "gxx * gyy - gxy^2 = " << gxx[x==0,y==0] * gyy[x==0,y==0] -
             gxy[x==0,y==0]*gxy[x==0,y==0] << endl;

        cout << endl;

        return 0;
    }
```

**C++ Code 225:** *main.cpp "Classify Critical Point with second derivative test"*

To compile it, type:
**g++ -o main main.cpp -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

```
f : x^(4)+y^(4)
fx : 4*x^(3)
fy : 4*y^(3)
fxx : 12*x^(2)
fxy : 0
fyy : 12*y^(2)

g : x^(4)-y^(4)
gx : 4*x^(3)
gy : -4*y^(3)
gxx : 12*x^(2)
gxy : 0
gyy : -12*y^(2)

H(x,y) for f:
[12*x^(2)     0   ]
[    0    12*y^(2)]


H(x,y) for g:
[12*x^(2)     0   ]
[    0        0   ]


fxx * fyy - fxy^2 = 144*x^(2)*y^(2)
gxx * gyy - gxy^2 = -144*x^(2)*y^(2)

The second derivative test at (0,0).

fxx * fyy - fxy^2 = 0
gxx * gyy - gxy^2 = 0
```

**Figure 23.207:** *The computation to classify the critical point for nonlinear system of equations $f(x,y) = x^4 + y^4$ and $g(x,y) = x^4 - y^4$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Classify Relative Extrema using Second Derivative Test with SymbolicC++ Exercise 19/**).*

## xiv. Geometric Interpretation of Sphere and Complex Number

A sphere in $\mathbb{R}^3$ is represented by this equation

$$x^2 + y^2 + z^2 = 1 \tag{23.227}$$

with the radius of the sphere is 1 toward all the main axis: $x, y$, and $z$ axis. Now we want to see the equation in explicit term with independent variables of $x$ and $y$, it will be

$$\begin{aligned} z^2 &= 1 - x^2 - y^2 \\ -z^2 &= x^2 + y^2 - 1 \\ z^2 &= (-1)(x^2 + y^2 - 1) \\ z &= \sqrt{-1}\sqrt{x^2 + y^2 - 1} \end{aligned}$$

why we need to make $z$ in $i = \sqrt{-1}$ term? Because if we have that term of complex number, then the other multiplication for the right hand side of $z$ can be seen as hypothenuse minus a bit of length (In pythagoras to compute the hypothenuse you will have the square root of the sum of the adjacent squared and opposite squared).

What happens when the adjacent and opposite is 0? There will be no right side triangle that can be seen by human eyes or can be drawn on paper / $\mathbb{R}^2$. But we can still construct the sphere

$$\begin{aligned} z &= \sqrt{-1}\sqrt{0^2 + 0^2 - 1} \\ &= \sqrt{-1}\sqrt{-1} \\ z &= i \cdot i \end{aligned}$$

we know that the square root of all numbers, real or even complex will have two definitions

$$z = \begin{cases} -1 \\ 1 \end{cases}$$

this is in accordance with the sphere with center at $(0, 0, 0)$, when $x = 0$ and $y = 0$ the values for $z$ are $z = 1$ and $z = -1$.

## XIV. Hermitian, Unitary, and Normal Matrices

[DF*] The transpose operation is less important for complex matrices than for real matrices.

**Definition 23.59: Conjugate Transpose**

If $A$ is a complex matrix, then the conjugate transpose of $A$, denoted by $A^*$, is defined by

$$A^* = \overline{A}^T \tag{23.228}$$

[DF*] The order in which the transpose and conjugation operations are performed in computing $A^* = \overline{A}^T$ does not matter

$$(\overline{A^T}) = (\overline{A})^T$$

[DF*] In the case where $A$ has real entries we have

$$A^* = (\overline{A})^T = A^T$$

so $A^*$ is the same as $A^T$ for real matrices.

[DF*] **Example:**
Find the conjugate transpose $A^*$ of the matrix

$$A = \begin{bmatrix} 2+i & -3i & 0 \\ 3 & 1-5i & i \end{bmatrix}$$

**Solution:**
We have

$$\overline{A} = \begin{bmatrix} 2-i & 3i & 0 \\ 3 & 1+5i & -i \end{bmatrix}$$

and hence

$$A^* = \overline{A}^T = \begin{bmatrix} 2-i & 3 \\ 3i & 1+5i \\ 0 & -i \end{bmatrix}$$

**Theorem 23.132: Algebraic Properties of the Conjugate Transpose Operation**

If $k$ is a complex scalar, and if $A, B$, and $C$ are complex matrices whose sizes are such that the stated operations can be performed, then:

(a) $(A^*)^* = A$

(b) $(A+B)^* = A^* + B^*$

(c) $(A-B)^* = A^* - B^*$

(d) $(kA)^* = \overline{k}A^*$

(e) $(AB)^* = B^*A^*$

[DF*] **Remark**
Note that the relationship

$$\boldsymbol{u} \cdot \boldsymbol{v} = \overline{\boldsymbol{v}}^T \boldsymbol{u}$$

can be expressed in terms of the conjugate transpose as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v}^* \boldsymbol{u} \tag{23.229}$$

**Definition 23.60: Hermitian and Unitary Matrices**

A square complex matrix $A$ is said to be unitary if

$$A^{-1} = A^* \tag{23.230}$$

and is said to be Hermitian if

$$A^* = A \tag{23.231}$$

[DF*] If $A$ is a real matrix, then

$$A^* = A^T$$

in which case, the unitary matrix formula for real matrix becomes

$$A^{-1} = A^T$$

and the Hermitian matrix formula for real matrix becomes

$$A^T = A$$

The unitary matrices are complex generalizations of the real orthogonal matrices and Hermitian matrices are complex generalizations of the real symmetric matrices.

[DF*] Hermitian matrices are easy to recognize because their diagonal entries are real, and the entries that are symmetrically positioned across the main diagonal are complex conjugates. For example,

$$A = \begin{bmatrix} 2 & 3i & 2+6i \\ -3i & -6 & 5-7i \\ 2-6i & 5+7i & 1 \end{bmatrix}$$

is Hermitian.

**Theorem 23.133: Eigenvalues for Hermitian Matrix**

The eigenvalues of a Hermitian matrix are real numbers.

**Theorem 23.134: Eigenvectors for Hermitian Matrix**

If $A$ is a Hermitian matrix, then eigenvectors from different eigenspaces are orthogonal.

**Theorem 23.135: Equivalent Statements for Unitary Matrix**

If $A$ is an $n \times n$ matrix with complex entries, then the following are equivalent.

(a) $A$ is unitary.

(b) $||A\boldsymbol{x}|| = ||\boldsymbol{x}||$ for all $\boldsymbol{x}$ in $\mathbb{C}^n$.

(c) $A\boldsymbol{x} \cdot A\boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{y}$ for all $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{C}^n$.

(d) The column vectors of $A$ form an orthonormal set in $\mathbb{C}^n$ with respect to the complex Euclidean inner product.

(e) The row vectors of $A$ form an orthonormal set in $\mathbb{C}^n$ with respect to the complex Euclidean inner product.

[DF*] Unitary matrices are the complex analogs of the real orthogonal matrices.

**Definition 23.61: Unitary Diagonalizability**

A square complex matrix is said to be unitarily diagonalizable if there is a unitary matrix $P$ such that $P^*AP = D$ is a complex diagonal matrix. Any such matrix $P$ is said to unitarily diagonalize $A$.

**Theorem 23.136: Unitary Diagonalizability for Hermitian Matrix**

Every $n \times n$ Hermitian matrix $A$ has an orthonormal set of $n$ eigenvectors and is unitarily diagonalized by any $n \times n$ matrix $P$ whose column vectors form an orthonormal set of eigenvectors of $A$.

[DF*] The procedure for unitarily diagonalizing Hermitian matrix $A$ is exactly the same as that for orthogonally diagonalizing a symmetric matrix:

1. Find a basis for each eigenspace of $A$.
2. Apply the Gram-Schmidt process to each of these bases to obtain orthonormal bases for the eigenspaces.
3. Form the matrix $P$ whose column vectors are the basis vectors obtained in Step 2. This will be a unitary matrix and will unitarily diagonalize $A$.

[DF*] We defined a square matrix with real entries to be skew-symmetric if

$$A^T = -A$$

A skew-symmetric matrix must have zeros on the main diagonal, and each entry off the main diagonal must be the negative of its mirror image about the main diagonal. Here is an example.

$$A = \begin{bmatrix} 0 & 3 & -5 \\ -3 & 0 & 6 \\ 5 & -6 & 0 \end{bmatrix}$$

[DF*] The complex analogs of the skew-symmetric matrices are the matrices for which

$$A^* = -A$$

Such matrices are said to be skew-Hermitian.

A skew-Hermitian matrix $A$ has the property

$$A^* = \overline{A}^T = -A$$

it must be that $A$ has zeros or pure imaginary numbers on the main diagonal, and that the complex conjugate of each entry off the main diagonal is the negative of the real number of its mirror image about the main diagonal. Here is an example.

$$A = \begin{bmatrix} i & 2-i & 6 \\ -2-1 & 3i & 4i \\ -6 & 4i & 0 \end{bmatrix}$$

[DF*] Hermitian matrices enjoy many, but not all, of the properties of real symmetric matrices. For example, we know that real symmetric matrices are orthogonally diagonalizable and Hermitian matrices are unitarily diagonalizable. However, whereas the real symmetric matrices are the only orthogonally diagonalizable matrices, the Hermitian matrices do not constitute the entire class of unitarily diagonalizable complex matrices; that is, there exist unitarily diagonalizable matrices that are not Hermitian.

Specifically, it can be proved that a square complex matrix $A$ is unitarily diagonalizable if and only if

$$AA^* = A^*A \tag{23.232}$$

Matrices with this property are said to be normal. Normal matrices include the Hermitian, skew-Hermitian, and unitary matrices in the complex case and the symmetric, skew-symmetric, and orthogonal matrices in the real case. The nonzero skew-symmetric matrices are particularly interesting because they are examples of real matrices that are not orthogonally diagonalizable but are unitarily diagonalizable.

[DF*] We have seen that Hermitian matrices have real eigenvalues.

The eigenvalues of skew-Hermitian matrix are either zero or purely imaginary (have real part of zero) and that the eigenvalues of unitary matrices have modulus 1, $|\lambda| = 1$.

**Figure 23.208**

[DF*] **Example:**
Show that the matrix

$$A = \frac{1}{\sqrt{2}} \begin{bmatrix} e^{i\theta} & e^{-i\theta} \\ ie^{i\theta} & -ie^{-i\theta} \end{bmatrix}$$

is unitary for all real values of $\theta$.
**Solution:**
A square matrix $A$ is said to be unitary if

$$A^{-1} = A^* = \overline{A}^T$$

thus

$$A = \frac{1}{\sqrt{2}} \begin{bmatrix} e^{i\theta} & e^{-i\theta} \\ ie^{i\theta} & -ie^{-i\theta} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos\ \theta + i \sin\ \theta & \cos\ \theta - i \sin\ \theta \\ -\sin\ \theta + i \cos\ \theta & -\sin\ \theta - i \cos\ \theta \end{bmatrix}$$

then

$$\overline{A}^T = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos\ \theta - i \sin\ \theta & -\sin\ \theta - i \cos\ \theta \\ \cos\ \theta + i \sin\ \theta & -\sin\ \theta + i \cos\ \theta \end{bmatrix}$$

we will have

$$A^* A = \frac{1}{\sqrt{2}} \begin{bmatrix} e^{-i\theta} & -ie^{-i\theta} \\ e^{i\theta} & ie^{i\theta} \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} e^{i\theta} & e^{-i\theta} \\ ie^{i\theta} & -ie^{-i\theta} \end{bmatrix}$$

$$A^* A = \frac{1}{2} \begin{bmatrix} e^{-i\theta} e^{i\theta} + (-ie^{-i\theta})(ie^{i\theta}) & e^{-i\theta} e^{-i\theta} + (-ie^{-i\theta})(-ie^{-i\theta}) \\ e^{i\theta} e^{i\theta} + (ie^{i\theta})(ie^{i\theta}) & e^{i\theta} e^{-i\theta} + (ie^{i\theta})(-ie^{-i\theta}) \end{bmatrix}$$

$$A^* A = \frac{1}{2} \begin{bmatrix} 1+1 & e^{-2i\theta} - e^{-2i\theta} \\ e^{2i\theta} - e^{2i\theta} & 1+1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

It is proven that

$$A^* A = I$$

it means that $A^* = A^{-1}$ and that $A$ is unitary.

Now for anyone who try or wondering with another way to prove, we cannot apply the inverse formula that is using determinants for matrix with complex entries, because if we do, it will fail, suppose we try to compute $A^{-1}$ with the the inverse formula

$$A^{-1} = \frac{1}{\sqrt{2}} \frac{1}{e^{i\theta}(-ie^{-i\theta}) - e^{-i\theta}(ie^{i\theta})} \begin{bmatrix} -ie^{-i\theta} & -e^{-i\theta} \\ -ie^{i\theta} & e^{i\theta} \end{bmatrix}$$

$$A^{-1} = \frac{1}{\sqrt{2}} \frac{1}{-2i} \begin{bmatrix} -ie^{-i\theta} & -e^{-i\theta} \\ -ie^{i\theta} & e^{i\theta} \end{bmatrix} = \frac{1}{\sqrt{2}} \frac{1}{2} \begin{bmatrix} e^{-i\theta} & \frac{-e^{-i\theta}}{-i} \\ e^{i\theta} & \frac{e^{i\theta}}{-i} \end{bmatrix}$$

since

$$\frac{-e^{-i\theta}}{-i} = -\sin\ \theta - i\cos\ \theta$$
$$\frac{e^{i\theta}}{-i} = \cos\ \theta + i\sin\ \theta$$

so we will have

$$A^{-1} = \frac{1}{\sqrt{2}} \frac{1}{2} \begin{bmatrix} \cos\ \theta - i\sin\ \theta & i(\cos\ \theta - i\sin\ \theta) \\ \cos\ \theta + i\sin\ \theta & -\sin\ \theta + i\cos\ \theta \end{bmatrix}$$

it still won't make $A^* = A^{-1}$. Thus, always remember that we cannot apply inverse formula for real matrix for a matrix with complex entries, the formula to find complex matrix' inverse is:

$$A^{-1} = \overline{A}^T = A^*$$

the inverse of a complex matrix is its conjugate transpose. Thus, it is said to be unitary if the equality above holds.

[DF*] **Example:**
Prove that each entry on the main diagonal of a skew-Hermitian matrix is either zero or a pure imaginary number.

**Solution:**
A skew-Hermitian matrix will obey this formula

$$A^* = -A$$

or

$$A = -A^*$$

when we compute $A^*$ we will compute the conjugate transpose of a matrix $A$. So we will have the negative of the complex entries for all entries in matrix $A$.

Then, we multiply the conjugate transpose of matrix $A$ with $-1$ ($-A^*$), so we will obtain $A$ back again only if the diagonal entries are zero or complex numbers, because conjugate tranpose computation will turn a complex number of $a + bi$ into $a - bi$, with $a, b \in \mathbb{R}$.

Besides the diagonal entries, the entries that are symmetrically positioned across the main diagonal are complex conjugates.

[DF*] **Example:**
Let $A$ be any $n \times n$ matrix with complex entries, and define the matrices $B$ and $C$ to be

$$B = \frac{1}{2}(A + A^*)$$

and

$$C = \frac{1}{2i}(A - A^*)$$

(a) Show that $B$ and $C$ are Hermitian.

(b) Show that $A = B + iC$ and $A^* = B - iC$.

(c) What condition must $B$ and $C$ satisfy for $A$ to be normal?

**Solution:**

(a) Hermitian matrices have real entries in their diagonal, and the entries that are symmetrically positioned across the main diagonal are complex conjugates.

If $A$ is an $n \times n$ matrix with complex entries that means for dimension of 2

$$A = \begin{bmatrix} a_1 + b_1 i & a_2 + b_2 i \\ a_3 + b_3 i & a_4 + b_4 i \end{bmatrix}$$

$$A^* = \begin{bmatrix} a_1 - b_1 i & a_3 - b_3 i \\ a_2 - b_2 i & a_4 - b_4 i \end{bmatrix}$$

and for dimension of 3

$$A = \begin{bmatrix} a_1 + b_1 i & a_2 + b_2 i & a_3 + b_3 i \\ a_4 + b_4 i & a_5 + b_5 i & a_6 + b_6 i \\ a_7 + b_7 i & a_8 + b_8 i & a_9 + b_9 i \end{bmatrix}$$

$$A^* = \begin{bmatrix} a_1 - b_1 i & a_4 - b_4 i & a_7 - b_7 i \\ a_2 - b_2 i & a_5 - b_5 i & a_8 - b_8 i \\ a_3 - b_3 i & a_6 - b_6 i & a_9 - b_9 i \end{bmatrix}$$

with generalization to $n$ dimension we can tell that:

If $B = \frac{1}{2}(A + A^*)$ then the diagonal entries of $B$ will be real entries, and since the conjugate transpose operation will move the entry to the position that is symmetrically across it, vice versa, thus the entries that are symmetrically positioned across the main diagonal will be complex conjugates. Thus it is proven that

$$B^* = B$$

If $C = \frac{1}{2i}(A - A^*)$ then the diagonal entries of $C$ will be real entries as well

$$\begin{aligned} \frac{1}{2i}\left[(a_j + b_j i) - (a_j - b_j i)\right] &= \frac{1}{2i}\left[2b_j i\right] \\ &= b_j \end{aligned}$$

with $j = 1, 2, \ldots, n$ as the indices of the diagonal matrix entries. Moving on to the entries off the diagonal

$$\frac{1}{2i}\left[(a_k + b_k i) - (a_l - b_l i)\right] = \frac{1}{2i}\left[(a_k - a_l) + (b_k - b_l)i\right]$$

and the mirror of this entry (the one symmetrically positioned across the main diagonal toward this)

$$\frac{1}{2i}\left[(a_l + b_l i) - (a_k - b_k i)\right] = \frac{1}{2i}\left[(a_l - a_k) + (b_l - b_k)i\right]$$

with $j, k = 1, 2, \ldots, n$ and $j \neq k$ as the indices of the entries off the main diagonal of the matrix.

We know that

$$\frac{1}{2i}\left[(a_k - a_l) + (b_k - b_l)i\right]$$

is the complex conjugate of

$$\frac{1}{2i}\left[(a_l - a_k) + (b_l - b_k)i\right]$$

so the diagonal entries of $C$ will be real entries, and the entries that are symmetrically positioned across the main diagonal will be complex conjugates as well. Thus it is proven that

$$C^* = C$$

(b) We will use standard elimination operation

$$\begin{aligned} B &= \frac{1}{2}(A + A^*) \\ 2B &= A + A^* \end{aligned}$$

$$\begin{aligned} C &= \frac{1}{2i}(A - A^*) \\ 2iC &= A - A^* \end{aligned}$$

hence

$$\begin{aligned} 2A &= 2B + 2iC \\ A &= B + iC \end{aligned}$$

and

$$\begin{aligned} 2A^* &= 2B - 2iC \\ A^* &= B - iC \end{aligned}$$

(c) $A$ will be normal if and only if

$$AA^* = A^*A$$

Now if we substitute with $B$ and $C$

$$\begin{aligned} (B + iC)(B - iC) &= (B - iC)(B + iC) \\ BB - BiC + iCB + CC &= BB + BiC - iCB + CC \end{aligned}$$

this means that

$$\begin{aligned} -BiC + iCB &= BiC - iCB \\ 2iCB &= 2BiC \\ iCB &= iBC \\ CB &= BC \end{aligned}$$

so $B$ and $C$ have to be in these conditions:

$$B = C^{-1}$$

or $CB = BC$ commute or

$$B = C^* \quad \text{or} \quad C = B^*$$

$B$ and $C$ are Hermitian and unitarily diagonalizable.

[DF*] **Example:**
Show that if $A$ is an $n \times n$ matrix with complex entries, and if $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{C}^n$ that are expressed in column form, then

$$A\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \cdot A^*\boldsymbol{v}$$

and

$$\boldsymbol{u} \cdot A\boldsymbol{v} = A^*\boldsymbol{u} \cdot \boldsymbol{v}$$

**Solution:**
We know that the relationship

$$\boldsymbol{u} \cdot \boldsymbol{v} = \overline{\boldsymbol{v}}^T\boldsymbol{u}$$

can be expressed in conjugate transpose as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v}^*\boldsymbol{u}$$

If we expand the components of $\boldsymbol{u}$ and $\boldsymbol{v}$ like this

$$\boldsymbol{u} = \begin{bmatrix} u_{1a} + iu_{1b} \\ u_{2a} + iu_{2b} \\ \vdots \\ u_{na} + iu_{nb} \end{bmatrix}$$

$$\boldsymbol{v} = \begin{bmatrix} v_{1a} + iv_{1b} \\ v_{2a} + iv_{2b} \\ \vdots \\ v_{na} + iv_{nb} \end{bmatrix}$$

and

$$A = \begin{bmatrix} a_{11} + ib_{11} & a_{12} + ib_{12} & \dots & a_{1n} + ib_{1n} \\ a_{21} + ib_{21} & a_{22} + ib_{22} & \dots & a_{2n} + ib_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} + ib_{n1} & a_{n2} + ib_{n2} & \dots & a_{nn} + ib_{nn} \end{bmatrix}$$

$$A^* = \begin{bmatrix} a_{11} - ib_{11} & a_{21} - ib_{21} & \dots & a_{n1} - ib_{n1} \\ a_{12} - ib_{12} & a_{22} - ib_{22} & \dots & a_{n2} - ib_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} - ib_{1n} & a_{2n} - ib_{2n} & \dots & a_{nn} - ib_{nn} \end{bmatrix}$$

$$A\boldsymbol{u} = \begin{bmatrix} (a_{11} + ib_{11})(u_{1a} + iu_{1b}) + (a_{12} + ib_{12})(u_{2a} + iu_{2b}) + \cdots + (a_{1n} + ib_{1n})(u_{na} + iu_{nb}) \\ (a_{21} + ib_{21})(u_{1a} + iu_{1b}) + (a_{22} + ib_{22})(u_{2a} + iu_{2b}) + \cdots + (a_{2n} + ib_{2n})(u_{na} + iu_{nb}) \\ \vdots \\ (a_{n1} + ib_{n1})(u_{1a} + iu_{1b}) + (a_{n2} + ib_{n2})(u_{2a} + iu_{2b}) + \cdots + (a_{nn} + ib_{nn})(u_{na} + iu_{nb}) \end{bmatrix}$$

$$A\boldsymbol{u} \cdot \boldsymbol{v} = \begin{matrix} [(a_{11} + ib_{11})(u_{1a} + iu_{1b}) + (a_{12} + ib_{12})(u_{2a} + iu_{2b}) + \cdots + (a_{1n} + ib_{1n})(u_{na} + iu_{nb})][v_{1a} - iv_{1b}] + \\ [(a_{21} + ib_{21})(u_{1a} + iu_{1b}) + (a_{22} + ib_{22})(u_{2a} + iu_{2b}) + \cdots + (a_{2n} + ib_{2n})(u_{na} + iu_{nb})][v_{2a} - iv_{2b}] + \\ \vdots \quad + \\ [(a_{n1} + ib_{n1})(u_{1a} + iu_{1b}) + (a_{n2} + ib_{n2})(u_{2a} + iu_{2b}) + \cdots + (a_{nn} + ib_{nn})(u_{na} + iu_{nb})][v_{na} - iv_{nb}] \end{matrix}$$

now for the right hand side

$$A^*\boldsymbol{v} = \begin{bmatrix} (a_{11} - ib_{11})(v_{1a} + iv_{1b}) + (a_{21} - ib_{21})(v_{2a} + iv_{2b}) + \cdots + (a_{n1} - ib_{n1})(v_{na} + iv_{nb}) \\ (a_{12} - ib_{12})(v_{1a} + iv_{1b}) + (a_{22} - ib_{22})(v_{2a} + iv_{2b}) + \cdots + (a_{n2} - ib_{n2})(v_{na} + iv_{nb}) \\ \vdots \\ (a_{1n} - ib_{1n})(v_{1a} + iv_{1b}) + (a_{2n} - ib_{2n})(v_{2a} + iv_{2b}) + \cdots + (a_{nn} - ib_{nn})(v_{na} + iv_{nb}) \end{bmatrix}$$

$$\boldsymbol{u} \cdot A^*\boldsymbol{v} = \begin{bmatrix} [u_{1a} + iu_{1b}][\overline{(a_{11} - ib_{11})(v_{1a} + iv_{1b}) + (a_{21} - ib_{21})(v_{2a} + iv_{2b}) + \cdots + (a_{n1} - ib_{n1})(v_{na} + iv_{nb})}] \\ [u_{2a} + iu_{2b}][\overline{(a_{12} - ib_{12})(v_{1a} + iv_{1b}) + (a_{22} - ib_{22})(v_{2a} + iv_{2b}) + \cdots + (a_{n2} - ib_{n2})(v_{na} + iv_{nb})}] \\ \vdots \\ [u_{na} + iu_{nb}][\overline{(a_{1n} - ib_{1n})(v_{1a} + iv_{1b}) + (a_{2n} - ib_{2n})(v_{2a} + iv_{2b}) + \cdots + (a_{nn} - ib_{nn})(v_{na} + iv_{nb})}] \end{bmatrix}$$

$$\boldsymbol{u} \cdot A^*\boldsymbol{v} = \begin{matrix} [u_{1a} + iu_{1b}][(a_{11} + ib_{11})(v_{1a} - iv_{1b}) + (a_{21} + ib_{21})(v_{2a} - iv_{2b}) + \cdots + (a_{n1} + ib_{n1})(v_{na} - iv_{nb})] + \\ [u_{2a} + iu_{2b}][(a_{12} + ib_{12})(v_{1a} - iv_{1b}) + (a_{22} + ib_{22})(v_{2a} - iv_{2b}) + \cdots + (a_{n2} + ib_{n2})(v_{na} - iv_{nb})] + \\ \vdots \quad + \\ [u_{na} + iu_{nb}][(a_{1n} + ib_{1n})(v_{1a} - iv_{1b}) + (a_{2n} + ib_{2n})(v_{2a} - iv_{2b}) + \cdots + (a_{nn} + ib_{nn})(v_{na} - iv_{nb})] \end{matrix}$$

$$\boldsymbol{u} \cdot A^*\boldsymbol{v} = \begin{matrix} [(a_{11} + ib_{11})(u_{1a} + iu_{1b}) + (a_{12} + ib_{12})(u_{2a} + iu_{2b}) + \cdots + (a_{1n} + ib_{1n})(u_{na} + iu_{nb})][v_{1a} - iv_{1b}] + \\ [(a_{21} + ib_{21})(u_{1a} + iu_{1b}) + (a_{22} + ib_{22})(u_{2a} + iu_{2b}) + \cdots + (a_{2n} + ib_{2n})(u_{na} + iu_{nb})][v_{2a} - iv_{2b}] + \\ \vdots \quad + \\ [(a_{n1} + ib_{n1})(u_{1a} + iu_{1b}) + (a_{n2} + ib_{n2})(u_{2a} + iu_{2b}) + \cdots + (a_{nn} + ib_{nn})(u_{na} + iu_{nb})][v_{na} - iv_{nb}] \end{matrix}$$

it is proven then that

$$A\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \cdot A^*\boldsymbol{v}$$

To prove

$$\boldsymbol{u} \cdot A\boldsymbol{v} = A^*\boldsymbol{u} \cdot \boldsymbol{v}$$

will be the reader' homework, it is the same method as the one above.

[DF*] **Example:**

Show that if $A$ is a unitary matrix, then so is $A^*$.

**Solution:**

If $A$ is unitary then

$$A^{-1} = A^* = \overline{A}^T$$

Now, we will check whether $A^*$ is unitary or not

$$\overline{A^*}^T = (A^{-1})^{-1} = A$$

thus $A^*$ is also a unitary matrix.

[DF*] **Example:**
Show that the eigenvalues of a skew-Hermitian matrix are either zero or purely imaginary.

**Solution:**
A skew-Hermitian matrix $A$ has the property

$$A^* = \overline{A}^T = -A$$

$A$ has zeros or pure imaginary numbers on the main diagonal, and the complex conjugate of each entry off the main diagonal is the negative of its mirror image about the main diagonal.

We know that skew-Hermitian matrix is normal, suppose that $\boldsymbol{v}$ is an eigenvector of matrix $A$, thus

$$\begin{aligned} AA^* &= A^*A \\ AA^*\boldsymbol{v} &= A^*A\boldsymbol{v} \\ A(-A\boldsymbol{v}) &= A^*(A\boldsymbol{v}) \\ A(-\lambda_1\boldsymbol{v}) &= A^*(\lambda\boldsymbol{v}) \\ A(-\lambda_1\boldsymbol{v}) &= -A(\lambda\boldsymbol{v}) \end{aligned}$$

In order for

$$-\lambda = \lambda$$

the eigenvalue needs to be either

$$\lambda = bi$$

with $b$ as scalar, or

$$\lambda = 0$$

If we want to compute the eigenvalues of $A$, we will use the characteristic polynomial formula

$$\det(\lambda I - A) = 0$$

If $A$ has zeros or pure imaginary numbers then we will know that the characteristic polynomial will be like this

$$\det(\lambda I - A) = (\lambda - a_1 i)(\lambda - a_2 i)\dots(\lambda - 0)(\lambda - 0)\dots(c_1 + d_1 i)(c_2 + d_2 i)\dots$$

there will be $n$ terms of factors consisting of $(\lambda - a_j i)$, $p$ number of $(\lambda - 0)$ and $(c_k + d_k i)$, with $j = 1, 2, \dots m$ , $k = 1, 2, \dots, l$ and $p + m + l = n$. $a_j, c_k, d_k$ are real numbers.

The factor of the complex number in the characteristic polynomial

$$(c_1 + d_1 i)(c_2 + d_2 i)\dots$$

will only adjust the scalar for the main factor of the characteristic polynomial.

[DF*] **Example:**
Show that the eigenvalues of a unitary matrix have modulus 1.

**Solution:**
A unitary matrix $A$ has the property that

$$||A\boldsymbol{x}|| = ||\boldsymbol{x}||$$

for all $x$ in $\mathbb{C}^n$.

Thus if $A$ is unitary and $A\boldsymbol{x} = \lambda\boldsymbol{x}$ where $\boldsymbol{x} \neq \boldsymbol{0}$, we must have

$$|\lambda|||\boldsymbol{x}|| = ||A\boldsymbol{x}|| = ||\boldsymbol{x}||$$

and so

$$|\lambda| = 1$$

[DF*] **Example:**
Show that if $\boldsymbol{u}$ is a nonzero vector in $\mathbb{C}^n$ that is expressed in column form, then $P = \boldsymbol{u}\boldsymbol{u}^*$ is Hermitian.

**Solution:**
Let

$$\boldsymbol{u} = \begin{bmatrix} a_1 + b_1 i \\ a_2 + b_2 i \\ \vdots \\ a_n + b_n i \end{bmatrix}$$

$$\boldsymbol{u}^* = \begin{bmatrix} a_1 - b_1 i & a_2 - b_2 i & \dots & a_n - b_n i \end{bmatrix}$$

hence

$$P = \boldsymbol{u}\boldsymbol{u}^* = \begin{bmatrix} a_1^2 + b_1^2 & a_1a_2 + b_1b_2 - a_1b_2 i + a_2b_1 i & \dots & a_1a_n + b_1b_n - a_1b_n i + a_nb_1 i \\ a_1a_2 + b_1b_2 + a_1b_2 i - a_2b_1 i & a_2^2 + b_2^2 & \dots & a_2a_n + b_2b_n - a_2b_n i + a_nb_2 i \\ \vdots & \vdots & \ddots & \vdots \\ a_1a_n + b_1b_n + a_1b_n i - a_nb_1 i & a_2a_n + b_2b_n + a_2b_n i - a_nb_2 i & \dots & a_n^2 + b_n^2 \end{bmatrix}$$

matrix $P$ will be of size $n \times n$, the diagonal entries of matrix $P$ will be real entries and the entries that are symmetrically positioned across the main diagonal are complex conjugates. Thus $P$ is Hermitian.

[DF*] **Example:**
Show that if $\boldsymbol{u}$ is a unit vector in $\mathbb{C}^n$ that is expressed in column form, then $H = 1 - 2\boldsymbol{u}\boldsymbol{u}^*$ is Hermitian and unitary.

**Solution:**

For the short proof of Hermitian matrix:
If $H = I - 2\boldsymbol{u}\boldsymbol{u}^*$, then

$$\begin{aligned} H^* &= (I - 2\boldsymbol{u}\boldsymbol{u}^*)^* \\ &= I^* - 2\boldsymbol{u}^{**}\boldsymbol{u}^* \\ &= I - 2\boldsymbol{u}\boldsymbol{u}^* = H \end{aligned}$$

The long proof that $H$ is a Hermitian matrix:
Let $\boldsymbol{u}$ be a unit vector

$$\boldsymbol{u} = \frac{1}{k}\begin{bmatrix} a_1 + b_1 i \\ a_2 + b_2 i \\ \vdots \\ a_n + b_n i \end{bmatrix}$$

Let

$$k = \sqrt{\sum_{j=1}^{n} |a_j + b_j i|^2}$$

with $j = 1, 2, \ldots, n$, or we can also write it as

$$\boldsymbol{u} = \frac{1}{||\boldsymbol{u}||}\begin{bmatrix} a_1 + b_1 i \\ a_2 + b_2 i \\ \vdots \\ a_n + b_n i \end{bmatrix}$$

now let

$$\boldsymbol{u}^* = \frac{1}{||\boldsymbol{u}^*||}\begin{bmatrix} a_1 - b_1 i & a_2 - b_2 i & \ldots & a_n - b_n i \end{bmatrix}$$

let $\gamma = \left(\frac{2}{||\boldsymbol{u}||||\boldsymbol{u}^*||}\right)$ hence matrix $H = I - 2\boldsymbol{u}\boldsymbol{u}^*$ will be

$$H = \begin{bmatrix} 1 - \gamma(a_1^2 + b_1^2) & -\gamma(a_1 a_2 + b_1 b_2 - a_1 b_2 i + a_2 b_1 i) & \ldots & -\gamma(a_1 a_n + b_1 b_n - a_1 b_n i + a_n b_1 i) \\ -\gamma(a_1 a_2 + b_1 b_2 + a_1 b_2 i - a_2 b_1 i) & 1 - \gamma(a_2^2 + b_2^2) & \ldots & -\gamma(a_2 a_n + b_2 b_n - a_2 b_n i + a_n b_2 i) \\ \vdots & \vdots & \ddots & \vdots \\ -\gamma(a_1 a_n + b_1 b_n + a_1 b_n i - a_n b_1 i) & -\gamma(a_2 a_n + b_2 b_n + a_2 b_n i - a_n b_2 i) & \ldots & 1 - \gamma(a_n^2 + b_n^2) \end{bmatrix}$$

We have proven that $\boldsymbol{u}\boldsymbol{u}^*$ is Hermitian, any scalar multiple of a Hermitian matrix is still a Hermitian matrix, consequently $H = I - 2\boldsymbol{u}\boldsymbol{u}^*$ is also a Hermitian matrix.

A Hermitian matrix has the property of

$$H^* = H$$

Now if $H$ is also a unitary matrix, then it should be

$$H^{-1} = H^*$$

and it can be proven by showing that $HH^* = H^*H = I$.
The short proof:

$$\begin{aligned} HH^* &= (I - 2\boldsymbol{u}\boldsymbol{u}^*)(I - 2\boldsymbol{u}\boldsymbol{u}^*) \\ &= I - 2\boldsymbol{u}\boldsymbol{u}^* - 2\boldsymbol{u}\boldsymbol{u}^* + 4\boldsymbol{u}\boldsymbol{u}^*\boldsymbol{u}\boldsymbol{u}^* \\ &= I - 4\boldsymbol{u}\boldsymbol{u}^* + 4\boldsymbol{u}||\boldsymbol{u}||^2\boldsymbol{u}^* \\ &= I \end{aligned}$$

thus it is proven then that $H$ is Hermitian and unitary.

For the long proof: We will decompose one by one the matrix multiplication between $H$ and $H^*$ for the entry at row 1, column 1 as an example, it is numerical linear algebra field of

study

$$
\begin{aligned}
HH_{11}^{*} &= (1-\gamma(a_1^2+b_1^2))(1-\gamma(a_1^2+b_1^2)) \\
&+ \sum_{j=2}^{n}(-\gamma(a_1a_j+b_1b_j-a_1b_ji+a_jb_1i))(-\gamma(a_1a_j+b_1b_j+a_1b_ji-a_jb_1i)) \\
&= \left(1-2\left(\gamma(a_1^2+b_1^2)\right)+\gamma^2(a_1^4+2a_1^2b_1^2+b_1^4)\right)+\gamma^2\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right) \\
&= \left(1-2\left(\left(\frac{2}{\sqrt{\sum_{j=1}^{n}|a_j+b_ji|^2}\sqrt{\sum_{j=1}^{n}|a_j-b_ji|^2}}\right)(a_1^2+b_1^2)\right)\right) \\
&+ \left(\left(\frac{2}{\sqrt{\sum_{j=1}^{n}|a_j+b_ji|^2}\sqrt{\sum_{j=1}^{n}|a_j-b_ji|^2}}\right)^2(a_1^4+2a_1^2b_1^2+b_1^4)\right) \\
&+ \left(\frac{2}{\sqrt{\sum_{j=1}^{n}|a_j+b_ji|^2}\sqrt{\sum_{j=1}^{n}|a_j-b_ji|^2}}\right)^2\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right) \\
&= \left(1-2\left(\left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)(a_1^2+b_1^2)\right)\right) \\
&+ \left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)^2\left(a_1^4+2a_1^2b_1^2+b_1^4\right) \\
&+ \left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)^2\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right) \\
&= \left(1-2\left(\left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)(a_1^2+b_1^2)\right)\right) \\
&+ \left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)^2\left(a_1^4+2a_1^2b_1^2+b_1^4\right) \\
&+ \left(\frac{2}{\sum_{j=1}^{n}a_j^2+b_j^2}\right)^2\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right) \\
&= \frac{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)-4(a_1^2+b_1^2)}{\sum_{j=1}^{n}a_j^2+b_j^2}+\frac{4\left(a_1^4+2a_1^2b_1^2+b_1^4\right)}{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)^2} \\
&+ \frac{4\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right)}{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)^2} \\
&= \frac{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)-4(a_1^2+b_1^2)}{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)^2}\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)+\frac{4\left(a_1^4+2a_1^2b_1^2+b_1^4\right)}{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)^2} \\
&+ \frac{4\left(\sum_{j=2}^{n}a_1^2a_j^2+a_1^2b_j^2+a_j^2b_1^2+b_1^2b_j^2\right)}{\left(\sum_{j=1}^{n}a_j^2+b_j^2\right)^2}
\end{aligned}
$$

$$
\begin{aligned}
HH^*_{11} &= \frac{\left(\sum_{j=1}^n a_j^2 + b_j^2\right) - 4(a_1^2 + b_1^2)}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2}\left(\sum_{j=1}^n a_j^2 + b_j^2\right) + \frac{4\left(\sum_{j=1}^n a_1^2 a_j^2 + a_1^2 b_j^2 + a_j^2 b_1^2 + b_1^2 b_j^2\right)}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2} \\
&= \frac{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2} - \frac{-4(a_1^2 + b_1^2)}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2}\left(\sum_{j=1}^n a_j^2 + b_j^2\right) + \frac{4\left(\sum_{j=1}^n a_1^2 a_j^2 + a_1^2 b_j^2 + a_j^2 b_1^2 + b_1^2 b_j^2\right)}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2} \\
&= 1 + \frac{-4\left(\sum_{j=1}^n a_1^2 a_j^2 + a_1^2 b_j^2 + a_j^2 b_1^2 + b_1^2 b_j^2\right) + 4\left(\sum_{j=1}^n a_1^2 a_j^2 + a_1^2 b_j^2 + a_j^2 b_1^2 + b_1^2 b_j^2\right)}{\left(\sum_{j=1}^n a_j^2 + b_j^2\right)^2} \\
&= 1 + 0 \\
&= 1
\end{aligned}
$$

You can continue the symbolic computation for $HH^*_{ij}$ in general, the result will be

$$
\begin{cases}
HH^*_{ij} &= 1, \quad i = j \\
HH^*_{ij} &= 0, \quad i \neq j
\end{cases}
$$

The result is that $HH^*$ is an identity matrix of size $n$, so it is proven that $H$ is a Hermitian and unitary matrix.

Just for fun, if you are willing to manually compute you can use this as the unit vector $\boldsymbol{u}$

$$
\boldsymbol{u} = \begin{bmatrix} \frac{-1-i}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}
$$

$$
\boldsymbol{u}^* = \begin{bmatrix} \frac{-1+i}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{bmatrix}
$$

you can use that vector to compute $HH^* = (I - 2\boldsymbol{u}\boldsymbol{u}^*)(I - 2\boldsymbol{u}\boldsymbol{u}^*)$ the result will be an identity matrix of size 2.

[DF*] **Example:**
What can you say about the inverse of a matrix $A$ that is both Hermitian and unitary?
**Solution:**
The inverse is an identity matrix.

[DF*] **Example:**
Under what conditions is the following matrix normal?

$$
A = \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & c \\ 0 & b & 0 \end{bmatrix}
$$

**Solution:**
The conjugate transpose of $A$ will be

$$
A^* = \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & b \\ 0 & c & 0 \end{bmatrix}
$$

matrix normal will have to obey the equation

$$
AA^* = A^*A
$$

thus

$$AA^* = \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & c \\ 0 & b & 0 \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & b \\ 0 & c & 0 \end{bmatrix} = \begin{bmatrix} a^2 & 0 & 0 \\ 0 & c^2 & 0 \\ 0 & 0 & b^2 \end{bmatrix}$$

$$A^*A = \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & b \\ 0 & c & 0 \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & 0 & c \\ 0 & b & 0 \end{bmatrix} = \begin{bmatrix} a^2 & 0 & 0 \\ 0 & b^2 & 0 \\ 0 & 0 & c^2 \end{bmatrix}$$

so the condition for matrix $A$ to be normal is that $b = c$ with $a, b, c$ are complex scalars.

[DF*] **Example:**

What geometric interpretation might you reasonably give to multiplication by the matrices $P = \boldsymbol{u}\boldsymbol{u}^*$ and $H = I - 2\boldsymbol{u}\boldsymbol{u}^*$?

**Solution:**

If $P = \boldsymbol{u}\boldsymbol{u}^*$, then

$$\begin{aligned} P\boldsymbol{x} &= (\boldsymbol{u}\boldsymbol{u}^*)\boldsymbol{x} \\ &= (\boldsymbol{u}\overline{\boldsymbol{u}}^T)\boldsymbol{x} \\ &= \boldsymbol{u}(\overline{\boldsymbol{u}}^T\boldsymbol{x}) \\ &= (\boldsymbol{x} \cdot \boldsymbol{u})\boldsymbol{u} \end{aligned}$$

Thus multiplication of $\boldsymbol{x}$ by $P$ corresponds to $||\boldsymbol{u}||^2$ times the orthogonal projection of $\boldsymbol{x}$ onto $W = \text{span}\{\boldsymbol{u}\}$.

If $||\boldsymbol{u}|| = 1$, then multiplication of $\boldsymbol{x}$ by $H = I - 2\boldsymbol{u}\boldsymbol{u}^*$ corresponds to reflection of $\boldsymbol{x}$ about the hyperplane $\boldsymbol{u}^\perp$.

[DF*] **Example:**

Prove that if $A$ is an invertible matrix, then $A^*$ is invertible, and $(A^*)^{-1} = (A^{-1})^*$.

**Solution:**

An invertible matrix $A$ has to be a unitary matrix

$$A^{-1} = A^*$$

thus

$$\begin{aligned} (A^*)^{-1} &= (A^{-1})^{-1} \\ &= A \\ &= (A^*)^* \\ &= (A^{-1})^* \end{aligned}$$

[DF*] **Example:**

(a) Prove that $\det(\overline{A}) = \overline{\det(A)}$.

(b) Use the result in part $(a)$ and the fact that a square matrix and its transpose have the same determinant to prove that $\det(A^*) = \overline{\det(A)}$.

**Solution:**

(a) Prove that $\det(\overline{A}) = \overline{\det(A)}$.

(b) Use the result in part $(a)$ and the fact that a square matrix and its transpose have the same determinant to prove that $\det(A^*) = \overline{\det(A)}$.

[DF*] **Example:**
Prove that an $n \times n$ matrix with complex entries is unitary if and only if the columns of $A$ form an orthonormal set in $\mathbb{C}^n$.
**Solution:**
Suppose that we have

$$A = \begin{bmatrix} \boldsymbol{c}_1 & \boldsymbol{c}_2 & \dots & \boldsymbol{c}_n \end{bmatrix}$$

with $\boldsymbol{c}_i$ represent the $i$-th column of matrix $A$, $i = 1, 2, \dots, n$, and they will have these properties

$$\begin{aligned} ||\boldsymbol{c}_i|| &= 1, \quad i = 1, 2, \dots, n \\ \boldsymbol{c}_i \cdot \boldsymbol{c}_j &= 1 \\ \boldsymbol{c}_i \cdot \boldsymbol{c}_j &= 0, \quad i, j = 1, 2, \dots, n \quad i \neq j \end{aligned}$$

the dot product is the complex Euclidean inner product.
Now, if we multiply $A$ with $A^*$ we will obtain

$$AA^* = \begin{bmatrix} \boldsymbol{c}_1 & \boldsymbol{c}_2 & \dots & \boldsymbol{c}_n \end{bmatrix} \begin{bmatrix} \overline{\boldsymbol{c}_1^T} \\ \overline{\boldsymbol{c}_2^T} \\ \vdots \\ \overline{\boldsymbol{c}_n^T} \end{bmatrix}$$

thus the entries of the matrix will be

$$\begin{cases} AA^*_{ij} & = 1, \quad i = j \\ AA^*_{ij} & = 0, \quad i \neq j \end{cases}$$

when we obtain the result of $AA^* = A^*A = I$ then we know that this $n \times n$ complex matrix $A$ is a unitary matrix.

## i. C++ Computation: Eigenvalues and Eigenvectors of a Hermitian Matrix

Confirm that the Hermitian matrix

$$A = \begin{bmatrix} 2 & 1+i \\ 1-i & 3 \end{bmatrix}$$

has real eigenvalues and eigenvectors from different eigenspaces are orthogonal.

**Solution:**

The characteristic polynomial of $A$ is

$$\det(\lambda I - A) = \begin{vmatrix} \lambda - 2 & -1-i \\ -1+i & \lambda - 3 \end{vmatrix} = \lambda^2 - 5\lambda + 4 = (\lambda - 1)(\lambda - 4)$$

so the eigenvalues of $A$ are

$$\begin{aligned} \lambda_1 &= 1 \\ \lambda_2 &= 4 \end{aligned}$$

Bases for the eigenspaces of $A$ can be obtained by solving the linear system

$$\begin{bmatrix} \lambda - 2 & -1-i \\ -1+i & \lambda - 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

by substituting with $\lambda = 1$ and $\lambda = 4$ we will obtain the general solutions.

The general solutions of these systems are

$$\lambda_1 = 1: \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = t \begin{bmatrix} -1-i \\ 1 \end{bmatrix}$$

and

$$\lambda_2 = 4: \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = t \begin{bmatrix} \frac{1}{2}(1+i) \\ 1 \end{bmatrix}$$

Thus, bases for these eigenspaces are

$$\lambda_1 = 1: \quad \boldsymbol{v}_1 = \begin{bmatrix} -1-i \\ 1 \end{bmatrix}$$

and

$$\lambda_2 = 4: \quad \boldsymbol{v}_2 = \begin{bmatrix} \frac{1}{2}(1+i) \\ 1 \end{bmatrix}$$

The vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are orthogonal since

$$\begin{aligned} \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 &= (-1-i)\overline{\left(\frac{1}{2}(1+i)\right)} + (1)(\overline{1}) \\ &= \frac{1}{2}(-1-i)(1-i) + 1 \\ &= 0 \end{aligned}$$

and hence all scalar multiples of them are also orthogonal.

The C++ code is going to rely on **Armadillo** library to be able to compute the eigenvalues and eigenvectors of a complex matrix, the complex matrix $A$ is composed of two matrices, the first matrix is a Real part matrix we load it from a textfile named **matrixReal.txt**, and the second matrix is a Complex part matrix we load it from a textfile named **matrixComples.txt**.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat Real;
       mat Complex;
       Real.load("matrixReal.txt");
       Complex.load("matrixComplex.txt");
       cx_mat A(Real,Complex);
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix

       cout << "Eigenvalues:" << endl;

       int n = 2;
       for (int i = 1; i <= n; i++)
       {
              cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                  imag(eigval[i-1]) << "i" << endl;
       }

       cout << endl;
       cx_mat A1 = I*real(eigval[0]) - A;
       cx_mat A2 = I*real(eigval[1]) - A;

       cout << endl;
       cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
       cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
```

```
        cout << endl;

        cout << "Orthonormal Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
            cout << "x[l_" << i << "] :\n " <<eigvec.col(i-1) << endl;
        }

        cout << "det(A) = " << det(A) << endl;
        cout << "rank(A) = " << arma::rank(A) << endl;

        return 0;
    }
```

**C++ Code 226:** *main.cpp "Eigenvalues and eigenvectors of a hermitian matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A:
    (+2.000e+00,+0.000e+00)    (+1.000e+00,+1.000e+00)
    (+1.000e+00,-1.000e+00)    (+3.000e+00,+0.000e+00)

Eigenvalues:
λ_1 = 1 + 2.52703e-16i
λ_2 = 4 + 1.91387e-16i


Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Eigenvectors:
x[λ_1] :
     (+8.165e-01,+0.000e+00)
    (-4.082e-01,+4.082e-01)

x[λ_2] :
     (+4.082e-01,+4.082e-01)
    (+8.165e-01,+0.000e+00)

det(A) = (4,0)
rank(A) = 2
```

**Figure 23.209:** *The computation to obtain the eigenvalues and orthonormal eigenvectors of a Hermitian matrix. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Eigenvectors of a Hermitian Matrix with Armadillo/main.cpp**).*

## ii. C++ Computation: Inverse of A Unitary Matrix

Show that

$$A = \begin{bmatrix} \frac{1}{2}(1+i) & \frac{1}{2}(1+i) \\ \frac{1}{2}(1-i) & \frac{1}{2}(-1+i) \end{bmatrix}$$

is unitary, and then find $A^{-1}$.

**Solution:**

We will show that the row vectors

$$\boldsymbol{r}_1 = \begin{bmatrix} \frac{1}{2}(1+i) & \frac{1}{2}(1+i) \end{bmatrix}$$
$$\boldsymbol{r}_2 = \begin{bmatrix} \frac{1}{2}(1-i) & \frac{1}{2}(-1+i) \end{bmatrix}$$

are orthonormal. The relevant computations are

$$||\boldsymbol{r}_1|| = \sqrt{\left|\frac{1}{2}(1+i)\right|^2 + \left|\frac{1}{2}(1+i)\right|^2} = \sqrt{\frac{1}{2}+\frac{1}{2}} = 1$$
$$||\boldsymbol{r}_1|| = \sqrt{\left|\frac{1}{2}(1-i)\right|^2 + \left|\frac{1}{2}(-1+i)\right|^2} = \sqrt{\frac{1}{2}+\frac{1}{2}} = 1$$

$$\begin{aligned} \boldsymbol{r}_1 \cdot \boldsymbol{r}_2 &= \left(\frac{1}{2}(1+i)\right)\left(\overline{\frac{1}{2}(1-i)}\right) + \left(\frac{1}{2}(1+i)\right)\left(\overline{\frac{1}{2}(-1+i)}\right) \\ &= \left(\frac{1}{2}(1+i)\right)\left(\frac{1}{2}(1+i)\right) + \left(\frac{1}{2}(1+i)\right)\left(\frac{1}{2}(-1-i)\right) \\ &= \frac{1}{2}i - \frac{1}{2}i = 0 \end{aligned}$$

Since we know that $A$ is unitary, it follows that

$$A^{-1} = A^* = \begin{bmatrix} \frac{1}{2}(1-i) & \frac{1}{2}(1+i) \\ \frac{1}{2}(1-i) & \frac{1}{2}(-1-i) \end{bmatrix}$$

This equation is also holds

$$AA^* = A^*A = I$$

The C++ code is using **Armadillo** to compute the complex conjugate of a complex matrix, to find the inverse of a complex matrix $A$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
```

```
int main(int argc, char** argv)
{
        mat Real;
        mat Complex;
        Real.load("matrixReal.txt");
        Complex.load("matrixComplex.txt");
        cx_mat A(Real,Complex);
        cx_mat eigvec;
        cx_vec eigval;

        mat I(2,2,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
                cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                    imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        cx_mat A1 = I*real(eigval[0]) - A;
        cx_mat A2 = I*real(eigval[1]) - A;

        cout << endl;
        cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
        cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
        cout << endl;

        cout << "Orthonormal Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
                cout << "x[l_" << i << "] :\n " <<eigvec.col(i-1) << endl;
        }

        cout << "det(A) = " << det(A) << endl;
        cout << "rank(A) = " << arma::rank(A) << endl;

        cx_mat A_unitary = A.t() ;
        cout <<"Complex conjugate of A:" <<endl;
        cout <<"A*:" << "\n" << A_unitary<<endl;

        cout <<"A A*:" << "\n" << A*A_unitary <<endl;
```

```
        cout <<"r1:" << "\n" << A.row(0) <<endl;
        cout <<"r2:" << "\n" << A.row(1) <<endl;
        cout <<"||r1||:" << norm(A.row(0)) <<endl;
        cout <<"||r2||:" << norm(A.row(1)) <<endl;
        cout <<"r1 . r2 :" << dot(A.row(0),A.row(1).t()) <<endl;

        return 0;
    }
```

**C++ Code 227:** *main.cpp "Inverse of a unitary matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In **Armadillo** if we want to create a Hermitian / conjugate transpose of a matrix we can use **.t()** as a member function of any matrix or vector expression. This function will also flip the sign of imaginary components.

```
        ...

        cx_mat A_unitary = A.t() ;

        ...
```

```
Matrix A:
    (+5.000e-01,+5.000e-01)    (+5.000e-01,+5.000e-01)
    (+5.000e-01,-5.000e-01)    (-5.000e-01,+5.000e-01)

Eigenvalues:
λ_1 = 0.866025 + 0.5i
λ_2 = -0.866025 + 0.5i


Eigenspaces for λ_1: 0
Eigenspaces for λ_2: 0

Orthonormal Eigenvectors:
x[λ_1] :
     (+8.881e-01,+0.000e+00)
    (+3.251e-01,-3.251e-01)

x[λ_2] :
     (-3.251e-01,-3.251e-01)
    (+8.881e-01,+0.000e+00)

det(A) = (-1,0)
rank(A) = 2
Complex conjugate of A:
A*:
    (+5.000e-01,-5.000e-01)    (+5.000e-01,+5.000e-01)
    (+5.000e-01,-5.000e-01)    (-5.000e-01,-5.000e-01)

A A*:
    (+1.000e+00,+0.000e+00)                      (0,0)
                      (0,0)    (+1.000e+00,+0.000e+00)

r1:
    (+5.000e-01,+5.000e-01)    (+5.000e-01,+5.000e-01)

r2:
    (+5.000e-01,-5.000e-01)    (-5.000e-01,+5.000e-01)

||r1||:1
||r2||:1
r1 . r2 :(0,0)
```

**Figure 23.210:** *The computation to obtain norm of row vectors of matrix $A$, the complex Euclidean inner product between the row vectors and $A^{-1} = A^*$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Inverse of a Unitary Matrix with Armadillo/main.cpp**).*

## iii. C++ Computation: Unitary Diagonalization of a Hermitian Matrix

Find a matrix $P$ that unitarily diagonalizes the Hermitian matrix

$$A = \begin{bmatrix} 2 & 1+i \\ 1-i & 3 \end{bmatrix}$$

**Solution:**

The eigenvalues of $A$ are

$$\lambda_1 = 1$$
$$\lambda_2 = 4$$

The bases for the corresponding eigenspaces are

$$\lambda_1 = 1: \quad \boldsymbol{v}_1 = \begin{bmatrix} -1-i \\ 1 \end{bmatrix}$$

and

$$\lambda_2 = 4: \quad \boldsymbol{v}_2 = \begin{bmatrix} \frac{1}{2}(1+i) \\ 1 \end{bmatrix}$$

Since the eigenspace has only one basis vector, the Gram-Schmidt process is simply a matter of normalizing these basis vectors.

$$\boldsymbol{p}_1 = \frac{\boldsymbol{v}_1}{||\boldsymbol{v}_1||} = \begin{bmatrix} \frac{-1-i}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$$

$$\boldsymbol{p}_2 = \frac{\boldsymbol{v}_2}{||\boldsymbol{v}_2||} = \begin{bmatrix} \frac{1+i}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \end{bmatrix}$$

Thus, $A$ is unitarily diagonalized by the matrix

$$P = \begin{bmatrix} \boldsymbol{p}_1 & \boldsymbol{p}_2 \end{bmatrix} = \begin{bmatrix} \frac{-1-i}{\sqrt{3}} & \frac{1+i}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{6}} \end{bmatrix}$$

You may check the result by showing that

$$P^* A P = \begin{bmatrix} \frac{-1+i}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{1-i}{\sqrt{6}} & \frac{2}{\sqrt{6}} \end{bmatrix} = \begin{bmatrix} 2 & 1+i \\ 1-i & 3 \end{bmatrix} \begin{bmatrix} \frac{-1-i}{\sqrt{3}} & \frac{1+i}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{6}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
```

```
{
        mat Real;
        mat Complex;
        Real.load("matrixReal.txt");
        Complex.load("matrixComplex.txt");
        cx_mat A(Real,Complex);
        cx_mat eigvec;
        cx_vec eigval;

        mat I(2,2,fill::eye);
        cout <<"Matrix A:" << "\n" << A <<endl;

        eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
            square matrix

        cout << "Eigenvalues:" << endl;

        int n = 2;
        for (int i = 1; i <= n; i++)
        {
               cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                   imag(eigval[i-1]) << "i" << endl;
        }

        cout << endl;
        cx_mat A1 = I*real(eigval[0]) - A;
        cx_mat A2 = I*real(eigval[1]) - A;

        cout << endl;
        cout << "Eigenspaces for l_1: " << n - arma::rank(A1) << endl;
        cout << "Eigenspaces for l_2: " << n - arma::rank(A2) << endl;
        cout << endl;

        cout << "Orthonormal Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
               cout << "p[l_" << i << "] :\n " << eigvec.col(i-1) << endl;
        }

        //cout << "det(A) = " << det(A) << endl;
        //cout << "rank(A) = " << arma::rank(A) << endl;

        cout << "\np1 . p2 = " << dot(eigvec.col(0),eigvec.col(1).t()) <<
            endl;

        cx_mat P(3,3,fill::eye);
        cx_mat Pt(3,3,fill::eye);
        P = eigvec;
```

```
        Pt = P.t();

        cx_mat D(3,3,fill::eye);
        D = Pt*A*P;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (i !=j)
                {
                    D(i,j) = 0;
                }
            }
        }

        cout <<"\nMatrix P:" << "\n" << P <<endl;
        cout <<"Matrix P*:" << "\n" << Pt <<endl;
        cout <<"Matrix P*AP:" << "\n" << D <<endl;

        mat RealP;
        mat ComplexP;
        RealP.load("matrixRealP.txt");
        ComplexP.load("matrixComplexP.txt");
        cx_mat P2(RealP,ComplexP);
        cout <<"\nMatrix P2:" << "\n" << P2 <<endl;
        cout <<"Matrix P2*:" << "\n" << P2.t() <<endl;
        cout <<"Matrix P2*AP2:" << "\n" << (P2.t())*A*P2 <<endl;

        return 0;
    }
```

**C++ Code 228:** *main.cpp "Unitary diagonalization of a hermitian matrix"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We check the orthonormal vectors that are obtained from the book [11] and from the computation obtained by **Armadillo** is the first basis vector

$$p_1 = \begin{bmatrix} \frac{2}{\sqrt{6}} \\ \frac{-1+i}{\sqrt{6}} \end{bmatrix}$$

while the book [11] states the first basis vector as

$$p_1 = \begin{bmatrix} \frac{-1-i}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$$

both will resulting in the same diagonal matrix with diagonal entries are the eigenvalues.

```
...
...

cx_mat P(3,3,fill::eye);
cx_mat Pt(3,3,fill::eye);
P = eigvec;
Pt = P.t();

cx_mat D(3,3,fill::eye);
D = Pt*A*P;
for (int i = 0; i < n; i++)
{
       for (int j = 0; j < n; j++)
       {
              if (i !=j)
              {
                     D(i,j) = 0;
              }
       }
}

cout <<"\nMatrix P:" << "\n" << P <<endl;
cout <<"Matrix P*:" << "\n" << Pt <<endl;
cout <<"Matrix P*AP:" << "\n" << D <<endl;

mat RealP;
mat ComplexP;
RealP.load("matrixRealP.txt");
ComplexP.load("matrixComplexP.txt");
cx_mat P2(RealP,ComplexP);
cout <<"\nMatrix P2:" << "\n" << P2 <<endl;
cout <<"Matrix P2*:" << "\n" << P2.t() <<endl;
cout <<"Matrix P2*AP2:" << "\n" << (P2.t())*A*P2 <<endl;
```

- To check the orthogonality between $p_1$ and $p_2$ we can use complex Euclidean inner product

$$\begin{aligned} p_1 \cdot p_2 &= \left(\frac{-1-i}{\sqrt{3}}\right)\overline{\left(\frac{1+i}{\sqrt{6}}\right)} + \left(\frac{1}{\sqrt{3}}\right)\left(\overline{\frac{2}{\sqrt{6}}}\right) \\ &= \frac{-2}{\sqrt{18}} + \frac{2}{\sqrt{18}} \\ &= 0 \end{aligned}$$

if we use the basis vector $p_1$ obtained from **Armadillo**, we will still obtain the same result, that they are orthogonal

$$\begin{aligned} p_1 \cdot p_2 &= \left(\frac{2}{\sqrt{6}}\right)\overline{\left(\frac{1+i}{\sqrt{6}}\right)} + \left(\frac{-1+1}{\sqrt{6}}\right)\left(\overline{\frac{2}{\sqrt{6}}}\right) \\ &= \frac{2-2i}{\sqrt{6}} + \frac{2i-2}{\sqrt{6}} \\ &= 0 \end{aligned}$$

```
cout << "\np1 . p2 = " << dot(eigvec.col(0),eigvec.col(1).t())
    << endl;
```

We can say that for Hermitian matrix there are at least 2 orthonormal basis vectors. It could occurs since we are in the complex number dimension.

```
Matrix A:
    (+2.000e+00,+0.000e+00)    (+1.000e+00,+1.000e+00)
    (+1.000e+00,-1.000e+00)    (+3.000e+00,+0.000e+00)

Eigenvalues:
λ_1 = 1 + 2.52703e-16i
λ_2 = 4 + 1.91387e-16i


Eigenspaces for λ_1: 1
Eigenspaces for λ_2: 1

Orthonormal Eigenvectors:
p[λ_1] :
     (+8.165e-01,+0.000e+00)
    (-4.082e-01,+4.082e-01)

p[λ_2] :
     (+4.082e-01,+4.082e-01)
    (+8.165e-01,+0.000e+00)


p1 . p2 = (4.44089e-16,0.666667)

Matrix P:
    (+8.165e-01,+0.000e+00)    (+4.082e-01,+4.082e-01)
    (-4.082e-01,+4.082e-01)    (+8.165e-01,+0.000e+00)

Matrix P*:
    (+8.165e-01,-0.000e+00)    (-4.082e-01,-4.082e-01)
    (+4.082e-01,-4.082e-01)    (+8.165e-01,-0.000e+00)

Matrix P*AP:
    (+1.000e+00,+2.776e-17)                     (0,0)
                     (0,0)    (+4.000e+00,+0.000e+00)


Matrix P2:
    (-5.774e-01,-5.774e-01)    (+4.082e-01,+4.082e-01)
    (+5.774e-01,+0.000e+00)    (+8.165e-01,+0.000e+00)

Matrix P2*:
    (-5.774e-01,+5.774e-01)    (+5.774e-01,-0.000e+00)
    (+4.082e-01,-4.082e-01)    (+8.165e-01,-0.000e+00)

Matrix P2*AP2:
    (+1.000e+00,+0.000e+00)    (+5.543e-05,+0.000e+00)
    (+5.543e-05,+0.000e+00)    (+4.000e+00,+0.000e+00)
```

**Figure 23.211:** *The computation to find the matrix P that unitarily diagonalizes the hermitian matrix A. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Unitary Diagonalization of a Hermitian Matrix with Armadillo/main.cpp**).*

# XV. Linear Transformations

[DF*] A matrix transformation $T_A : \mathbb{R}^n \to \mathbb{R}^m$ is defined as a mapping of the form

$$T_A(\boldsymbol{x}) = A\boldsymbol{x}$$

in which $A$ is an $m \times n$ matrix.

**Definition 23.62: Linear Transformation**

If $T : V \to W$ is a function from a vector space $V$ to a vector space $W$, then $T$ is called a linear transformation from $V$ to $W$ if the following two properties hold for all vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $V$ and for all scalars $k$:

(a) $T(k\boldsymbol{u}) = kT(\boldsymbol{u})$ (Homogeneity property)

(b) $T(\boldsymbol{u} + \boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$ (Additivity property)

In the special case where $V = W$, the linear transformation $T$ is called a linear operator on the vector space $V$.

[DF*] The homogeneity and additivity properties of a linear transformation $T : V \to W$ can be used in combination to show that if $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are vectors in $V$ and $k_1$ and $k_2$ are any scalars, then

$$T(k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2) = k_1T(\boldsymbol{v}_1) + k_2T(\boldsymbol{v}_2)$$

More generally, if $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r$ are vectors in $V$ and $k_1, k_2, \ldots, k_r$ are any scalars, then

$$T(k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_r\boldsymbol{v}_r) = k_1T(\boldsymbol{v}_1) + k_2T(\boldsymbol{v}_2) + \cdots + k_rT(\boldsymbol{v}_r) \tag{23.233}$$

**Theorem 23.137: Linear Transformation**

If $T : V \to W$ is a linear transformation, then:

(a) $T(\boldsymbol{0}) = \boldsymbol{0}$

(b) $T(\boldsymbol{u} - \boldsymbol{v}) = T(\boldsymbol{u}) - T(\boldsymbol{v})$ for all $\boldsymbol{u}$ and $\boldsymbol{v}$ in $V$

[DF*] **Matrix Transformations**
We have based the definition of a general linear transformation on the homogeneity and additivity properties of matrix transformations, it follows that a matrix transformation $T_A : \mathbb{R}^n \to \mathbb{R}^m$ is also a linear transformation in this more general sense with $V = \mathbb{R}^n$ and $W = \mathbb{R}^m$.

[DF*] **The Zero Transformation**
Let $V$ and $W$ be any two vector spaces. The mapping $T : V \to W$ such that $T(\boldsymbol{v}) = \boldsymbol{0}$ for every $\boldsymbol{v}$ in $V$ is a linear transformation called the zero transformation. To see that $T$ is linear, observe that

$$\begin{aligned} T(\boldsymbol{u} + \boldsymbol{v}) &= \boldsymbol{0} \\ T(\boldsymbol{u}) &= \boldsymbol{0} \\ T(\boldsymbol{v}) &= \boldsymbol{0} \\ T(k\boldsymbol{u}) &= \boldsymbol{0} \end{aligned}$$

Therefore,

$$T(\boldsymbol{u}+\boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$$
$$T(k\boldsymbol{u}) = kT(\boldsymbol{u})$$

[DF*] **The Identity Operator**
Let $V$ be any vector space. The mapping $I : V \to V$ defined by $I(\boldsymbol{v}) = \boldsymbol{v}$ is called the identity operator on $V$. $I$ is linear.

[DF*] **Dilation and Contraction Operators**
If $V$ is a vector space and $k$ is any scalar, then the mapping $T : V \to V$ given by $T(\boldsymbol{x}) = k\boldsymbol{x}$ is a linear operator on $V$, for if $c$ is any scalar and if $\boldsymbol{u}$ and $\boldsymbol{v}$ are any vectors in $V$, then

$$T(c\boldsymbol{u}) = k(c\boldsymbol{u}) = c(k\boldsymbol{u}) = cT(\boldsymbol{u})$$
$$T(\boldsymbol{u}+\boldsymbol{v}) = k(\boldsymbol{u}+\boldsymbol{v}) = k\boldsymbol{u}+k\boldsymbol{v} = T(\boldsymbol{u})+T(\boldsymbol{v})$$

If $0 < k < 1$, then $T$ is called the contraction of $V$ with factor $k$, and if $k > 1$, it is called the dilation of $V$ with factor $k$.

[DF*] **A Linear Transformation from $P_n$ to $P_{n+1}$**
Let $\boldsymbol{p} = p(x) = c_0 + c_1x + \cdots + c_nx^n$ be a polynomial in $P_n$, and define the transformation $T : P_n \to P_{n+1}$ by

$$T(\boldsymbol{p}) = T(p(x)) = xp(x) = c_0x + c_1x^2 + \cdots + c_nx^{n+1}$$

This transformation is linear because for any scalar $k$ and any polynomials $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ in $P_n$ we have

$$T(k\boldsymbol{p}) = T(kp(x)) = x(kp(x)) = k(xp(x)) = kT(\boldsymbol{p}))$$

and

$$\begin{aligned} T(\boldsymbol{p}_1+\boldsymbol{p}_2) &= T(p_1(x)+p_2(x)) \\ &= x(p_1(x)+p_2(x)) \\ &= xp_1(x)+xp_2(x) \\ &= T(\boldsymbol{p}_1)+T(\boldsymbol{p}_2) \end{aligned}$$

[DF*] **A Linear Transformation Using an Inner Product**
Let $V$ be an inner product space, let $\boldsymbol{v}_0$ be any fixed vector in $V$, and let $T : V \to R$ be the transformation

$$T(\boldsymbol{x}) =< \boldsymbol{x}, \boldsymbol{v}_0 >$$

that maps a vector $\boldsymbol{x}$ into its inner product with $\boldsymbol{v}_0$. This transformation is linear, for if $k$ is any scalar, and if $\boldsymbol{u}$ and $\boldsymbol{v}$ are any vectors in $V$, then it follows from properties of inner products that

$$T(k\boldsymbol{u}) =< k\boldsymbol{u}, \boldsymbol{v}_0 >= k < \boldsymbol{u}, \boldsymbol{v}_0 >= kT(\boldsymbol{u})$$
$$T(\boldsymbol{u}+\boldsymbol{v}) =< \boldsymbol{u}+\boldsymbol{v}, \boldsymbol{v}_0 >=< \boldsymbol{u}, \boldsymbol{v}_0 > + < \boldsymbol{v}, \boldsymbol{v}_0 >= T(\boldsymbol{u})+T(\boldsymbol{v})$$

[DF*] **Example: Transformations on Matrix Spaces**
Let $M_{nn}$ be the vector space of $n \times n$ matrices. In each part determine whether the transformation is linear.

(a) $T_1(A) = A^T$

(b) $T_2(A) = \det(A)$

**Solution:**

(a) It follows that

$$T_1(kA) = (kA)^T = kA^T = kT_1A$$
$$T_1(A+B) = (A+B)^T = A^T + B^T = T_1(A) + T_1(B)$$

so $T_1$ is linear.

(b) It follows that

$$T_2(kA) = \det(kA) = k^n \det(A) = k^n T_2(A)$$

Thus, $T_2$ is not homogeneous and hence not linear if $n > 1$. Note that additivity also fails because we showed that $\det(A+B)$ and $\det(A) + \det(B)$ are not generally equal.

[DF*] **Translation Is Not Linear**

A Theorem states that a linear transformation maps $\mathbf{0}$ to $\mathbf{0}$. This property is useful for identifying transformations that are not linear. For example, if $\boldsymbol{x}_0$ is a fixed nonzero vector in $\mathbb{R}^2$, then the transformation

$$T(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{x}_0$$

has the geometric effect of translating each point $\boldsymbol{x}$ in a direction parallel to $\boldsymbol{x}_0$ through a distance of $||\boldsymbol{x}_0||$. This cannot be a linear transformation since $T(\mathbf{0}) = \boldsymbol{x}_0$, so $T$ does not map $\mathbf{0}$ to $\mathbf{0}$.

**Figure 23.212:** *$T(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{x}_0$ translates each point $\boldsymbol{x}$ along a line parallel to $\boldsymbol{x}_0$ through a distance $||\boldsymbol{x}_0||$.*

[DF*] **The Evaluation Transformation**

Let $V$ be a subspace of $F(-\infty, \infty)$, let

$$x_1, x_2, \ldots, x_n$$

be distinct real numbers, and let $T : V \to \mathbb{R}^n$ be the transformation

$$T(f) = (f(x_1), f(x_2), \ldots, f(x_n)) \tag{23.234}$$

that associates with $f$ the $n$-tuple of function values at $x_1, x_2, \ldots, x_n$. We call this the evaluation transformation on $V$ at $x_1, x_2, \ldots, x_n$. Thus, for example, if

$$x_1 = -1, \quad x_2 = 2, \quad x_3 = 4$$

and if $f(x) = x^2 - 1$, then

$$T(f) = (f(x_1), f(x_2), f(x_3)) = (0, 3, 15)$$

The evaluation transformation is linear, for if $k$ is any scalar, and if $f$ and $g$ are any functions in $V$, then

$$\begin{aligned} T(kf) &= ((kf)(x_1), (kf)(x_2), \ldots, (kf)(x_n)) \\ &= (kf(x_1), kf(x_2), \ldots, kf(x_n)) \\ &= k(f(x_1), f(x_2), \ldots, f(x_n)) \\ &= kT(f) \end{aligned}$$

and

$$\begin{aligned} T(f+g) &= ((f+g)(x_1), (f+g)(x_2), \ldots, (f+g)(x_n)) \\ &= (f(x_1) + g(x_1), f(x_2) + g(x_2), \ldots, f(x_n) + g(x_n)) \\ &= (f(x_1), f(x_2), \ldots, f(x_n)) + (g(x_1), g(x_2), \ldots, g(x_n)) \\ &= T(f) + T(g) \end{aligned}$$

[DF*] If $\mathbb{R}^n \to \mathbb{R}^n$ is a matrix transformation, say multiplication by $A$, and if $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$ are the standard basis vectors for $\mathbb{R}^n$, then $A$ can be expressed as

$$A = [T(\boldsymbol{e}_1) \mid T(\boldsymbol{e}_2) \mid \ldots \mid T(\boldsymbol{e}_n)]$$

It follows from this that the image of any vector $\boldsymbol{v} = (c_1, c_2, \ldots, c_n)$ in $\mathbb{R}^n$ under multiplication by $A$ can be expressed as

$$T(\boldsymbol{v}) = c_1 T(\boldsymbol{e}_1) + c_2 T(\boldsymbol{e}_2) + \cdots + c_n T(\boldsymbol{e}_n)$$

This formula tells us that for a matrix transformation the image of any vector is expressible as a linear combination of the images of the standard basis vectors.

**Theorem 23.138: Linear Transformations from Images of Basis Vectors**

Let $T : V \to W$ be a linear transformation, where $V$ is finite dimensional. If $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a basis for $V$, then the image of any vector $\boldsymbol{v}$ in $V$ can be expressed as

$$T(\boldsymbol{v}) = c_1 T(\boldsymbol{v}_1) + c_2 T(\boldsymbol{v}_2) + \cdots + c_n T(\boldsymbol{v}_n) \tag{23.235}$$

where $c_1, c_2, \ldots, c_n$ are the coefficients required to express $\boldsymbol{v}$ as a linear combination of the vectors in $S$.

[DF*] **A Linear Transformation from $C^1(-\infty, \infty)$ to $F(-\infty, \infty)$**
Let $V = C^1(-\infty, \infty)$ be the vector space of functions with continuous first derivatives on $(-\infty, \infty)$, and let $W = F(-\infty, \infty)$ be the vector space of all real-valued functions defined

on $(-\infty, \infty)$. Let $D : V \to W$ be the transformation that maps a function $\boldsymbol{f} = f(x)$ into its derivative, that is

$$D(\boldsymbol{f}) = f'(x)$$

From the properties of differentiation, we have

$$D(\boldsymbol{f} + \boldsymbol{g}) = D(k\boldsymbol{f}) = kD(\boldsymbol{f})$$

and

$$D(\boldsymbol{f} + \boldsymbol{g}) = D(\boldsymbol{f}) + D(\boldsymbol{g})$$

Thus, $D$ is a linear transformation.

[DF*] **An Integral Transformation**

Let $V = C(-\infty, \infty)$ be the vector space of continuous functions on the interval $(-\infty, \infty)$, let $W = C^1(-\infty, \infty)$ be the vector space of functions with continuous first derivatives on $(-\infty, \infty)$, and let $J : V \to W$ be the transformation that maps a function $f$ in $V$ into

$$J(f) = \int_0^x f(t)\, dt$$

For example, if $f(x) = x^2$, then

$$J(f) = \int_0^x t^2\, dt = \frac{t^3}{3}|_0^x = \frac{x^3}{3}$$

The transformation $J : V \to W$ is linear, for if $k$ is any constant, and if $f$ and $g$ are any functions in $V$, then properties of the integral imply that

$$\begin{aligned} J(kf) &= \int_0^x kf(t)\, dt \\ &= k\int_0^x f(t)\, dt \\ &= kJ(f) \end{aligned}$$

$$\begin{aligned} J(f+g) &= \int_0^x (f(t) + g(t))\, dt \\ &= \int_0^x f(t)\, dt + \int_0^x g(t)\, dt \\ &= J(f) + J(g) \end{aligned}$$

[DF*] If $A$ is an $m \times n$ matrix, then the null space of $A$ consists of all vectors $\boldsymbol{x}$ in $\mathbb{R}^n$ such that $A\boldsymbol{x} = \boldsymbol{0}$, and the column space of $A$ consists of all vectors $\boldsymbol{b}$ in $\mathbb{R}^m$ for which there is at least one vector $\boldsymbol{x}$ in $\mathbb{R}^n$ such that $A\boldsymbol{x} = \boldsymbol{b}$.

From the viewpoint of matrix transformations, the null space of $A$ consists of all vectors in $\mathbb{R}^n$ that multiplication by $A$ maps into $\boldsymbol{0}$, and the column space of $A$ consists of all vectors in $\mathbb{R}^m$ that are images of at least one vector in $\mathbb{R}^n$ under multiplication by $A$.

**Definition 23.63: Kernel and Range**

If $T : V \to W$ is a linear transformation, then the set of vectors in $V$ that maps $T$ into $\boldsymbol{0}$ is called the kernel of $T$, and is denoted by $\ker(T)$.

The set of all vectors in $W$ that are images under $T$ of at least one vector in $V$ is called the range of $T$ and is denoted by $R(T)$.

[DF*] **Kernel and Range of a Matrix Transformation**
If $T_A : \mathbb{R}^n \to \mathbb{R}^m$ is multiplication by the $m \times n$ matrix $A$, then, the kernel of $T_A$ is the null space of $A$, and the range of $T_A$ is the column space of $A$.

[DF*] **Kernel and Range of the Zero Transformation**
Let $T : V \to W$ be the zero transformation. Since $T$ maps every vector in $V$ into $\mathbf{0}$, it follows that $\ker(T) = V$. Moreover, since $\mathbf{0}$ is the only image under $T$ of vectors in $V$, it folows that $R(T) = \{\mathbf{0}\}$

[DF*] **Kernel and Range of the Identity Transformation**
Let $I : V \to V$ be the identity operator. Since $I(\boldsymbol{v}) = \boldsymbol{v}$ for all vectors in $V$, every vector in $V$ is the image of some vector (itself); thus $R(I) = V$. Since the only vector that $I$ maps into $\mathbf{0}$ is $\mathbf{0}$, it follows that $\ker(I) = \{\mathbf{0}\}$.

[DF*] **Kernel and Range of an Orthogonal Projection**
The points that $T$ maps into $\mathbf{0} = (0, 0, 0)$ are precisely those on the $z$-axis, so $\ker(T)$ is the set of points of the form $(0, 0, z)$.

$T$ maps the points in $\mathbb{R}^3$ to the $xy$-plane, where each point in that plane is the image of each point on the vertical line above it. Thus $R(T)$ is the set of points in the form $(x, y, 0)$.

[DF*] **Kernel and Range of a Rotation**
Let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be the linear operator that rotates each vector in the $xy$-plane through the angle $\theta$, it follows that $R(T) = R^2$. Moreover, the only vector that rotates into $\mathbf{0}$ is $\mathbf{0}$, so $\ker(T) = \{\mathbf{0}\}$.

[DF*] **Kernel of a Differentiation Transformation**
Let $V = C^1(-\infty, \infty)$ be the vector space of functions with continuous first derivatives on $(-\infty, \infty)$, let $W = F(-\infty, \infty)$ be the vector space of all real-valued functions defined on $(-\infty, \infty)$, and let $D : V \to W$ be the differentiation transformation $D(\boldsymbol{f}) = f'(x)$. The kernel of $D$ is the set of functions in $V$ with derivative zero. From calculus, this is the set of constant functions on $(-\infty, \infty)$.

[DF*] Most of the time, $\ker(T)$ and $R(T)$ turned out to be subspaces. They were either zero subspace or the entire vector space. In another case, the kernal was a line through the origin, and the range was a plane through the origin, both of which are subspaces of $\mathbb{R}^3$.

**Theorem 23.139: Kernel and Range of a Linear Transformation**

If $T : V \to W$ is a linear transformation, then:

(a) The kernel of $T$ is a subspace of $V$.

(b) The range of $T$ is a subspace of $W$.

[DF*] To show that $\ker(T)$ is a subspace, we must show that it contains at least one vector and is closed under addition and scalar multiplication.

[DF*] To show that $R(T)$ is a subspace, we must show that it contains at least one vector and is closed under addition and scalar multiplication.

[DF*] **Application to Differential Equations**
Differential equations of the form

$$y'' + \omega^2 y = 0 \tag{23.236}$$

with $\omega$ as a positive constant. The equation above arise in the study of vibrations. The set of all solutions of this equation on the interval $(-\infty, \infty)$ is the kernel of the linear

transformation $D : C^2(-\infty, \infty) \to C(-\infty, \infty)$, given by

$$D(y) = y'' + \omega^2 y$$

It is proved in standard textbooks on differential equations that the kernel is a two-dimensional subspace of $C^2(-\infty, \infty)$, so that if we can find two linearly independent solutions of the differential equation $D(y) = 0$, then all other solutions can be expressed as linear combinations of those two. We will find the general solution of

$$y'' + \omega^2 y = 0$$

the characteristic equation is $r^2 + \omega^2 = 0$ with the roots

$$r = \pm i\omega$$

the roots are conjugate complex numbers, hence the general solution is

$$y = c_1 e^{r_1 x} + c_2 e^{r_2 x}$$

and

$$\begin{aligned} y_1(x) &= e^{i\omega x} \\ y_2(x) &= e^{-i\omega x} \end{aligned}$$

the functions $y_1(x)$ and $y_2(x)$ are solutions of the differential equation. Based on the theorem, if $y_1$ and $y_2$ are solutions, then any linear combinations of $y_1$ and $y_2$ are also a solution.

$$\begin{aligned} y_1(x) + y_2(x) &= e^{i\omega x} + e^{-i\omega x} \\ &= (\cos\ \omega x + i \sin\ \omega x) + (\cos\ \omega x - i \sin\ \omega x) \\ &= 2 \cos\ \omega x \end{aligned} \tag{23.237}$$

and

$$\begin{aligned} y_1(x) - y_2(x) &= e^{i\omega x} - e^{-i\omega x} \\ &= (\cos\ \omega x + i \sin\ \omega x) - (\cos\ \omega x - i \sin\ \omega x) \\ &= 2i \sin\ \omega x \end{aligned} \tag{23.238}$$

neglecting the constant multipliers 2 and $2i$, respectively, we have obtained a pair of real-valued solutions

$$\begin{aligned} y_1(x) &= \cos\ \omega x \\ y_2(x) &= \sin\ \omega x \end{aligned}$$

they are solutions of the differential equations $D(y) = 0$. These functions are linearly independent since neither is a scalar multiple of the other, and thus

$$y = c_1 \cos\ \omega x + c_2 \sin\ \omega x \tag{23.239}$$

is a "general solution" of $D(y) = 0$ in the sense that every choice of $c_1$ and $c_2$ produces a solution, and every solution is of this form.

**Definition 23.64: Rank and Nullity of Linear Transformations**

Let $T : V \to W$ be a linear transformation. If the range of $T$ is finite-dimensional, then its dimension is called the rank of $T$; and if the kernel of $T$ is finite-dimensional, then its dimension is called the nullity of $T$. The rank of $T$ is denoted by $\text{rank}(T)$ and the nullity of $T$ by $\text{nullity}(T)$.

**Theorem 23.140: Dimension Theorem for Linear Transformations**

If $T : V \to W$ is a linear transformation from an $n$-dimensional vector space $V$ to a vector sace $W$, then

$$\text{rank}(T) + \text{nullity}(T) = n \tag{23.240}$$

[DF*] In the special case where $A$ is an $m \times n$ matrix and $T_A : R^n \to R^m$ is multiplication by $A$, the kernel of $T_A$ is the null space of $A$, and the range of $T_A$ is the column space of $A$. Thus, it follows that

$$\text{rank}(T_A) + \text{nullity}(T_A) = n$$

[DF*] **Example:**
Let $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$ be vectors in a vector space $V$, and let $T : V \to \mathbb{R}^3$ be a linear transformation for which

$$\begin{aligned} T(\boldsymbol{v}_1) &= (2, -1, 4) \\ T(\boldsymbol{v}_2) &= (-3, 2, 1) \\ T(\boldsymbol{v}_3) &= (0, 5, 1) \end{aligned}$$

Find $T(3\boldsymbol{v}_1 - 2\boldsymbol{v}_2 + \boldsymbol{v}_3)$.
**Solution:**
The computation is quite simple

$$\begin{aligned} T(3\boldsymbol{v}_1 - 2\boldsymbol{v}_2 + \boldsymbol{v}_3) &= 3T(\boldsymbol{v}_1) - 2T(\boldsymbol{v}_2) + T(\boldsymbol{v}_3) \\ &= 3(2, -1, 4) - 2(-3, 2, 1) + (0, 5, 1) \\ &= (6, -3, 12) - (-6, 4, 2) + (0, 5, 1) \\ &= (12, -2, 11) \end{aligned}$$

[DF*] **Example:**
Let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be the linear operator given by the formula

$$T(x, y) = (x - 3y, -2x + 6y)$$

(a) Which of the following vectors are in $R(T) : (1, -2), (3, 1), (-2, 4)$?

(b) Which of the following vectors are in $\ker(T) : (1, -3), (3, 1), (-6, -2)$?

**Solution:**

(a) We can this method

$$\begin{aligned} x - 3y &= b_1 \\ -2x + 6y &= b_2 \end{aligned}$$

with $(b_1, b_2)$ is the vector we want to test whether it is in $R(T)$ or not.

$$\begin{aligned} x - 3y &= 1 \\ -2x + 6y &= -2 \end{aligned}$$

the equations are linear combination of each other, thus $(1, -2)$ is in $R(T)$.

$$\begin{aligned} x - 3y &= 3 \\ -2x + 6y &= 1 \end{aligned}$$

the equations are not linear combination of each other, thus $(3, 1)$ is not in $R(T)$.

$$\begin{aligned} x - 3y &= -2 \\ -2x + 6y &= 4 \end{aligned}$$

the equations are linear combination of each other, thus $(-2, 4)$ is in $R(T)$.

(b) We will compute one by one

$$T(1, -3) = (1 - 3(-3), -2(1) + 6(-3)) = (10, -20)$$

$(1, -3)$ is not in $\ker(T)$.

$$T(3, 1) = (3 - 3(1), -2(3) + 6(1)) = (0, 0)$$

$(3, 1)$ is in $\ker(T)$

$$T(-6, -2) = (-6 - 3(-2), -2(-6) + 6(-2)) = (0, 0)$$

$(-6, -2)$ is in $\ker(T)$

[DF*] **Example:**
Let $T : P_2 \to P_3$ be the linear transformation defined by

$$T(p(x)) = xp(x)$$

(a) Which of the following are in $\ker(T)$: $x^2, 0, 1 + x$?

(b) Which of the following are in $R(T)$: $x + x^2, 1 + x, 3 - x^2$?

**Solution:**

(a) We will compute one by one

$$T(x^2) = x(x^2) = x^3$$

$x^2$ is not in $\ker(T)$.

$$T(0) = x(0) = 0$$

0 is in $\ker(T)$.

$$T(1 + x) = x(1 + x) = x + x^2$$

$1 + x$ is not in $\ker(T)$. The only vector that will map $T(p(x))$ into $\mathbf{0}$ is 0.

(b) Since $x + x^2 = x(1 + x)$, $x + x^2$ is in $R(T)$.

Neither $1 + x$ nor $3 - x^2$ can be expressed as $xp(x)$ with $p(x)$ in $P_2$, so neither are in $R(T)$.

[DF*] **Example:**
Find a basis for the kernel of

(a) the linear operator $T : \mathbb{R}^2 \to \mathbb{R}^2$ given by the formula

$$T(x, y) = (x - 3y, -2x + 6y)$$

(b) the linear transformation $T : P_2 \to P_3$ defined by

$$T(p(x)) = xp(x)$$

**Solution:**

(a) The kernel of $T(x, y)$ is the set of all vectors $(3x, x)$ or the line $y = 3x$.

The vector $(3, 1)$ is a basis for this space. If you input the basis or multiplication of the basis in the $T(x, y) = (x - 3y, -2x + 6y)$ we will obtain $\mathbf{0} = (0, 0)$.

(b) Since we need to obtain $T(x) = 0$, then

$$T(0) = x(0) = 0$$

the basis for the kernel is $\{0, 0, 0\}$ from standard basis of $P_2 : \{1, x, x^2\}$.

[DF*] **Example:**
Find a basis for the range of

(a) the linear operator $T : \mathbb{R}^2 \to \mathbb{R}^2$ given by the formula

$$T(x, y) = (x - 3y, -2x + 6y)$$

(b) the linear transformation $T : P_2 \to P_3$ defined by

$$T(p(x)) = xp(x)$$

(c) the matrix operator $T_A : \mathbb{R}^4 \to \mathbb{R}^3$ is multiplication by the $3 \times 4$ matrix $A$

$$A = \begin{bmatrix} 4 & 1 & -2 & -3 \\ 2 & 1 & 1 & -4 \\ 6 & 0 & -9 & 9 \end{bmatrix}$$

**Solution:**

(a) Since $x - 3y$ and $-2x + 6y$ can assume any real value, $R(T)$ is the set of all vectors $(x, -2x)$ or the line $y = -2x$. The vector $(1, -2)$ is a basis for this space.

(b) $R(T)$ is the set of all polynomials in $P^3$ with constant term 0. Thus the basis for the range / $R(T)$ is $\{x, x^2, x^3\}$.

(c) A basis for $R(T)$ is a basis for the column space of $A$.
First we will transform $A$ into its row echelon form

$$A_{ref} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{2} & -2 \\ 0 & 1 & 4 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The columns of $A$ corresponding to the columns of $A_{ref}$ containing leading $1s$ are

$$\begin{bmatrix} 4 \\ 2 \\ 6 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} -3 \\ -4 \\ 9 \end{bmatrix}$$

Thus, $(4, 2, 6)$, $(1, 1, 0)$, and $(-3, -4, 9)$ form a basis for $R(T)$.

[DF*] **Example:**
Let $T$ be multiplicaiton by the matrix $A$.

$$\begin{bmatrix} 1 & -1 & 3 \\ 5 & 6 & -4 \\ 7 & 4 & 2 \end{bmatrix}$$

Find

(a) a basis for the range of $T$.
(b) a basis for the kernel of $T$.
(c) the rank and nullity of $T$.
(d) the rank and nullity of $A$.

**Solution:**

(a) A row echelon form of $A$ is

$$A_{ref} = \begin{bmatrix} 1 & -1 & 3 \\ 0 & 1 & \frac{19}{11} \\ 0 & 0 & 0 \end{bmatrix}$$

thus columns 1 and 2 of $A$ is

$$\begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix}, \quad \begin{bmatrix} -1 \\ 6 \\ 4 \end{bmatrix}$$

form a basis for the column space of $A$, which is the range of $T$.

(b) The reduced row echelon form of $A$ is

$$A_{rref} = \begin{bmatrix} 1 & 0 & \frac{14}{11} \\ 0 & 1 & -\frac{19}{11} \\ 0 & 0 & 0 \end{bmatrix}$$

so a general solution of $A\boldsymbol{x} = \mathbf{0}$ is

$$x_1 = -\frac{14}{11}s$$
$$x_2 = \frac{19}{11}s$$
$$x_3 = s$$

so

$$\begin{bmatrix} -14 \\ 19 \\ 11 \end{bmatrix}$$

is a basis for the null space of $A$, which is $\ker(T)$.

(c) $R(T)$ is two-dimensinoal, so $\text{rank}(T) = 2$.

$\ker(T)$ is one-dimensional, so $\text{nullity}(T) = 1$.

(d) The column space of $A$ is two-dimensional, so $\text{rank}(A) = 2$. The null space of $A$ is one-dimensional, so $\text{nullity}(A) = 1$.

[DF*] **Example:**
Describe the kernel and range of

(a) the orthogonal projection on the $xz$-plane.

(b) the orthogonal projection on the $yz$-plane.

(c) the orthogonal projection on the plane defined by the equation $y = x$.

**Solution:**

(a) The kernel is the $y$-axis, the range is the entire $xy$-plane.

(b) The kernel is the $x$-axis, the range is the entire $yz$-plane.

(c) The kernel is the line through the origin perpendicular to the plane $y = x$, the range is the entire plane $y = x$.

[DF*] **Example:**
In each part, use the given information to find the nullity of the linear transformation $T$.

(a) $T : \mathbb{R}^5 \to \mathbb{R}^7$ has rank 3.

(b) $T : P_4 \to P_3$ has rank 1.

(c) The range of $T : \mathbb{R}^6 \to \mathbb{R}^3$ is $\mathbb{R}^3$.

(d) $T : M_{22} \to M_{22}$ has rank 3.

**Solution:**

(a) $\text{nullity}(T) = 5 - \text{rank}(T) = 2$

(b) $\dim(P_4) = 5$, so $\text{nullity}(T) = 5 - \text{rank}(T) = 4$

(c) Since $R(T) = R^3$, $T$ has rank 3.

$$\text{nullity}(T) = 6 - \text{rank}(T) = 3$$

(d) $\text{nullity}(T) = 4 - \text{rank}(T) = 1$

[DF*] **Example:**
Let $A$ be a $5 \times 7$ matrix with rank 4.

(a) What is the dimension of the solution space of $A\boldsymbol{x} = \mathbf{0}$ ?

(b) Is $A\boldsymbol{x} = \boldsymbol{b}$ consistent for all vectors $\boldsymbol{b}$ in $\mathbb{R}^5$? Explain.

**Solution:**

(a) Remember that the dimension of matrix $m \times n$ is $n$ (the number of column), thus

$$\dim(A) = \text{nullity}(A) + \text{rank}(A)$$

$\text{nullity}(A) = 7 - \text{rank}(A) = 3$, so the solution space of $A\boldsymbol{x} = \mathbf{0}$ has dimension 3.

(b) No, since $\text{rank}(A) = 4$ while $\mathbb{R}^5$ has 5 dimension. To be consistent it has to be in $\mathbb{R}^4$.

[DF*] **Example:**
Let $T : V \to \mathbb{R}^3$ be a linear transformation from any vector space to $\mathbb{R}^3$. Give a geometric description of $R(T)$.

**Solution:**
$R(T)$ must be a subspace of $\mathbb{R}^3$, thus the possibilities are a line through the origin, a plane through the origin, the origin only, or all of $\mathbb{R}^3$. It is the definition of subspaces of $\mathbb{R}^3$ from the General Vector Spaces chapter.

[DF*] **Example:**
Let $T : \mathbb{R}^3 \to \mathbb{R}^3$ be multiplication by

$$A = \begin{bmatrix} 1 & 3 & 4 \\ 3 & 4 & 7 \\ -2 & 2 & 0 \end{bmatrix}$$

(a) Show that the kernel of $T$ is a line through the origin, and find parametric equations for it.

(b) Show that the range of $T$ is a plane through the origin, and find an equation for it.

**Solution:**

(a) A reduced row echelon form of matrix $A$ is

$$A_{rref} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

A general solution of $A\boldsymbol{x} = \mathbf{0}$ is

$$\begin{aligned} x_1 - x_2 &= 0 \\ x_2 + x_3 &= 0 \end{aligned}$$

$$\begin{aligned} x_1 &= -t \\ x_2 &= -t \\ x_3 &= t \end{aligned}$$

so

$$\begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

is a basis for the null space of $A$, the kernel of $T$. Vector $(-1,-1,1)$ is a line through the origin, the parametric equation is

$$(x,y,z) = (-t-t,t) = (-1,-1,1)t$$

with variable $t$ as the parameter.

(b) From the reduced row echelon form of matrix $A$, the columns of $A$ corresponds to the columns of $A_{rref}$ containing leading $1s$ are

$$\begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}, \quad \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

thus, $(1,3,-2)$ and $(3,4,2)$ form a basis for $R(T)$ (both vectors span $\mathbb{R}^3$ and not a linear combination to the other). The geometric description of this range of $T$ is a plane through the origin with parametric equations

$$(x,y,z) = (s,3s,-2s) + (3t,4t,2t) = (1,3,-2)s + (3,4,2)t$$

with $s,t$ as the parameters variables.

[DF*] **Example:**

(a) Show that if $a_1, a_2, b_1$, and $b_2$ are any scalars, then the formula

$$F(x,y) = (a_1x + b_1y, a_2x + b_2y)$$

defines a linear operator on $\mathbb{R}^2$.

(b) Does the formula $F(x,y) = (a_1x^2 + b_1y^2, a_2x^2 + b_2y^2)$ define a linear operator on $\mathbb{R}^2$? Explain.

**Solution:**

(a) We will need to show that $F(x,y)$ follows two properties which are homogeneity and additivity. First, for the homogeneity property

$$\begin{aligned} F(kx,ky) &= (a_1kx + b_1ky, a_2kx + b_2ky) \\ &= k(a_1x + b_1y, a_2x + b_2y) \\ &= kF(x,y) \end{aligned}$$

thus, $F(x,y)$ follows the homogeneity property. Now for the next property,

$$\begin{aligned} F(x+v,y+w) &= (a_1(x+v) + b_1(y+w), a_2(x+v) + b_2(y+w)) \\ &= (a_1x + b_1y, a_2x + b_2y) + (a_1v + b_1w, a_2v + b_2w) \\ &= F(x,y) + F(v,w) \end{aligned}$$

it proven that $F(x,y)$ follows the additivity property. Since $F(x,y)$ follows homogeneity and additivity property then it is a linear operator on $\mathbb{R}^2$.

(b) No, because it does not follow the homogeneity property

$$\begin{aligned} F(kx,ky) &= (a_1k^2x^2 + b_1k^2y^2, a_2k^2x^2 + b_2k^2y^2) \\ &= k^2(a_1x^2 + b_1y^2, a_2x^2 + b_2y^2) \\ &= k^2F(x,y) \\ &\neq kF(x,y) \end{aligned}$$

[DF*] **Example:**
Let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ be a basis for a vector space $V$, and let $T : V \to W$ be a linear transformation. Show that if

$$T(\boldsymbol{v}_1) = T(\boldsymbol{v}_2) = T(\boldsymbol{v}_n) = \mathbf{0}$$

then $T$ is the zero transformation.
**Solution:**
The zero transformation will have these properties

$$\begin{aligned} T(\boldsymbol{v}_i + \boldsymbol{v}_j) &= 0 \\ T(\boldsymbol{v}_i) &= 0 \\ T(\boldsymbol{v}_j) &= 0 \\ T(k\boldsymbol{v}_i) &= 0 \end{aligned}$$

for $i, j = 1, 2, \ldots, n$, with $i \neq j$. Since $T : V \to W$ is a linear transformation then it will follow the homogeneity and additivity properties

$$\begin{aligned} T(\boldsymbol{v}_i + \boldsymbol{v}_j) &= T(\boldsymbol{v}_i) + T(\boldsymbol{v}_j) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

$$\begin{aligned} T(k\boldsymbol{v}_i) &= kT(\boldsymbol{v}_i) \\ &= k(0) \\ &= 0 \end{aligned}$$

thus, $T$ is the zero transformation.

[DF*] **Example:**
Let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ be a basis for a vector space $V$, and let $T : V \to V$ be a linear operator. Show that if

$$T(\boldsymbol{v}_1) = \boldsymbol{v}_1, \quad T(\boldsymbol{v}_2) = \boldsymbol{v}_2, \ldots, T(\boldsymbol{v}_n) = \boldsymbol{v}_n$$

then $T$ is the identity transformation on $V$.

**Solution:**
Let $\boldsymbol{w} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n$ be any vector in $V$. Then

$$\begin{aligned} T(\boldsymbol{w}) &= c_1T(\boldsymbol{v}_1) + c_2T(\boldsymbol{v}_2) + \cdots + c_nT(\boldsymbol{v}_n) \\ &= c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n \\ &= \boldsymbol{w} \end{aligned}$$

Since $\boldsymbol{w}$ was an arbitrary vector in $V$, $T$ must be the identity operator.

[DF*] **Example:**
Prove: If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a basis for $V$ and $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_n$ are vectors in $W$, not necessarily distinct, then there exists a linear transformation $T : V \to W$ such that

$$T(\boldsymbol{v}_1) = \boldsymbol{w}_1, \quad T(\boldsymbol{v}_2) = \boldsymbol{w}_2, \ldots, T(\boldsymbol{v}_n) = \boldsymbol{w}_n$$

**Solution:**
If $T : V \to W$ is a linear transformation, then it has to obey these two properties

$$\begin{aligned} T(\mathbf{0}) &= \mathbf{0} \\ T(\boldsymbol{v}_i - \boldsymbol{v}_j) &= T(\boldsymbol{v}_i) - T(\boldsymbol{v}_j) \end{aligned}$$

for all $i, j = 1, 2, \ldots, n$ and $i \neq j$. Now we know that since $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a basis for $V$, then

$$\boldsymbol{v}_i - \boldsymbol{v}_j \neq 0$$

thus

$$\begin{aligned} T(\boldsymbol{v}_i - \boldsymbol{v}_j) &= T(\boldsymbol{v}_i) - T(\boldsymbol{v}_j) \\ &= \boldsymbol{w}_i - \boldsymbol{w}_j \end{aligned}$$

if we make an assumption that $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_n$ are vectors in $W$ not necessarily distinct, then we will encounter this (assuming $\boldsymbol{w}_i = \boldsymbol{w}_j$)

$$\begin{aligned} T(\boldsymbol{v}_i - \boldsymbol{v}_j) &= \boldsymbol{w}_i - \boldsymbol{w}_j \\ &= 0 \end{aligned}$$

the consequence is $\boldsymbol{v}_i - \boldsymbol{v}_j$ will be in kernel of $T$ too besides $\mathbf{0}$. Now, let $\boldsymbol{x}$ be any vector in $V$ with $\boldsymbol{x} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + \boldsymbol{v}_n$, $c_1, c_2, \ldots, c_n$ are scalars then

$$\begin{aligned} T(\boldsymbol{x}) &= T(c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_n\boldsymbol{v}_n) \\ &= c_1T(\boldsymbol{v}_1) + c_2T(\boldsymbol{v}_2) + \cdots + c_nT(\boldsymbol{v}_n) \\ &= c_1\boldsymbol{w}_1 + c_2\boldsymbol{w}_2 + \cdots + c_n\boldsymbol{w}_n \end{aligned}$$

$T(\boldsymbol{x}) \in W$, and for any scalars $k$

$$\begin{aligned} T(k\boldsymbol{x}) &= T(c_1k\boldsymbol{v}_1 + c_2k\boldsymbol{v}_2 + \cdots + c_nk\boldsymbol{v}_n) \\ &= kc_1T(\boldsymbol{v}_1) + kc_2T(\boldsymbol{v}_2) + \cdots + kc_nT(\boldsymbol{v}_n) \\ &= k(c_1\boldsymbol{w}_1 + c_2\boldsymbol{w}_2 + \cdots + c_n\boldsymbol{w}_n) \end{aligned}$$

$T$ follows the homogeneity and additivity properties, thus the linear transformation $T$ is proven to exist.

[DF*] **Example:**
Let $D : P_3 \to P_2$ be the differentiation transformation $D(\boldsymbol{p}) = p'(x)$. What is the kernel of $D$?
**Solution:**
If $p'(x) = 0$, then $p(x)$ is a constant , so $\ker(D)$ consists of all constant polynomials.

[DF*] **Example:**
Let $V$ be the vector space of real-valued functions with continuous derivatives of all orders on the interval $(-\infty, \infty)$, and let $W = F(-\infty, \infty)$ be the vector space of real-valued functions defined on $(-\infty, \infty)$.

(a) Find a linear transformation $T : V \to W$ whose kernel is $P_3$.

(b) Find a linear transformation $T : V \to W$ whose kernel is $P_n$.

**Solution:**

(a) The fourth derivative of any polynomial of degree 3 or less is 0, so

$$T(f(x)) = f^{(4)}(x)$$

has $\ker(T) = P_3$.

(b) The $n+1$-th derivative of any polynomial of degree $n$ or less is 0, so

$$T(f(x)) = f^{(n+1)}(x)$$

has $\ker(T) = P_n$.

[DF*] The vector space $\mathbb{R}^n$ is the "mother" of all real $n$-dimensional vector spaces in the sense that any such space might differ from $\mathbb{R}^n$ in the notation used to represent vectors, but not in its algebraic structure.

**Definition 23.65: One-to-One**

If $T : V \to W$ is a linear transformation from a vector space $V$ to a vector space $W$, then $T$ is said to be one-to-one if $T$ maps distinct vectors in $V$ into distinct vectors in $W$.

**Definition 23.66: Onto**

If $T : V \to W$ is a linear transformation from a vector space $V$ to a vector space $W$, then $T$ is said to be onto (or onto $W$) if every vector in $W$ is the image of at least one vector in $V$.

**Theorem 23.141: Kernel and One-to-One**

If $T : V \to W$ is a linear transformation, then the following statements are equivalent.

(a) $T$ is one-to-one.

(b) $\ker(T) = \{\mathbf{0}\}$

**Theorem 23.142: Linear Operator in a Finite-Dimensional Vector Space**

If $V$ is a finite-dimensional vector space, and if $T : V \to V$ is a linear operator, then the following statements are equivalent.

(a) $T$ is one-to-one.

(b) $\ker(T) = \{\mathbf{0}\}$.

(c) $T$ is onto [i.e., $R(T) = V$]

[DF*] **Dilations and Contractions Are One-to-One and Onto**
If $V$ is a finite-dimensional vector space and $c$ is any nonzero scalar, then the linear operator $T : V \to V$ defined by $T(\boldsymbol{v}) = c\boldsymbol{v}$ is one-to-one and onto.

We know that the operator $T$ is onto (and hence one-to-one) for if $\boldsymbol{v}$ is any vector in $V$ then that vector is the image of the vector $(1/c)\boldsymbol{v}$.

[DF*] **Matrix Operators**
If $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is the matrix operator $T_A(\boldsymbol{x}) = A\boldsymbol{x}$, then it follows from theorem in the Eigenvalues and Eigenvectors section that

(a) The range of $T_A$ is $\mathbb{R}^n$.

(b) $T_A$ is one-to-one.

so $T_A$ is one-to-one and onto if and only if $A$ is invertible.

[DF*] **Shifting Operators**

Let $V = \mathbb{R}^\infty$ be the sequence and consider the linear "shifting operators" on $V$ defined by

$$T_1(u_1, u_2, \dots, u_n, \dots) = (0, u_1, u_2, \dots, u_n, \dots)$$
$$T_2(u_1, u_2, \dots, u_n, \dots) = (u_2, u_3, \dots, u_n, \dots)$$

$T_1$ is one-to-one because distinct dequences in $\mathbb{R}^\infty$ obviously have distinct images. This operator is not onto because no vector in $\mathbb{R}^\infty$ maps into the sequence $(1, 0, 0, \dots, 0, \dots)$, for example.

$T_2$ is not one-to-one because the vectors $(1, 0, 0, \dots, 0, \dots)$ and $(2, 0, 0, \dots, 0, \dots)$ both map into $(0, 0, 0, \dots, 0, \dots)$. This operator is onto because every possible sequence of real numbers can be obtained with an appropriate choice of the numbers $u_2, u3, \dots, u_n, \dots$.

[DF*] **Basic Transformations That Are One-to-One and Onto**

The linear transformation $T_1 : P_3 \to \mathbb{R}^4$ and $T_2 : M_{22} \to \mathbb{R}^4$ defined by

$$T_1(a + bx + cx^2 + dx^3) = (a, b, c, d)$$

$$T_2\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = (a, b, c, d)$$

are both one-to-one and onto. Their kernels contain only the zero vector.

[DF*] **A One-to-One Linear Transformation**

Let $T : P_n \to P_{n+1}$ be the linear transformation

$$T(\boldsymbol{p}) = T(p(x)) = xp(x)$$

If

$$\boldsymbol{p} = p(x) = c_0 + c_1 x + \cdots + c_n x^n$$
$$\boldsymbol{q} = q(x) = d_0 + d_1 x + \cdots + d_n x^n$$

are distinct polynomials, then they differ in at least one coefficient. Thus,

$$T(\boldsymbol{p}) = c_0 x + c_1 x^2 + \cdots + c_n x^{n+1}$$
$$T(\boldsymbol{q}) = d_0 x + d_1 x^2 + \cdots + d_n x^{n+1}$$

also differ in at least one coefficient. It follows that $T$ is one-to-one since it maps distinct polynomials $\boldsymbol{p}$ and $\boldsymbol{q}$ into distinct polynomials $T(\boldsymbol{p})$ and $T(\boldsymbol{q})$.

[DF*] **A Transformation That Is Not One-to-One**

Let

$$D : C^1(-\infty, \infty) \to F(-\infty, \infty)$$

be the differentiation transformation. This linear transformation is not one-to-one because it maps functions that differ by constant into the same function. For example,

$$D(x^2) = D(x^2 + 1) = 2x$$

Remember that $F(-\infty, \infty)$ represents the set of functions with derivatives of all orders on $(-\infty, \infty)$, while $C^1(-\infty, \infty)$ represents functions with the first derivatives on $(-\infty, \infty)$ that is continuous, and $C(-\infty, \infty)$ is the set of continuous functions on $(-\infty, \infty)$.

[DF*] Let $T : V \to W$ be a linear transformation, where $V$ and $W$ are finite-dimensional. Then

1. If $\dim(W) < \dim(V)$, then $T$ cannot be one-to-one.
2. If $\dim(V) < \dim(W)$, then $T$ cannot be onto.

These observations tell us, that any linear tansformation from $\mathbb{R}^3$ to $\mathbb{R}^2$ must map some distinct points of $\mathbb{R}^3$ into the same point in $\mathbb{R}^2$, and it also tells us that there is no linear transformation that maps $\mathbb{R}^2$ onto all of $\mathbb{R}^3$.

**Definition 23.67: Isomorphism**

If a linear transformation $T : V \to W$ is both one-to-one and onto, then $T$ is said to be an isomorphism, and the vector spacesd $V$ and $W$ are said to be isomorphic.

[DF*] Isomorphic vector spaces have the same "algebraic form," even though they may consist of different kinds of objects.

THe isomorphism

$$a_0 + a_1 x + a_2 x^2 \vec{T}(a_0, a_1, a_2)$$

matches up vector operations in $P_2$ and $\mathbb{R}^3$.

**Theorem 23.143: Isomorphism for $n$-Dimensional Vector Space**

Every real $n$-dimensional vector space is isomorphic to $\mathbb{R}^n$.

[DF*] Let $V$ be a real $n$-dimensional vector space. To prove that $V$ is isomorphic to $\mathbb{R}^n$ we must find a linear transformation $T : V \to \mathbb{R}^n$ that is one-to-one and onto. For this purpose, let

$$\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$$

be any basis for $V$, let

$$\boldsymbol{u} = k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_n\boldsymbol{v}_n \tag{23.241}$$

be the representation of a vector $\boldsymbol{u}$ in $V$ as a linear combination of the basis vectors, and define the transformation $T : V \to \mathbb{R}^n$ by

$$T(\boldsymbol{u}) = (k_1, k_2, \ldots, k_n) \tag{23.242}$$

We will show that $T$ is an isomorphism (linear, one-to-one, and onto). To prove the lienarity, let $\boldsymbol{u}$ and $\boldsymbol{v}$ be vectors in $V$, let $c$ be a scalar, and let

$$\begin{aligned} \boldsymbol{u} &= k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_n\boldsymbol{v}_n \\ \boldsymbol{v} &= d_1\boldsymbol{v}_1 + d_2\boldsymbol{v}_2 + \cdots + d_n\boldsymbol{v}_n \end{aligned} \tag{23.243}$$

be the representations of $\boldsymbol{u}$ and $\boldsymbol{v}$ as linear combinations of the basis vectors. Then it follows that

$$\begin{aligned} T(c\boldsymbol{u}) &= T(ck_1\boldsymbol{v}_1 + ck_2\boldsymbol{v}_2 + \cdots + ck_n\boldsymbol{v}_n) \\ &= (ck_1, ck_2, \ldots, ck_n) \\ &= c(k_1, k_2, \ldots, k_n) \\ &= cT(\boldsymbol{u}) \end{aligned}$$

and it folows that

$$\begin{aligned} T(\boldsymbol{u}+\boldsymbol{v}) &= T((k_1+d_1)\boldsymbol{v}_1+(k_2+d_2)\boldsymbol{v}_2+\cdots+(k_n+d_n)\boldsymbol{v}_n) \\ &= (k_1+d_1, k_2+d_2, \ldots, k_n+d_n) \\ &= (k_1, k_2, \ldots, k_n)+(d_1, d_2, \ldots, d_n) \\ &= T(\boldsymbol{u})+T(\boldsymbol{v}) \end{aligned}$$

which shows that $T$ is linear. To show that $T$ is one-to-one, we must show that if $\boldsymbol{u}$ and $\boldsymbol{v}$ are distinct vectors in $V$, then so are their images in $\mathbb{R}^n$. But if $\boldsymbol{u} \neq \boldsymbol{v}$, and if the representations of these vectors in terms of the basis vectors, then we must have $k_i \neq d_i$ for at least one $i$. Thus,

$$T(\boldsymbol{u}) = (k_1, k_2, \ldots, k_n) \neq (d_1, d_2, \ldots, d_n) = T(\boldsymbol{v})$$

which shows that $\boldsymbol{u}$ and $\boldsymbol{v}$ have distinct images under $T$. Finally, the transformation $T$ is onto, for if

$$\boldsymbol{w} = (k_1, k_2, \ldots, k_n)$$

is any vector in $\mathbb{R}^n$, then it folows that $\boldsymbol{w}$ is the image under $T$ of the vector

$$\boldsymbol{u} = k1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_n\boldsymbol{v}_n$$

**Remark**
Note that the isomorphism $T$ of the foregoing proof is the coordinate map

$$\boldsymbol{u}\vec{T}(k_1, k_2, \ldots, k_n) = (\boldsymbol{u})_S$$

that maps $\boldsymbol{u}$ into its coordinate vector with respect to the basis $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$. Since there are generally many possible bases for a given vector space $V$, there are generally many possible isomorphisms between $V$ and $\mathbb{R}^n$, one for each different basis.

[DF*] **The Natural Isomorphism from $P_{n-1}$ to $\mathbb{R}^n$**
The mapping

$$a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \xrightarrow{T} (a_0, a_1, \ldots, a_{n-1})$$

from $P_{n-1}$ to $\mathbb{R}^n$ is one-to-one, onto, and linear. This is called the natural isomorphism from $P_{n-1}$ to $\mathbb{R}^n$ because, as the following computations show, it maps the natural basis $\{1, x, x^2, \ldots, x^{n-1}\}$ for $P_{n-1}$ into the standard basis for $\mathbb{R}^n$:

$$\begin{array}{ccccc} 1 & = & 1+0x+0x^2+\cdots+0x^{n-1} & \xrightarrow{T} & (1,0,0,\ldots,0) \\ x & = & 0+x+0x^2+\cdots+0x^{n-1} & \xrightarrow{T} & (0,1,0,\ldots,0) \\ \vdots & & \vdots & \vdots & \vdots \\ x^{n-1} & = & 0+0x+0x^2+\cdots+0x^{n-2}+x^{n-1} & \xrightarrow{T} & (0,0,0,\ldots,0,1) \end{array}$$

[DF*] **The Natural Isomorphism from $M_{22}$ to $\mathbb{R}^4$**
The matrices

$$E_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad E_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad E_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

form a basis for the vector space $M_{22}$ of $2 \times 2$ matrices. An isomorphism $T : M_{22} \to \mathbb{R}^4$ can be constructed by first writing a matrix $A$ in $M_{22}$ in terms of the basis vectors as

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} = a_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + a_2 \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + a_3 \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} + a_4 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

and then defining $T$ as

$$T(A) = (a_1, a_2, a_3, a_4)$$

Thus, for example,

$$\begin{bmatrix} 1 & -3 \\ 4 & 6 \end{bmatrix} \xrightarrow{T} (1, -3, 4, 6)$$

More generally, this idea can be used to show that the vector space $M_{mn}$ of $m \times n$ matrices with real entries is isomorphic to $\mathbb{R}^{mn}$.

[DF*] **Differentiation by Matrix Multiplication**

Consider the differentiation transformation $D : P_3 \to P_2$ on the vector space of polynomials of degree three or less. If we map $P_3$ and $P_2$ into $\mathbb{R}^4$ and $\mathbb{R}^3$, respectively, by the natural isomorphisms, then the transformation $D$ produces a corresponding matrix transformation from $\mathbb{R}^4$ to $\mathbb{R}^3$. Specifically, the derivative transformation

$$a_0 + a_1x + a_2x^2 + a_3x^3 \xrightarrow{T} a_1 + 2a_2x + 3a_3x^2$$

produces the matrix transformation

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ 2a_2 \\ 3a_3 \end{bmatrix}$$

Thus, for example, the derivative

$$\frac{d}{dx}(2 + x + 4x^2 - x^3) = 1 + 8x - 3x^2$$

can be calculated as the matrix product

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 4 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 8 \\ -3 \end{bmatrix}$$

This idea is useful for constructing numerical algorithms to perform derivative calculations.

```
Suppose we have a polynomial P_{3} : a0 + a1x + a2x^2 + a3x^3
We will compute the differentiation transformation
D: P_{3} -> P_{2} with matrix multiplication

 A = [[0,1,0,0],[0,0,2,0],[0,0,0,3]]

 x = [[a0],[a1],[a2],[a3]]

 A*x = [[a1],[2*a2],[3*a3]]

 d/dx (a0 + a1x + a2x^2 + a3x^3)  = a1 + 2*a2 x + 3*a3 x^2

 If we try to input value for a0, a1, a2, a3

 a0 = 2, a1 = 1, a2 = 4, a3 = -1

 d/dx (2 + x + 4 x^2 - x^3)  = 1 + 8 x + -3 x^2
```

**Figure 23.213:** *The computation of the differentiation transformation from $P_3$ to $P_2$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Differentiation of Polynomial with GiNaC/main.cpp**).*

[DF*] In the case where $V$ is a real $n$-dimensional inner product space, both $V$ and $\mathbb{R}^n$ have, in addition to their algebraic structure, a geometric structure arising from their respective inner products. Thus, it is reasonable to inquire if there exists an isomorphism from $V$ to $\mathbb{R}^n$ that preserves the geometric structure as well as the algebraic structure. For example, we would want orthogonal vectors in $V$ to have orthogonal counterparts in $\mathbb{R}^n$, and we would want orthonormal sets in $V$ to correspond to orthonormal sets in $\mathbb{R}^n$.

In order for an isomorphism to preserve geometric structure, it obviously has to preserve inner products, since notions of length, angle, and orthogonality are all based on the inner product. Thus, if $V$ and $W$ are inner product spaces, then we call an isomorphism $T : V \to W$ an inner product space isomorphism if

$$< T(\boldsymbol{u}), T(\boldsymbol{v}) >=< \boldsymbol{u}, \boldsymbol{v} >$$

It can be proved that if $V$ is any real $n$-dimensional inner product space and $\mathbb{R}^n$ has the Euclidean inner product (the dot product), then there exists an inner product space isomorphism from $V$ to $\mathbb{R}^n$. Under such an isomorphism, the inner product space $V$ has the same algebraic and geometric structure as $\mathbb{R}^n$. In this sense, every $n$-dimensional inner product space is a "carbon copy" of $\mathbb{R}^n$ with the Euclidean inner product that differs only in the notation used to represent vectors.

[DF*] **An Inner Product Space Isomorphism**

Let $\mathbb{R}^n$ be the vector space of real $n$-tuples in comma-delimited form, let $M_n$ be the vector space of real $n \times 1$ matrices, let $\mathbb{R}^n$ have the Euclidean inner product $< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{u} \cdot \boldsymbol{v}$, and let $M_n$ have the inner product $< \boldsymbol{u}, \boldsymbol{v} >= \boldsymbol{u}^T \boldsymbol{v}$ in which $\boldsymbol{u}$ and $\boldsymbol{v}$ are expressed in column form. The mapping $T : \mathbb{R}^n \to M_n$ defined by

$$(v_1, v_2, \ldots, v_n) \xrightarrow{T} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

is an inner product space isomorphism, so the distinction between the inner product space $\mathbb{R}^n$ and the inner product space $M_n$ is essentially notational, a fact that we have used many times in this text.

[DF*] **Example:**

In each part, find $\ker(T)$, and determine whether the linear transformation $T$ is one-to-one.

(a) $T : \mathbb{R}^2 \to \mathbb{R}^2$, where $T(x, y) = (5x - y, 0)$

(b) $T : \mathbb{R}^2 \to \mathbb{R}^3$, where $T(x, y) = (y, x, x - y)$

**Solution:**

(a) To find $\ker(T)$ we need to compute $T(x, y) = (0, 0)$. $T(x, y) = 0, 0$ if

$$\begin{aligned} 5x - y &= 0 \\ x &= \frac{1}{5}y \end{aligned}$$

so

$$\ker(T) = \{k\left(\frac{1}{5}, 1\right)\}$$

since $\ker(T) \neq \{\mathbf{0}\}$ then $T$ is not one-to-one.

(b) By inspection, we need to have

$$\begin{aligned} y &= 0 \\ x &= 0 \\ x - y &= 0 \\ x &= y = 0 \end{aligned}$$

so, $\ker(T) = (0,0,0)$, since $\ker(T) = \mathbf{0}$ then $T$ is one-to-one.

[DF*] **Example:**
In each part, determine whether multiplication by $A$ is one-to-one linear transformation.

(a)

$$A = \begin{bmatrix} 1 & -3 & 2 \\ -2 & 6 & -4 \end{bmatrix}$$

(b)

$$A = \begin{bmatrix} 2 & 5 \\ -1 & 3 \\ 2 & 4 \end{bmatrix}$$

**Solution:**

(a) By inspection we will get

$$T(x,y) = (x - 3y + 2z, -2x + 6x - 4y)$$

in order for $T(x,y) = (0,0)$ we need to obey

$$\begin{aligned} x - 3y + 2z &= 0 \\ x &= 3y - 2z \\ x &= 3s - 2t \\ y &= s \\ z &= t \end{aligned}$$

with $s, t$ are scalar, or the parameters. The second term $-2x + 6x - 4y$ is a linear combination of the first term, so we do not need to recompute. We will obtain

$$\ker(T) = \{s(3,1,0) + t(-2,0,1)\}$$

since $\ker(T) \neq \mathbf{0}$, then multiplication by $A$ is not one-to-one.

(b) By inspection we will get

$$T(x,y) = (2x + 5y, -x + 3y, 2x + 4y)$$

in order for $T(x,y) = (0,0)$ we need to obey

$$\begin{aligned} 2x + 5y &= 0 \\ -x + 3y &= 0 \\ 2x + 4y &= 0 \\ x &= -\frac{5}{2}y \\ x &= 3y \\ x &= -2y \end{aligned}$$

the equations above can only be solved if and only if $x = 0$ and $y = 0$, thus $\ker(T) = \mathbf{0}$, and the multiplication by $A$ is a one-to-one linear transformation.

[DF*] **Example:**
As indicated in the accompanying figure, let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be the orthogonal projection on the line $y = x$

**Figure 23.214**

(a) Find the kernel of $T$.

(b) Is $T$ one-to-one? Justify your conclusion.

**Solution:**

(a) All points on the line $y = -x$ are mapped to 0, so $\ker(T) = \{k(-1, 1)\}$.

(b) No, because $\ker(T) \neq \mathbf{0}$, so $T$ is not one-to-one.

[DF*] **Example:**
Let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be the linear operator that reflects each point about the $y$-axis.

(a) Find the kernel of $T$.

(b) Is $T$ one-to-one? Justify your conclusion.

**Solution:**

(a) All points on the line are mapped to $T(x, y) = (-x, y)$, so $\ker(T) = \{(0, 0)\} = \mathbf{0}$.

(b) $\ker(T) = \mathbf{0}$, so $T$ is one-to-one.

[DF*] **Example:**
In each part, use the given information to determine whether the linear transformation $T$ is one-to-one

(a) $T : \mathbb{R}^m \to \mathbb{R}^m$; $\text{nullity}(T) = 1$

(b) $T : \mathbb{R}^n \to \mathbb{R}^n$; $\text{rank}(T) = n$

(c) $T : \mathbb{R}^m \to \mathbb{R}^n$; $n < m$

(d) $T : \mathbb{R}^n \to \mathbb{R}^n$; $R(T) = \mathbb{R}^n$

**Solution:**

(a) If $\text{nullity}(T) = 1$, $T$ cannot be one-to-one.

(b) Since $\text{rank}(T) = n$ / $\text{nullity}(T) = 0$ then $T$ is one-to-one

(c) $T : \mathbb{R}^m \to \mathbb{R}^n$ is not one-to-one since the dimension of the vector space that represent the image of the transformation ($n$) is smaller then the dimension of the vector space that represent the original vectors ($m$).

(d) Based on theorem, since the dimension of the original vector space and the vector space of the images are finite and the same, if $T$ is onto then $T$ is one-to-one.

[DF*] **Example:**
Prove: If $V$ and $W$ are finite-dimensional vector spaces such that $\dim(W) < \dim(V)$, then there is no one-to-one linear transformation $T : V \to W$.
**Solution:**
If there were such a transformation $T$, then it would have nullity 0 (because it would be one-to-one), and have rank no greater than the dimension of $W$ (because its range is a subspace $W$).

Then by the dimension theorem

$$\dim(V) = \text{rank}(T) + \text{nullity}(T) \leq \dim(W)$$

which contradicts that $\dim(W) < \dim(V)$. Thus there is no such transformation.

[DF*] **Example:**

(a) Find an isomorphism between the vector space of all $3 \times 3$ symmetric matrices and $\mathbb{R}^6$.

(b) Find two different isomorphisms between the vector space of all $2 \times 2$ matrices and $\mathbb{R}^4$.

(c) Find an isomorphism between the vector space of all polynomials of degree at most 3 such that $p(0) = 0$ and $\mathbb{R}^3$.

(d) Find an isomorphism between the vector spaces $\text{span}\{1, \sin(x), \cos(x)\}$ and $\mathbb{R}^3$.

**Solution:**

(a)

$$T\left(\begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}\right) = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}$$

(b)

$$T\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

and

$$T\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

(c)

$$T(ax^3 + bx^2 + cx) = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

(d)

$$T(a + b\sin(x) + c\cos(x)) = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

[DF*] **Example:**
Let $V$ be the vector space $C^1[0,1]$ and let $T : V \to R$ be defined by

$$T(\boldsymbol{f}) = f(0) + 2f'(0) + 3f'(1)$$

verify that $T$ is a linear transformation. Determine whether $T$ is one-to-one, and justify your conclusion.
**Solution:**
We will examine the basic properties to prove if $T$ is a linear transformation

$$\begin{aligned} T(k\boldsymbol{f}) &= kf(0) + 2kf'(0) + 3kf'(1) \\ &= k(f(0) + 2f'(0) + 3f'(1)) \\ &= kT(\boldsymbol{f}) \end{aligned}$$

$$\begin{aligned} T(\boldsymbol{f} + \boldsymbol{g}) &= (f(0) + 2f'(0) + 3f'(1)) + (g(0) + 2g'(0) + 3g'(1)) \\ &= (f(0) + g(0)) + 2(f'(0) + g'(0)) + 3(f'(1) + g'(1)) \\ &= T(\boldsymbol{f}) + T(\boldsymbol{g}) \end{aligned}$$

Thus $T$ is a linear transformation.

Let $\boldsymbol{f} = f(x) = x^2(x-1)^2$, then

$$f'(x) = 4x^3 - 6x^2 + 2x = 2x(x-1)(2x-1)$$

so

$$\begin{aligned} f(0) &= 0 \\ f'(0) &= 0 \\ f'(1) &= 0 \end{aligned} \tag{23.244}$$

$$T(\boldsymbol{f}) = f(0) + 2f'(0) + 3f'(1) = 0$$

so $\ker(T) \neq \mathbf{0}$ and $T$ is not one-to-one.

[DF*] **Example:**
Derive a method for using matrix multiplication to differentiate functions in the vector space $\text{span}(1, \sin(x), \cos(x), \sin(2x), \cos(2x))$. Use your method to find the derivative of $3 - 4\sin(x) + \sin(2x) + 5\cos(2x)$.
**Solution:**
The derivative transformation

$$a_0 + a_1\sin(x) + a_2\cos(x) + a_3\sin(2x) + a_4\cos(2x) \underset{\rightarrow}{D} a_1\cos(x) - a_2\sin(x) + 2a_3\cos(2x) - 2a_4\sin(x)$$

produces the matrix transformation

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}$$

now, if we want to compute the derivative of $3 - 4\sin(x) + \sin(2x) + 5\cos(2x)$, we can calculate with the matrix product.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & -2 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ -4 \\ 0 \\ 1 \\ 5 \end{bmatrix} = \begin{bmatrix} -4 \\ 0 \\ 2 \\ -10 \end{bmatrix}$$

or it can be written as

$$3 - 4\sin(x) + \sin(2x) + 5\cos(2x) \underset{\rightarrow}{D} -4\cos(x) + 2\cos(2x) - 10\sin(2x)$$

[DF*] **Example:**
Let $\boldsymbol{a}$ be a fixed vector in $\mathbb{R}^3$. Does the formula $T(\boldsymbol{v}) = \boldsymbol{a} \times \boldsymbol{v}$ define a one-to-one linear operator on $\mathbb{R}^3$ ? Explain your reasoning.
**Solution:**
No, because

$$T(\boldsymbol{a}) = \boldsymbol{a} \times \boldsymbol{a} = \boldsymbol{0}$$

so $\ker(T) \neq \boldsymbol{0}$ and $T$ is not one-to-one.

[DF*] **Example:**
Does an inner product space isomorphism map orthonormal sets to orthonormal sets? Justify your answer.
**Solution:**
Yes, since

$$< T(\boldsymbol{u}), T(\boldsymbol{v}) >=< \boldsymbol{u}, \boldsymbol{v} >$$

so

$$\begin{aligned} ||T(\boldsymbol{u})|| &= \sqrt{< T(\boldsymbol{u}), T(\boldsymbol{u}) >} \\ &= \sqrt{< \boldsymbol{u}, \boldsymbol{u} >} \\ &= ||\boldsymbol{u}|| \\ &= 1 \end{aligned}$$

and $T(\boldsymbol{u})$ is orthogonal to $T(\boldsymbol{v})$ if $\boldsymbol{u}$ is orthogonal to $\boldsymbol{v}$.

**Definition 23.68: Composition of Linear Transformations**

If $T_1 : U \to V$ and $T_2 : V \to W$ are linear transformations, then the composition of $T_2$ with $T_1$, denoted by $T_2 \circ T_1$ (which is read "$T_2$ circle $T_1$"), is the function defined by the formula

$$(T_2 \circ T_1)(\boldsymbol{u}) = T_2(T_1(\boldsymbol{u})) \tag{23.245}$$

where $\boldsymbol{u}$ is a vector in $U$.

**Remark**
Observe that this definition requires that the domain $T_2$ (which is $V$) contain the range $T_1$. This is essential for the formula $T_2(T_1(\boldsymbol{u}))$ to make sense.

**Theorem 23.144: Linear Transformations in Composition**

If $T_1 : U \to V$ and $T_2 : V \to W$ are linear transformations, then $(T_2 \circ T_1) : U \to W$ is also a linear transformation.

[DF*] **Composition of Linear Transformations**
Let $T_1 : P_1 \to P_2$ and $T_2 : P_2 \to P_2$ be the linear transformations given by the formulas

$$\begin{aligned} T_1(p(x)) &= xp(x) \\ T_2(p(x)) &= p(2x+4) \end{aligned}$$

Then the composition $(T_2 \circ T_1) : P_1 \to P_2$ is given by the formula

$$\begin{aligned} (T_2 \circ T_1)(p(x)) &= T_2(T_1(p(x))) \\ &= T_2(xp(x)) \\ &= (2x+4)p(2x+4) \end{aligned}$$

In particular, if $p(x) = c_0 + c_1 x$, then

$$\begin{aligned} (T_2 \circ T_1)(p(x)) &= (T_2 \circ T_1)(c_0 + c_1 x) \\ &= (2x+4)(c_0 + c_1(2x+4)) \\ &= c_0(2x+4) + c_1(2x+4)^2 \end{aligned}$$

[DF*] **Composition with the Identity Operator**
If $T : V \to V$ is any linear operator, and if $I : V \to V$ is the identity operator, then for all vectors $\boldsymbol{v}$ in $V$, we have

$$\begin{aligned} (T \circ I)(\boldsymbol{v}) &= T(I(\boldsymbol{v})) = T(\boldsymbol{v}) \\ (I \circ T)(\boldsymbol{v}) &= I(T(\boldsymbol{v})) = T(\boldsymbol{v}) \end{aligned}$$

It follows that $T \circ I$ and $I \circ T$ are the same as $T$; that is,

$$T \circ I = T \tag{23.246}$$

$$I \circ T = T \tag{23.247}$$

[DF*] Compositions can be defined for more than two linear transformations. For example, if

$$\begin{aligned} T_1 &: U \to V \\ T_2 &: V \to W \\ T_3 &: W \to Y \end{aligned}$$

are linear transformations, then the composition $T_3 \circ T_2 \circ T_1$ is defined by

$$(T_3 \circ T_2 \circ T_1)(\boldsymbol{u}) = T_3(T_2(T_1(\boldsymbol{u})))$$

[DF*] A matrix operator $T_A : \mathbb{R}^n \to \mathbb{R}^n$ is one-to-one if and only if the matrix $A$ is invertible, in which case the inverse operator is $T_{A^{-1}}$.

We then showed that if $\boldsymbol{w}$ is the image of a vector $\boldsymbol{x}$ under the operator $T_A$, then $\boldsymbol{x}$ is the image under $T_{A^{-1}}$ of the vector $\boldsymbol{w}$.

Recall that if $T : V \to W$ is a linear transformations, then the range of $T$, denoted by $R(T)$, is the subspace of $W$ consisting of all images under $T$ of vectors in $V$. If $T$ is one-to-one, then each vector $\boldsymbol{w}$ in $R(T)$ is the image of a unique vector $\boldsymbol{v}$ in $V$. This uniqueness allows us to define a new function, called the inverse of $T$ and denoted by $T^{-1}$, that maps $\boldsymbol{w}$ back into $\boldsymbol{v}$.
[DF*] It follows from the definition of $T^{-1}$ that

$$T^{-1}(T(\boldsymbol{v})) = T^{-1}(\boldsymbol{w}) = \boldsymbol{v} \tag{23.248}$$

$$T(T^{-1}(\boldsymbol{w})) = T(\boldsymbol{v}) = \boldsymbol{w} \tag{23.249}$$

so that $T$ and $T^{-1}$, when applied in succession in either order, cancel the effect of each other.

**Remark**
It is important to note that if $T : V \to W$ is a one-to-one linear transformation, then the domain of $T^{-1}$ is the range of $T$, where the range may or may not be all of $W$. However, in the special case where $T : V \to V$ is a one-to-one linear operator and $V$ is $n$-dimensional, then it follows that $T$ must also be onto, so the domain $T^{-1}$ is all of $V$.
[DF*] **An Inverse Transformation**
The linear transformation $T : P_n \to P_{n+1}$ given by

$$T(\boldsymbol{p}) = T(p(x)) = xp(x)$$

is one-to-one; thus, $T$ has an inverse. In this case the range of $T$ is not all of $P_{n+1}$ but rather the subspace of $P_{n+1}$ consisting of polynomials with a zero constant term. This is evident from the formula for $T$:

$$T(c_0 + c_1x + \cdots + c_nx^n) = c_0x + c_1x^2 + \cdots + c_nx^{n+1}$$

It follows that $T^{-1} : R(T) \to P_n$ is given by the formula

$$T^{-1}(c_0x + c_1x^2 + \cdots + c_nx^{n+1}) = c_0 + c_1x + \cdots + c_nx^n$$

For example, in the case where $n \geq 3$,

$$T^{-1}(2x - x^2 + 5x^3 + 3x^4) = 2 - x + 5x^2 + 3x^3$$

**Theorem 23.145: Composition of One-To-One Linear Transformations**

If $T_1 : U \to V$ and $T_2 : V \to W$ are one-to-one linear transformations, then:

(a) $T_2 \circ T_1$ is one-to-one.

(b) $(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1}$

[DF*] The inverse of a composition is the composition of the inverses in the reverse order. This result can be extended to compositions of three or more linear transformations; for example,

$$(T_3 \circ T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1} \circ T_3^{-1} \tag{23.250}$$

In the case where $T_A, T_B$, and $T_C$ are matrix operators on $\mathbb{R}^n$, the formula above can be written as

$$(T_C \circ T_B \circ T_A)^{-1} = T_A^{-1} \circ T_B^{-1} \circ T_C^{-1} \tag{23.251}$$

or alternatively as

$$(T_{CBA})^{-1} = T_{A^{-1}B^{-1}C^{-1}} \tag{23.252}$$

[DF*] **Example:**
For the following, determine if the composition $(T_2 \circ T_1)$ is defined, and if so find the domain and codomain of $(T_2 \circ T_1)$, and find $T_2 \circ T_1$.

(a) $T_1(x,y) = (3y, 2x)$, $T_2(x,y) = (x - 2y, x + y)$

(b) $T_1(x,y,z) = (y - z, x)$, $T_2(x,y) = (x, y, x - y)$

(c) $T_1(x,y) = (x + y, y - x, x - y)$, $T_2(x,y) = (y, x + y, 2x)$

(d) $T_1(x,y) = (4x, -y)$, $T_2(x,y) = (-x, 4y, x - 4y)$

**Solution:**

(a)

$$\begin{aligned} T_2(T_1(x,y)) &= T_2(3y, 2x) \\ &= (3y - 2(2x), 3y + 2x) \\ &= (3y - 4x, 3y + 2x) \end{aligned}$$

(b)

$$\begin{aligned} T_2(T_1(x,y,z)) &= T_2(y - z, x) \\ &= (y - z, x, y - z - x) \end{aligned}$$

(c)

$$T_2(T_1(x,y)) = T_2(x + y, y - x, x - y)$$

Since we know that the domain of $T_2$ is $(x,y) \in \mathbb{R}^2$, not $(x + y, y - x, x - y) \in \mathbb{R}^3$, then we cannot continue further and it cannot be defined.

(d) $T_1(x,y) = (4x, -y)$, $T_2(x,y) = (-x, 4y, x - 4y)$

$$\begin{aligned} T_2(T_1(x,y)) &= T_2(4x, -y) \\ &= (-(4x), 4(-y), 4x - 4(-y)) \\ &= (-4x, -4y, 4x + 4y) \end{aligned}$$

[DF*] **Example:**
Let $T_1 = M_{22} \to R$ and $T_2 : M_{22} \to M_{22}$ be the linear transformations given by $T_1(A) = \text{tr}(A)$ and $T_2(A) = A^T$.

(a) Find $(T_1 \circ T_2)(A)$, where

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

(b) Can you find $(T_2 \circ T_1)(A)$? Explain.

**Solution:**

(a)
$$(T_1 \circ T_2)(A) = T_1(A^T) = \text{tr}\left(\begin{bmatrix} a & c \\ b & d \end{bmatrix}\right) = a + d$$

(b) $(T_2 \circ T_1)(A)$ does not exist because $T_1(A)$ is not a $2 \times 2$ matrix.

[DF*] **Example:**
Let $T_1 : V \to V$ be the dilation $T_1(\boldsymbol{v}) = 4\boldsymbol{v}$. Find a linear operator $T_2 : V \to V$ such that $T_1 \circ T_2 = I$ and $T_2 \circ T_1 = I$.
**Solution:** We know that
$$T_2(\boldsymbol{v}) = \frac{1}{4}\boldsymbol{v}$$
then
$$(T_1 \circ T_2)(\boldsymbol{v}) = T_1\left(\frac{1}{4}\boldsymbol{v}\right) = 4\left(\frac{1}{4}\boldsymbol{v}\right) = \boldsymbol{v}$$
and
$$(T_2 \circ T_1)(\boldsymbol{v}) = T_2(4\boldsymbol{v}) = \frac{1}{4}(4\boldsymbol{v}) = \boldsymbol{v}$$

[DF*] **Example:**
Let $T : \mathbb{R}^3 \to \mathbb{R}^3$ be the orthogonal projection of $\mathbb{R}^3$ onto the $yz$-plane. Show that $T \circ T = T$.

**Solution:**
By inspection, the orthogonal projection of $\mathbb{R}^3$ onto the $yz$-plane will be
$$T(x, y, z) = (0, y, z)$$
Then
$$T(T(x, y, z)) = T(x, y, 0) = (x, y, 0) = T(x, y, z)$$
so
$$T \circ T = T$$

[DF*] **Example:**
In each part, let $T : \mathbb{R}^3 \to \mathbb{R}^3$ be multiplication by $A$. Determine whether $T$ has an inverse; if so, find
$$T^{-1}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right)$$

(a)
$$A = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 1 & 1 \\ -1 & 1 & 0 \end{bmatrix}$$

(b)
$$A = \begin{bmatrix} 2 & 6 & 10 \\ 0 & 1 & -1 \\ 2 & 4 & 6 \end{bmatrix}$$

**Solution:** By theorem we know that if $\det(A) = 0$, then $T$ does not have an inverse, while if $\det(A) \neq 0$, then $T$ has an inverse.

(a) Since $\det(A) = -2 \neq 0$, then $T$ has an inverse

$$A^{-1} = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & -1 \\ \frac{1}{2} & -\frac{1}{2} & 0 \\ -1 & 2 & 1 \end{bmatrix}$$

so

$$T^{-1}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = A^{-1}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{2}x_1 - \frac{1}{2}x_2 - x_3 \\ \frac{1}{2}x_1 - \frac{1}{2}x_2 \\ -x_1 + 2x_2 + x_3 \end{bmatrix}$$

(b) Since $\det(A) = -12 \neq 0$, then $T$ has an inverse

$$A^{-1} = \begin{bmatrix} -\frac{5}{6} & -\frac{1}{3} & \frac{4}{3} \\ \frac{1}{6} & \frac{2}{3} & -\frac{1}{6} \\ \frac{1}{6} & -\frac{1}{3} & -\frac{1}{6} \end{bmatrix}$$

so

$$T^{-1}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = A^{-1}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\frac{5}{6}x_1 - \frac{1}{3}x_2 + \frac{4}{3}x_3 \\ \frac{1}{6}x_1 + \frac{2}{3}x_2 - \frac{1}{6}x_3 \\ \frac{1}{6}x_1 - \frac{1}{3}x_2 - \frac{1}{6}x_3 \end{bmatrix}$$

```
Matrix A_1:
   1.0000   3.0000   1.0000
   1.0000   1.0000   1.0000
  -1.0000   1.0000        0

Matrix A_2:
    2.0000    6.0000   10.0000
         0    1.0000   -1.0000
    2.0000    4.0000    6.0000

det(A_1) = -2
det(A_2) = -12
rank(A_1) = 3
rank(A_2) = 3

(A_1)^{-1} =
    0.5000  -0.5000  -1.0000
   0.5000  -0.5000        0
  -1.0000   2.0000   1.0000


(A_2)^{-1} =
   -0.8333  -0.3333   1.3333
   0.1667   0.6667  -0.1667
   0.1667  -0.3333  -0.1667
```

**Figure 23.215:** *The computation to compute the determinant and inverse of matrices A from the example above. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Determinant and Inverse of Square Matrices with Armadillo/main.cpp**).*

[DF*] **Example:**
Let $T : \mathbb{R}^n \to \mathbb{R}^n$ be the linear operator defined by the formula

$$T(x_1, x_2, \ldots, x_n) = (a_1x_1, a_2x_2, \ldots, a_nx_n)$$

where $a_1, a_2, \ldots, a_n$ are constants.

(a) Under what conditions will $T$ have an inverse?

(b) Assuming that the conditions determined in part(a) are satisfied, find a formula for $T^{-1}(x_1, x_2, \ldots, x_n)$.

**Solution:**

(a) For $T$ to have an inverse, all the $a_i, i = 1, 2, \ldots, n$ must be nonzero.

(b)
$$T^{-1}(x_1, x_2, \ldots, x_n) = \left(\frac{1}{a_1}x_1, \frac{1}{a_2}x_2, \ldots, \frac{1}{a_n}x_n\right)$$

[DF*] **Example:**
Let $T_1 : P_2 \to P_3$ and $T_2 : P_3 \to P_3$ be the linear transformations given by the formulas

$$T_1(p(x)) = xp(x)$$

and

$$T_2(p(x)) = p(x+1)$$

(a) Find formulas for $T_1^{-1}(p(x)), T_2^{-1}(p(x))$, and $(T_2 \circ T_1)^{-1}(p(x))$.

(b) Verify that $(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1}$.

**Solution:**

(a) Since $T_1(p(x)) = xp(x)$,
$$T_1^{-1}(p(x)) = \frac{1}{x}p(x)$$
Since $T_2(p(x)) = p(x+1)$,
$$T_2^{-1}(p(x)) = p(x-1)$$
so
$$\begin{aligned}(T_2 \circ T_1)^{-1}(p(x)) &= (T_1^{-1} \circ T_2^{-1})(p(x)) \\ &= T_1^{-1}(p(x-1)) \\ &= \frac{1}{x}p(x-1)\end{aligned}$$

(b) We will do the verification from the left hand side, since the right hand side already proven in part $(a)$
$$\begin{aligned}(T_2 \circ T_1)^{-1} &= (T_2(xp(x)))^{-1} \\ &= (xp(x+1))^{-1} \\ &= \frac{1}{x}p(x-1)\end{aligned}$$
it is proven then that $(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1}$.

[DF*] **Example:**
Let $T : P_1 \to \mathbb{R}^2$ be the function defined by the formula

$$T(p(x)) = (p(0), p(1))$$

(a) Find $T(2-x)$.

(b) Show that $T$ is a linear transformation.

(c) Show that $T$ is one-to-one.

(d) Find $T^{-1}(3,5)$, and sketch its graph.

**Solution:**

(a)

$$\begin{aligned} T(2-x) &= (2-(0), 2-(1)) \\ &= (2,1) \end{aligned}$$

(b)

$$\begin{aligned} T(p(x)+q(x)) &= (p(0)+q(0), p(1)+q(1)) \\ &= (p(0), p(1)) + (q(0), q(1)) \\ &= T(p(x)) + T(q(x)) \end{aligned}$$

and for any scalar $k$,

$$\begin{aligned} T(kp(x)) &= (kp(0), kp(1)) \\ &= k(p(0), p(1)) \\ &= kT(p(x)) \end{aligned}$$

so $T$ is a linear transformation.

(c) Let $p(x) = a_0 + a_1 x$, then

$$T(p(x)) = (a_0, a_0 + a_1)$$

so if

$$T(p(x)) = (0,0)$$

then $a_0 = a_1 = 0$ and $p$ is the zero polynomial, so $\ker(T) = \{\mathbf{0}\}$.

(d) Since $T(p(x)) = (a_0, a_0 + a_1)$, then $T^{-1}(3,5)$ has $a_0 = 3$ and $a_0 + a_1 = 5$ or $a_1 = 2$.

Thus,

$$T^{-1}(3,5) = 3 + 2x$$

[DF*] **Example:**
Determine whether $T_1 \circ T_2 = T_2 \circ T_1$

(a) $T_1 : \mathbb{R}^2 \to \mathbb{R}^2$ is the reflection about the $x$-axis, and $T_2 : \mathbb{R}^2 \to \mathbb{R}^2$ is the reflection about the $y$-axis.

(b) $T_1 : \mathbb{R}^2 \to \mathbb{R}^2$ is the orthogonal projection on the $x$-axis, and $T_2 : \mathbb{R}^2 \to \mathbb{R}^2$ is the counterclockwise rotation through an angle $\theta$.

(c) $T_1 : \mathbb{R}^3 \to \mathbb{R}^3$ is a dilation by a factor $k$, and $T_2 : \mathbb{R}^3 \to \mathbb{R}^3$ is the counterclockwise rotation about the $z$-axis through an angle $\theta$.

**Solution:**

(a) $T_1 \circ T_2 = T_2 \circ T_1$

(b) $T_1 \circ T_2 \neq T_2 \circ T_1$

(c) $T_1 \circ T_2 = T_2 \circ T_1$

[DF*] **Example:**
The Fundamental Theorem of Calculus implies that integration and differentiation reverse the actions of each other. Define a transformation $D : P_n \to P_{n-1}$ by $D(p(x)) = p'(x)$, and define $J : P_{n-1} \to P_n$ by

$$J(p(x)) = \int_0^x p(t)\, dt$$

(a) Show that $D$ and $J$ are linear transformations.

(b) Explain why $J$ is not the inverse transformation of $D$.

(c) Can the domains and/or codomains of $D$ and $J$ be restricted so they are inverse linear transformations?

**Solution:**

(a) If

$$\boldsymbol{p} = p(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$$

and

$$\boldsymbol{q} = q(x) = d_0 + d_1x + d_2x^2 + \cdots + d_nx^n$$

are distinct polynomials, then they differ in at least one coefficient. Thus $D : P_n \to P_{n-1}$ is given by

$$D(\boldsymbol{p}) = D(p(x)) = p'(x)$$

The range of $D$ is not all of $P_{n+1}$ but rather the subspace of $P_{n+1}$, with the formula for $D$:

$$D(c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_nx^n) = c_1 + 2c_2x + 3c_3x^2 + \cdots + nc_nx^{n-1}$$

We will show that $D$ follows the two properties of Homogeneity and Additivity

$$\begin{aligned} D(kp(x)) &= D(k(c_0 + c_1x + c_2x^2 + \cdots + c_nx^n)) \\ &= kc_1 + 2kc_2x + \cdots + nkc_nx^{n-1} \\ &= k(c_1 + 2c_2x + 3c_3x^2 + \cdots + nc_nx^{n-1}) \\ &= kD(p(x)) \end{aligned}$$

$$\begin{aligned} D(p(x) + q(x)) &= D((c_0 + c_1x + c_2x^2 + \cdots + c_nx^n) + (d_0 + d_1x + d_2x^2 + \cdots + d_nx^n)) \\ &= (c_1 + 2c_2x + \cdots + nc_nx^{n-1}) + (d_1 + 2d_2x + \cdots + nd_nx^{n-1}) \\ &= D(p(x)) + D(q(x)) \end{aligned}$$

thus $D$ is a linear transformation.

$D$ is one-to-one (since it maps distinct polynomials $\boldsymbol{p}$ and $\boldsymbol{q}$ into distinct polynomials $D(\boldsymbol{p})$ and $D(\boldsymbol{q})$); thus, $D$ has an inverse. But it is not proven yet that the inverse of $D$ is $J$.

The formula of $J$:

$$J(p(x)) = \int_0^x p(t)\, dt$$

$$J(c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_nx^n) = c_0x + \frac{c_1}{2}x^2 + \frac{c_2}{3}x^3 + \frac{c_3}{4}x^4 + \cdots + \frac{c_n}{n+1}x^{n+1}$$

We will show that $J$ follows the two properties of Homogeneity and Additivity

$$\begin{aligned} J(kp(x)) &= J(k(c_0 + c_1x + c_2x^2 + \cdots + c_nx^n)) \\ &= c_0kx + \frac{c_1}{2}kx^2 + \frac{c_2}{3}kx^3 + \frac{c_3}{4}kx^4 + \cdots + \frac{c_n}{n+1}kx^{n+1} \\ &= k\left(c_0x + \frac{c_1}{2}x^2 + \frac{c_2}{3}x^3 + \frac{c_3}{4}x^4 + \cdots + \frac{c_n}{n+1}x^{n+1}\right) \\ &= kJ(p(x)) \end{aligned}$$

$$\begin{aligned}
J(p(x)+q(x)) &= J((c_0+c_1x+c_2x^2+\cdots+c_nx^n)+(d_0+d_1x+d_2x^2+\cdots+d_nx^n)) \\
&= \left(c_0x+\frac{c_1}{2}x^2+\cdots+\frac{c_n}{n+1}x^{n+1}\right)+\left(d_0x+\frac{d_1}{2}x^2+\cdots+\frac{d_n}{n+1}x^{n+1}\right) \\
&= J(p(x))+J(q(x))
\end{aligned}$$

thus $J$ is a linear transformation.

$J$ is one-to-one (since it maps distinct polynomials $\boldsymbol{p}$ and $\boldsymbol{q}$ into distinct polynomials $J(\boldsymbol{p})$ and $J(\boldsymbol{q})$); thus, $J$ has an inverse. But it is not proven yet that the inverse of $J$ is $D$, or vice versa.

(b) We will see whether $J$ is the inverse transformation of $D$ or not , we know that

$$D(\boldsymbol{p}) = D(p(x)) = p'(x)$$

then if we take $p'(x)$ as the domain for the linear transformation $J$, we will obtain

$$\begin{aligned}
J(p'(x)) &= \int_0^x p'(t)\,dt \\
J(c_1+2c_2x+3c_3x^2+\cdots+nc_nx^{n-1}) &= c_1x+2\frac{c_2}{2}x^2+3\frac{c_3}{3}x^3+4\frac{c_4}{4}x^4+\cdots+n\frac{c_n}{n}x^n \\
&= c_1x+c_2x^2+c_3x^3+c_4x^4+\cdots+c_nx^n
\end{aligned}$$

thus $J$ is not the inverse transformation of $D$ since $J(p'(x)) \neq p(x)$.

(c) The domain of $D$ if restricted to $\{P_0\}$ (polynomial with only constant term) and the codomain of $J$ if restricted to $\{0\}$ then $D$ and $J$ will be inverse linear transformations.

[DF*] A general linear transformation from any $n$-dimensional vector space $V$ to any $m$-dimensional vector space $W$ can be performed using an appropriate matrix transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$.

[DF*] Suppose that $V$ is an $n$-dimensional vector space, $W$ is an $m$-dimensional vector space, and that $T : V \to W$ is a linear transformation. Suppose further that $B$ is a basis for $V$, and that $B'$ is a basis for $W$, and that for each vector $\boldsymbol{x}$ in $V$, the coordinate matrices for $\boldsymbol{x}$ and $T(\boldsymbol{x})$ are $[\boldsymbol{x}]_B$ and $[T(\boldsymbol{x})]_{B'}$, respectively.

It will be our goal to find an $m \times n$ matrix $A$ such that multiplication by $A$ maps the vector $[\boldsymbol{x}]_B$ into the vector $[T(\boldsymbol{x})]_{B'}$ for each $\boldsymbol{x}$ in $V$.

If we can do so, then we will be able to execute the linear transformation $T$ by using matrix multiplication and the following indirect procedure:

**Finding $T(\boldsymbol{x})$ Indirectly**

1. Compute the coordinate vector $[\boldsymbol{x}]_B$.
2. Multiply $[\boldsymbol{x}]_B$ on the left by $A$ to produce $[T(\boldsymbol{x})]_{B'}$.
3. Reconstruct $T(\boldsymbol{x})$ from its coordinate vector $[T(\boldsymbol{x})]_{B'}$.

The key to executing this plan is to find an $m \times n$ matrix $A$ with the property that

$$A[\boldsymbol{x}]_B = [T(\boldsymbol{x})]_{B'} \tag{23.253}$$

For this purpose, let $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ be a basis for the $n$-dimensional space $V$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_m\}$ a basis for the $m$-dimensional space $W$. Since the equation (23.253) must hold for all vectors in $V$, it must hold, in particular, for the basis vectors in $B$; that is,

$$A[\boldsymbol{u}_1]_B = [T(\boldsymbol{u}_1)]_{B'}, \quad A[\boldsymbol{u}_2]_B = [T(\boldsymbol{u}_2)]_{B'}, \ldots, \quad A[\boldsymbol{u}_n]_B = [T(\boldsymbol{u}_n)]_{B'} \tag{23.254}$$

But

$$[\boldsymbol{u}_1]_B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \quad [\boldsymbol{u}_2]_B = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \ldots, \quad [\boldsymbol{u}_n]_B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

so

$$A[\boldsymbol{u}_1]_B = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}$$

$$A[\boldsymbol{u}_2]_B = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}$$

$$\vdots \quad \vdots \quad \vdots$$

$$A[\boldsymbol{u}_n]_B = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

Substituting these results into () yields

$$\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} = [T(\boldsymbol{u}_1)]_{B'}, \quad \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} = [T(\boldsymbol{u}_2)]_{B'}, \ldots, \quad \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = [T(\boldsymbol{u}_n)]_{B'}$$

which shows that the successive columns of $A$ are the coordinate vectors of

$$T(\boldsymbol{u}_1), T(\boldsymbol{u}_2), \ldots, T(\boldsymbol{u}_n)$$

with respect to the basis $B'$. Thus, the matrix $A$ that completes the link is

$$A = [[T(\boldsymbol{u}_1)]_{B'} | [T(\boldsymbol{u}_2)]_{B'} | \ldots | [T(\boldsymbol{u}_n)]_{B'}] \tag{23.255}$$

We will call this the matrix for $T$ relative to the bases $B$ and $B'$ and we will denote it by the symbol $[T]_{B',B}$. Using this notation, Formula (23.255) can be written as

$$[T]_{B',B} = [[T(\boldsymbol{u}_1)]_{B'}|[T(\boldsymbol{u}_2)]_{B'}|\ldots|[T(\boldsymbol{u}_n)]_{B'}] \tag{23.256}$$

and from (23.253), this matrix has the property

$$[T]_{B',B}[\boldsymbol{x}]_B = [T(\boldsymbol{x})] = [T(\boldsymbol{x})]_{B'} \tag{23.257}$$

In the special case where $T_A : \mathbb{R}^n \to \mathbb{R}^m$ is multiplication by $A$, and where $B$ and $B'$ are the standard bases for $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively, then

$$[T]_{B',B} = A \tag{23.258}$$

**Remark**
Observe that in the notation $[T]_{B',B}$ the right subscript is a basis for the domain of $T$, and the left subscript is a basis for the image space of $T$.

[DF*] **Example:**
Let $T : P_1 \to P_2$ be the linear transformation defined by

$$T(p(x)) = xp(x)$$

Find the matrix for $T$ with respect to the standard bases

$$B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$$
$$B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$$

where

$$\boldsymbol{u}_1 = 1, \quad \boldsymbol{u}_2 = x$$
$$\boldsymbol{v}_1 = 1, \quad \boldsymbol{v}_2 = x, \quad \boldsymbol{v}_3 = x^2$$

**Solution:** From the given formula for $T$ we obtain

$$T(\boldsymbol{u}_1) = T(1) = (x)(1) = x$$
$$T(\boldsymbol{u}_2) = T(x) = (x)(x) = x^2$$

By inspection, the coordinate vectors for $T(\boldsymbol{u}_1)$ and $T(\boldsymbol{u}_2)$ relative to $B'$ are

$$[T(\boldsymbol{u}_1)]_{B'} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad [T(\boldsymbol{u}_2)]_{B'} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, the matrix for $T$ with respect to $B$ and $B'$ is

$$[T]_{B',B} = [[T(\boldsymbol{u}_1)]_{B'}|[T(\boldsymbol{u}_2)]_{B'}] = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

[DF*] In the special case where $V = W$ (so that $T : V \to V$ is a linear operator), it is usual to take $B = B'$ when constructing a matrix for $T$. In this case the resulting amtrix is called the amtrix for $T$ relative to the basis $B$ and is usually denoted by $[T]_B$ rather than $[T]_{B,B}$. If $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$, then Formulas (23.256) and (23.257) become

$$[T]_B = [[T(\boldsymbol{u}_1)]_B|[T(\boldsymbol{u}_2)]_B|\ldots|[T(\boldsymbol{u}_n)]_B] \tag{23.259}$$

$$[T]_B[\boldsymbol{x}]_B = [T(\boldsymbol{x})]_B \tag{23.260}$$

In the special case where $T : \mathbb{R}^n \to \mathbb{R}^n$ is a matrix operator, say multiplication by $A$, and $B$ is the standard basis for $\mathbb{R}^n$, then Formula (23.259) simplifies to

$$[T]_B = A \tag{23.261}$$

[DF*] Recall that identity operator $T : V \to V$ maps every vector in $V$ into itself, that is, $I(\boldsymbol{x}) = \boldsymbol{x}$ for every vector $\boldsymbol{x}$ in $V$.

If $V$ is $n$-dimensional, then the matrix for $I$ relative to any basis $B$ for $V$ is the $n \times n$ identity matrix.

[DF*] If $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ is a basis for a finite-dimensional vector space $V$, and if $I : V \to V$ is the identity operator on $V$, then

$$I(\boldsymbol{u}_1) = \boldsymbol{u}_1, \quad I(\boldsymbol{u}_2) = \boldsymbol{u}_2, \ldots, \quad I(\boldsymbol{u}_n) = \boldsymbol{u}_n$$

Therefore,

$$[I]_B = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & \ldots & 1 \end{bmatrix}$$

with $[I(\boldsymbol{u}_i)]_B$ is a column vector with entry of 1 at row $i$-th and zero elsewhere.

**Theorem 23.146: Matrices of Compositions Transformation**

If $T_1 : U \to V$ and $T_2 : V \to W$ are linear transformations, and if $B$, and $B''$, and $B'$ are bases for $U, V$, and $W$, respectively, then

$$[T_2 \circ T_1]_{B',B} = [T_2]_{B',B''}[T_1]_{B'',B} \tag{23.262}$$

**Theorem 23.147: Matrices of Inverse Transformation**

If $T : V \to V$ is a linear operator, and if $B$ is a basis for $V$, then the following are equivalent.

(a) $T$ is one-to-one.

(b) $[T]_B$ is invertible.

Moreover, when these equivalent conditions hold,

$$[T^{-1}]_B = [T]_B^{-1} \tag{23.263}$$

**Remark**

In (23.262), observe how the interior subscript $B''$ (the basis for the intermediate space $V$) seems to "cancel out," leaving only the bases for the domain and image space of the composition as subscripts. This cancellation of interior subscripts suggests the following extension of Formula (23.262) to compositions of three linear transformations:

$$[T_3 \circ T_2 \circ T_1]_{B',B} = [T_3]_{B',B'''}[T_2]_{B''',B''}[T_1]_{B'',B} \tag{23.264}$$

[DF*] **Example:**
Let $T : P_2 \to P_3$ be the linear transformation defined by $T(p(x)) = xp(x)$.

(a) Find the matrix for $T$ relative to the standard bases

$$B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$$

and

$$B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{v}_4\}$$

where

$$\boldsymbol{u}_1 = 1, \quad \boldsymbol{u}_2 = x, \quad \boldsymbol{u}_3 = x^2$$
$$\boldsymbol{v}_1 = 1, \quad \boldsymbol{v}_2 = x, \quad \boldsymbol{v}_3 = x^2, \quad \boldsymbol{v}_4 = x^3$$

(b) Verify that the matrix $[T]_{B',B}$ obtained in part (a) satisfies Formula(5) for every vector $\boldsymbol{x} = c_0 + c_1 x + c_2 x^2$ in $P_2$

**Solution:**

(a) From the given formula for $T$ we obtain

$$T(\boldsymbol{u}_1) = T(1) = (x)(1) = x$$
$$T(\boldsymbol{u}_2) = T(x) = (x)(x) = x^2$$
$$T(\boldsymbol{u}_3) = T(x^2) = (x)(x^2) = x^3$$

By inspection, the coordinate vectors for $T(\boldsymbol{u}_1), T(\boldsymbol{u}_2)$, and $T(\boldsymbol{u}_3)$ relative to $B'$ are

$$[T(\boldsymbol{u}_1)]_{B'} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad [T(\boldsymbol{u}_2)]_{B'} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad [T(\boldsymbol{u}_3)]_{B'} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, the matrix for $T$ with respect to $B$ and $B'$ is

$$[T]_{B',B} = [[T(\boldsymbol{u}_1)]_{B'} | [T(\boldsymbol{u}_2)]_{B'} | [T(\boldsymbol{u}_3)]_{B'}] = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(b)

$$[T]_{B',B}[\boldsymbol{x}]_B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$[T(\boldsymbol{x})]_{B'} = [c_0 x + c_1 x^2 + c_2 x^3]_{B'} = \begin{bmatrix} 0 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

[DF*] **Example:**
Let $T : P_2 \to P_2$ be the linear operator defined by

$$T(a_0 + a_1 x + a_2 x^2) = a_0 + a_1(x-1) + a_2(x-1)^2$$

(a) Find the matrix for $T$ relative to the standard basis $B = \{1, x, x^2\}$ for $P_2$.

(b) Verify that the matrix $[T]_B$ obtained in part (a) satisfies Formula (23.257) for every vector $\boldsymbol{x} = a_0 + a_1 x + a_2 x^2$ in $P_2$.

**Solution:**

(a) We will have

$$\begin{aligned} T(1) &= 1 \\ T(x) &= x - 1 = -1 + x \\ T(x^2) &= (x-1)^2 \end{aligned}$$

Thus the matrix for $T$ relative to $B$ is

$$\begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

(b) The steps to verify are

$$[T]_B[\boldsymbol{x}]_B = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_0 - a_1 + a_2 \\ a_1 - 2a_2 \\ a_2 \end{bmatrix}$$

For $\boldsymbol{x} = a_0 + a_1 x + a_2 x^2$,

$$\begin{aligned} T(\boldsymbol{x}) &= a_0 + a_1(x-1) + a_2(x-1)^2 \\ &= a_0 - a_1 + a_2 + (a_1 - 2a_2)x + a_2 x^2 \end{aligned}$$

so

$$[T(\boldsymbol{x})]_B = \begin{bmatrix} a_0 - a_1 + a_2 \\ a_1 - 2a_2 \\ a_2 \end{bmatrix}$$

[DF*] **Example:**
Let $T : \mathbb{R}^2 \to \mathbb{R}^3$ be defined by

$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} -x_2 \\ 0 \\ 2x_1 + x_2 \end{bmatrix}$$

Find the matrix $[T]_{B',B}$ relative to the bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}, \quad \boldsymbol{v}_3 = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$$

**Solution:** First we will compute the mapping

$$T(\boldsymbol{u}_1) = T\left(\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} -2 \\ 0 \\ 4 \end{bmatrix} = 4\boldsymbol{v}_1 - 2\boldsymbol{v}_2 - \boldsymbol{v}_3$$

$$T(\boldsymbol{u}_2) = T\left(\begin{bmatrix} 2 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix} = 3\boldsymbol{v}_1 - \boldsymbol{v}_2$$

Expressing these vectors as linear combinations of $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$, we obtain

$$T(\boldsymbol{u}_1) = 4\boldsymbol{v}_1 - 2\boldsymbol{v}_2 - \frac{2}{3}\boldsymbol{v}_3$$
$$T(\boldsymbol{u}_2) = 3\boldsymbol{v}_1 - \frac{3}{2}\boldsymbol{v}_2 + \frac{1}{3}\boldsymbol{v}_3$$

Thus,

$$[T(\boldsymbol{u}_1)]_{B'} = \begin{bmatrix} 4 \\ -2 \\ -\frac{2}{3} \end{bmatrix}, \quad [T(\boldsymbol{u}_2)]_{B'} = \begin{bmatrix} 3 \\ -\frac{3}{2} \\ \frac{1}{3} \end{bmatrix}$$

so

$$[T]_{B',B} = [[T(\boldsymbol{u}_1)]_{B'} | [T(\boldsymbol{u}_2)]_{B'}] = \begin{bmatrix} 4 & 3 \\ -2 & -\frac{3}{2} \\ -\frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

```
Old basis B = {u1, u2}
u1:
   1.0000
   2.0000

u2:
   2.0000
  -1.0000

New basis B' = {v1, v2, v3}
v1:
   1.0000
   1.0000
   1.0000

v2:
   2.0000
   2.0000
        0

v3:
   3.0000
        0
        0

T(u1):
  -2.0000
        0
   4.0000

T(u2):
   1.0000
        0
   3.0000

Matrix from new basis B':
   1.0000   2.0000   3.0000
   1.0000   2.0000        0
   1.0000        0        0
```

```
xterm
Matrix from new basis B':
   1.0000   2.0000   3.0000
   1.0000   2.0000        0
   1.0000        0        0

Expressing T(u1) and T(u2) as the linear combinations of v1, v2, and v3:
[T(u1)]_{B'}:
   4.0000
  -2.0000
  -0.6667

[T(u2)]_{B'}:
   3.0000
  -1.5000
   0.3333

[T]:

        0  -1.0000
        0        0
   2.0000   1.0000
[T]_{B',B}:

   4.0000   3.0000
  -2.0000  -1.5000
  -0.6667   0.3333

 B' * Tu1:
  -2.0000
        0
   4.0000


 B' * Tu2:
   1.0000
        0
   3.0000
```

**Figure 23.216:** *The computation to compute the matrix $[T]_{B',B}$ relative to the bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Linear Transformation from R2 to R3 with Armadillo and GiNaC Exercise Number 5/main.cpp**).*

[DF*] **Example:**
Let $T : P_2 \to P_2$ be the linear operator defined by $T(p(x)) = p(2x+1)$, that is,

$$T(c_0 + c_1 x + c_2 x^2) = c_0 + c_1(2x+1) + c_2(2x+1)^2$$

(a) Find $[T]_B$ with respect to the basis $B = \{1, x, x^2\}$.
(b) Use the three-step procedure to compute $T(2 - 3x + 4x^2)$.
(c) Check the result obtained in part (b) by computing $T(2 - 3x + 4x^2)$ directly.

**Solution:**

(a)

$$\begin{aligned} T(1) &= 1 \\ T(x) &= 2x + 1 = 1 + 2x \\ T(x^2) &= (2x+1)^2 = 1 + 4x + 4x^2 \end{aligned}$$

thus

$$[T]_B = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 4 \\ 0 & 0 & 4 \end{bmatrix}$$

(b) The matrix for $2 - 3x + 4x^2$ is

$$\begin{bmatrix} 2 \\ -3 \\ 4 \end{bmatrix}$$

so

$$[T]_B \begin{bmatrix} 2 \\ -3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 4 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ -3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 10 \\ 16 \end{bmatrix}$$

Thus

$$T(2 - 3x + 4x^2) = 3 + 10x + 16x^2$$

(c) If we check the result directly, we will obtain

$$\begin{aligned} T(2 - 3x + 4x^2) &= 2 - 3(2x+1) + 4(2x+1)^2 \\ &= 2 - 6x - 3 + 16x^2 + 16x + 4 \\ &= 3 + 10x + 16x^2 \end{aligned}$$

[DF*] **Example:**
Let

$$v_1 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

and let

$$A = \begin{bmatrix} 3 & 2 \\ -2 & 1 \end{bmatrix}$$

be the matrix for $T : \mathbb{R}^2 \to \mathbb{R}^2$ relative to the basis $B = \{v_1, v_2\}$.

1. Find $[T(v_1)]_B$ and $[T(v_2)]_B$.
2. Find $T(v_1)$ and $T(v_2)$.

3. Find a formula for
$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right)$$
4. Use the formula obtained in part (c) to compute
$$T\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right)$$

**Solution:**

1. Since $A$ is the matrix for $T$ relative to $B$,
$$A = [[T(\boldsymbol{v}_1)]_B | [T(\boldsymbol{v}_2)]_B]$$
That is,
$$[T(\boldsymbol{v}_1)]_B = \begin{bmatrix} 3 \\ -2 \end{bmatrix}, \quad [T(\boldsymbol{v}_1)]_B = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$
2. Since
$$[T(\boldsymbol{v}_1)]_B = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$
then
$$T(\boldsymbol{v}_1) = 3\boldsymbol{v}_1 - 2\boldsymbol{v}_2 = \begin{bmatrix} 6 \\ -3 \end{bmatrix} - \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \end{bmatrix}$$
Similarly,
$$[T(\boldsymbol{v}_2)]_B = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$
then
$$T(\boldsymbol{v}_1) = 2\boldsymbol{v}_1 + \boldsymbol{v}_2 = \begin{bmatrix} 4 \\ -2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ -3 \end{bmatrix}$$
3. If
$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 = c_1 \begin{bmatrix} 2 \\ -1 \end{bmatrix} + c_2 \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$
then
$$\begin{aligned} x_1 &= 2c_1 + c_2 \\ x_2 &= -c_1 - c_2 \end{aligned}$$
Solving for $c_1$ and $c_2$ gives
$$\begin{aligned} c_1 &= x_1 + x_2 \\ c_2 &= -x_1 - 2x_2 \end{aligned}$$
so
$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = c_1T(\boldsymbol{v}_1) + c_2T(\boldsymbol{v}_2) = (x_1 + x_2)\begin{bmatrix} 4 \\ -1 \end{bmatrix} + (-x_1 - 2x_2)\begin{bmatrix} 5 \\ -3 \end{bmatrix}$$
$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} -x_1 - 6x_2 \\ 2x_1 + 5x_2 \end{bmatrix} = \begin{bmatrix} -1 & -6 \\ 2 & 5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

4. Use the formula obtained in part (c) to compute

$$T\left(\begin{bmatrix}1\\1\end{bmatrix}\right) = \begin{bmatrix}-1 & -6\\2 & 5\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}-7\\7\end{bmatrix}$$

[DF*] **Example:**
Let

$$A = \begin{bmatrix}2 & 0 & -1\\1 & 3 & 5\\0 & -2 & 6\end{bmatrix}$$

be the matrix for $T : P_2 \to P_2$ with respect to the basis $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$, where

$$\begin{aligned}\boldsymbol{v}_1 &= 2 - 2x^2\\ \boldsymbol{v}_2 &= 1 + 2x + 2x^2\\ \boldsymbol{v}_3 &= 3x + x^2\end{aligned}$$

(a) Find $[T(\boldsymbol{v}_1)]_B, [T(\boldsymbol{v}_2)]_B$, and $[T(\boldsymbol{v}_3)]_B$.
(b) Find $T(\boldsymbol{v}_1), T(\boldsymbol{v}_2)$, and $T(\boldsymbol{v}_3)$.
(c) Find a formula for $T(a_0 + a_1x + a_2x^2)$.
(d) Use the formula obtained in (c) to compute $T(1 + x^2)$.

**Solution:**

(a) Since $A$ is the matrix for $T$ relative to $B$, the columns of $A$ are $[T(\boldsymbol{v}_1)]_B, [T(\boldsymbol{v}_2)]_B$, and $[T(\boldsymbol{v}_3)]_B$, respectively. That is,

$$[T(\boldsymbol{v}_1)]_B = \begin{bmatrix}2\\1\\0\end{bmatrix}, \quad [T(\boldsymbol{v}_2)]_B = \begin{bmatrix}0\\3\\-2\end{bmatrix}$$

and

$$[T(\boldsymbol{v}_3)]_B = \begin{bmatrix}-1\\5\\6\end{bmatrix}$$

(b) Since

$$[T(\boldsymbol{v}_1)]_B = \begin{bmatrix}2\\1\\0\end{bmatrix}$$

then

$$\begin{aligned}T(\boldsymbol{v}_1) &= 2\boldsymbol{v}_1 + \boldsymbol{v}_2\\ &= 2(2 - 2x^2) + (1 + 2x + 2x^2)\\ &= 5 + 2x - 2x^2\end{aligned}$$

Similarly,

$$\begin{aligned}T(\boldsymbol{v}_2) &= 3\boldsymbol{v}_2 - 2\boldsymbol{v}_3\\ &= 3(1 + 2x + 2x^2) - 2(3x + x^2)\\ &= 3 + 4x^2\end{aligned}$$

and

$$\begin{aligned} T(\boldsymbol{v}_3) &= -\boldsymbol{v}_1 + 5\boldsymbol{v}_2 + 6\boldsymbol{v}_3 \\ &= -(2-2x^2) + 5(1+2x+2x^2) + 6(3x+x^2) \\ &= 3 + 28x + 18x^2 \end{aligned}$$

(c) If

$$\begin{aligned} a_0 + a_1x + a_2x^2 &= c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 \\ &= c_1(2-2x^2) + c_2(1+2x+2x^2) + c_3(3x+x^2) \end{aligned}$$

Then we will have

$$\begin{aligned} a_0 &= 2c_1 + c_2 \\ a_1 &= 2c_2 + 3c_3 \\ a_2 &= -2c_1 + 2c_2 + c_3 \end{aligned}$$

Solving for $c_1, c_2$, and $c_3$ gives

$$\begin{aligned} c_1 &= \frac{8}{28}a_0 + \frac{2}{28}a_1 - \frac{6}{28}a_2 \\ c_2 &= \frac{6}{14}a_0 - \frac{2}{14}a_1 + \frac{6}{14}a_2 \\ c_3 &= -\frac{2}{7}a_0 + \frac{3}{7}a_1 - \frac{2}{7}a_2 \end{aligned}$$

so

$$\begin{aligned} T(a_0 + a_1x + a_2x^2) &= c_1T(\boldsymbol{v}_1) + c_2T(\boldsymbol{v}_2) + c_3T(\boldsymbol{v}_3) \\ &= \left(\frac{8}{28}a_0 + \frac{2}{28}a_1 - \frac{6}{28}a_2\right)(5+2x-2x^2) \\ &+ \left(\frac{6}{14}a_0 - \frac{2}{14}a_1 + \frac{6}{14}a_2\right)(3+4x^2) \\ &+ \left(-\frac{2}{7}a_0 + \frac{3}{7}a_1 - \frac{2}{7}a_2\right)(3+28x+18x^2) \end{aligned}$$

(d)

$$T(1+x^2) = 1.21428 - 4.0x - 7.0x^2$$

[DF*] **Example:**
Let $T_1 : P_1 \to P_2$ be the linear transformation defined by

$$T_1(c_0 + c_1x) = 2c_0 - 3c_1x$$

and let $T_2 : P_2 \to P_3$ be the linear transformation defined by

$$T_2(c_0 + c_1x + c_2x^2) = 3c_0x + 3c_1x^2 + 3c_2x^3$$

Let $B = \{1, x\}$, $B'' = \{1, x, x^2\}$, and $B' = \{1, x, x^2, x^3\}$.

(a) Find $[T_2 \circ T_1]_{B',B}$, $[T_2]_{B',B}$, and $[T_1]_{B'',B}$.

(b) State a formula relating the matrices in part (a).

```
B:
[ 2  1  0 a0]
[ 0  2  3 a1]
[-2  2  1 a2]


B (in reduced row form) :
[      2            1            0           a0     ]
[      0            2            3           a1     ]
[      0            0          -7/2     a2+a0-3/2*a1]

B (in row reduced echelon form) :
[          1                    1/2                     0                   1/2*a0       ]
[          0                     1                     3/2                  1/2*a1       ]
[          0                     0                      1           -2/7*a2-2/7*a0+3/7*a1]
```

**Figure 23.217:** *The computation for part (c), the symbolic Gaussian elimination part. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Exercise Number 11(c) Chapter 8.4 Part 1 Gaussian Elimination for Symbolic Matrix with SymbolicC++/main.cpp**).*

```
c_{1} = (0.071428575)*a1-(0.21428572)*a2+(0.2857143)*a0

c_{2} = -(0.14285715)*a1+(0.42857143)*a2+(0.42857143)*a0

c_{3} = (0.42857143)*a1-(0.2857143)*a2-(0.2857143)*a0

T(v_{1}) = 5-2*x^2+2*x

T(v_{2}) = 3+4*x^2

T(v_{3}) = 3+18*x^2+28*x

T(a_{0} + a_{1} x + a_{2} x^2) = -(8.428572)*a2*x+(7.0)*a1*x^2-(7.428571)*x*a0+(1.2142856)*a1+(12.142858)*a1*x-(3.0000005)*a2*x^
2-(0.6428572)*a2-(4.0000005)*x^2*a0+(1.8571428)*a0

For T(a_{0} + a_{1} x + a_{2} x^2) :

The x^0-coefficient is (1.2142856)*a1-(0.6428572)*a2+(1.8571428)*a0

The x^1-coefficient is (12.142858)*a1-(8.428572)*a2-(7.428571)*a0

The x^2-coefficient is (7.0)*a1-(3.0000005)*a2-(4.0000005)*a0

T(1 + x^2) = 1.2142856-(7.000001)*x^2-(15.857143)*x
```

**Figure 23.218:** *The computation for part (c), the symbolic polynomial multiplication part. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Exercise Number 11(c) Chapter 8.4 Part 2 Symbolic Polynomial Multiplication with GiNaC/main.cpp**).*

(c) Verify that the matrices in part (a) satisfy the formula you stated in part (b).

**Solution:**

(a) We will compute one by one

$$\begin{aligned} T_1(1) &= 2 \\ T_1(x) &= -3x \\ T_2(1) &= 3x \\ T_2(x) &= 3x^2 \\ T_2(x^2) &= 3x^3 \end{aligned}$$

then we will have

$$\begin{aligned} (T_2 \circ T_1)(1) &= T_2(2) = 6x \\ (T_2 \circ T_1)(x) &= T_2(-3x) = -9x^2 \end{aligned}$$

Thus,

$$[T_2 \circ T_1]_{B',B} = \begin{matrix} 00 \\ 60 \\ 0-9 \\ 00 \end{matrix}$$

and

$$[T_1]_{B'',B} = \begin{bmatrix} 2 & 0 \\ 0 & -3 \\ 0 & 0 \end{bmatrix}$$

$$[T_2]_{B',B''} = \begin{bmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

remember that the number of column represents the number of basis from the old basis, and the number of row represents the number of basis from the new basis. This is from the last one

$$[T_2]_{B',B''} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$$

(b) The formula relating the matrices in part (a) is

$$[T_2 \circ T_1]_{B',B} = [T_2]_{B',B''}[T_1]_{B'',B}$$

(c) The verification that matrices in part (a) satisfy the formula in part (b):

$$\begin{bmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -3 \\ 0 & 0 \end{bmatrix} = \begin{matrix} 00 \\ 60 \\ 0-9 \\ 00 \end{matrix}$$

[DF*] **Example:**
Show that if $T : V \to V$ is a contraction or a dilation of $V$, then the matrix for $T$ relative to any basis for $V$ is a positive scalar multiple of the identity matrix.
**Solution:**
If $T$ is a contraction or dilation of $V$, then $T$ maps any basis $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ of $V$ to $\{k\boldsymbol{v}_1, k\boldsymbol{v}_2, \ldots, k\boldsymbol{v}_n\}$ where $k$ is a nonzero constant. Thus the matrix for $T$ relative to $B$ is

$$[T]_B = \begin{bmatrix} k & 0 & 0 & 0 & \ldots & 0 \\ 0 & k & 0 & 0 & \ldots & 0 \\ 0 & 0 & k & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 & k \end{bmatrix}$$

[DF*] **Example:**
Prove that if $B$ and $B'$ are the standard bases for $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively, then the matrix for a linear transformation $T : \mathbb{R}^n \to \mathbb{R}^m$ relative to the bases $B$ and $B'$ is the standard matrix for $T$.
**Solution:** The matrix for $T$ relative to $B$ is the matrix whose columns are the transforms of the basis vectors in $B$ in terms of the standard basis. Since $B$ is the standard basis for $\mathbb{R}^n$, this matrix is the standard matrix for $T$. Also, since $B'$ is the standard basis for $\mathbb{R}^m$, the resulting transformation will give vector components relative to the standard basis.

[DF*] **Example:**
In each part, suppose that $B = \{f_1, f_2, f_3\}$ is a basis for a subspace $V$ of the vector space of real-valued funcitons defined on the real line. Find the matrix with respect to $B$ for differentiation operator $D : V \to V$.

(a) $f_1 = 1, f_2 = \sin\ x, f_3 = \cos\ x$
(b) $f_1 = 1, f_2 = e^x, f_3 = e^{2x}$
(c) $f_1 = e^{2x}, f_2 = xe^{2x}, f_3 = x^2e^{2x}$
(d) Use the matrix in part (c) to compute $D(3e^{2x} - 4xe^{2x} + 8x^2e^{2x})$.

**Solution:**

(a)

$$\begin{aligned} D(f_1) &= D(1) = 0 \\ D(f_2) &= D(\sin\ x) = \cos\ x \\ D(f_3) &= D(\cos\ x) = -\sin\ x \end{aligned}$$

The matrix for $D$ relative to this basis is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

(b)

$$\begin{aligned} D(f_1) &= D(1) = 0 \\ D(f_2) &= D(e^x) = e^x \\ D(f_3) &= D(e^{2x}) = 2e^{2x} \end{aligned}$$

The matrix for $D$ relative to this basis is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Basically the formula is like this:

$$\begin{bmatrix} D(1) \\ D(e^x) \\ D(e^{2x}) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ e^x \\ x^{2x} \end{bmatrix}$$

(c)

$$\begin{aligned} D(\boldsymbol{f}_1) &= D(e^{2x}) = 2e^{2x} \\ D(\boldsymbol{f}_2) &= D(xe^{2x}) = e^{2x} + 2xe^{2x} \\ D(\boldsymbol{f}_3) &= D(x^2e^{2x}) = 2xe^{2x} + 2x^2e^{2x} \end{aligned}$$

The matrix for $D$ relative to this basis is

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \end{bmatrix}$$

Basically the formula is like this:

$$\begin{bmatrix} D(e^{2x}) \\ D(xe^{2x}) \\ D(x^2e^{2x}) \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} e^{2x} \\ xe^{2x} \\ x^2e^{2x} \end{bmatrix}$$

(d)

$$D(3e^{2x} - 4xe^{2x} + 8x^2e^{2x}) = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ -4 \\ 8 \end{bmatrix} = \begin{bmatrix} 6 \\ -5 \\ 8 \end{bmatrix}$$

Thus, $D(3e^{2x} - 4xe^{2x} + 8x^2e^{2x}) = 6e^{2x} - 5xe^{2x} + 8x^2e^{2x}$.

[DF*] The matrix for a linear operator $T : V \to V$ depends on the basis selected for $V$. One of the fundamental problems of linear algebra is to choose a basis for $V$ that makes the matrix for $T$ as simple as possible-a diagonal or a triangular matrix, for example.

[DF*] Standard bases do not necessarily produce the simplest matrices for linear operators. For example, consider the matrix operator $T : \mathbb{R}^2 \to \mathbb{R}^2$ whose standard matrix is

$$[T] = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix} \tag{23.265}$$

and view $[T]$ as the matrix for $T$ relative to the standard basis $B = \{\boldsymbol{e}_1, \boldsymbol{e}_2\}$ for $\mathbb{R}^2$. Let us compare this to the matrix for $T$ relative to the basis $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2\}$ for $\mathbb{R}^2$ in which

$$\boldsymbol{u}'_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}'_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{23.266}$$

Since

$$T(\boldsymbol{u}_1') = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 2\boldsymbol{u}_1'$$

and

$$T(\boldsymbol{u}_2') = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix} = 3\boldsymbol{u}_2'$$

it follows that

$$[T(\boldsymbol{u}_1')]_{B'} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

and

$$[T(\boldsymbol{u}_2')]_{B'} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

so the amtrix for $T$ relative to the basis $B'$ is

$$[T]_{B'} = [T(\boldsymbol{u}_1')_{B'} | T(\boldsymbol{u}_2')_{B'}] = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

This matrix, being diagonal, has a simpler form than $[T]$ and conveys clearly that the operator $T$ scales $\boldsymbol{u}_1'$ by a factor of 2 and $\boldsymbol{u}_2'$ by a factor of 3, information that is not immediately evident from $[T]$.

The problem of finding a basis that produces the simplest possible matrix for a linear operator $T : V \to V$ can be attacked by first finding a matrix for $T$ relative to any basis, typically a standard basis, where applicable, and then changing the basis in a way that simplifies the matrix.

[DF*] Recall that if $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ and $B' = \{\boldsymbol{u}_1', \boldsymbol{u}_2', \ldots, \boldsymbol{u}_n'\}$ are bases for a vector space $V$, then the transition matrices from $B$ to $B'$ and from $B'$ to $B$ are

$$P_{B \to B'} = [[\boldsymbol{u}_1]_{B'} | [\boldsymbol{u}_2]_{B'} | \ldots | [\boldsymbol{u}_n]_{B'}] \tag{23.267}$$

$$P_{B' \to B} = \left[[\boldsymbol{u}_1']_B | [\boldsymbol{u}_2']_B | \ldots | [\boldsymbol{u}_n']_B\right] \tag{23.268}$$

where the matrices $P_{B \to B'}$ and $P_{B' \to B}$ are inverses of each other.

If $\boldsymbol{v}$ is any vector in $V$, then

$$P_{B \to B'}[\boldsymbol{v}]_B = [\boldsymbol{v}]_{B'} \tag{23.269}$$

$$P_{B' \to B}[\boldsymbol{v}]_{B'} = [\boldsymbol{v}]_B \tag{23.270}$$

**Theorem 23.148: Transition Matrices and Identity Operator**

If $B$ and $B'$ are bases for a finite-dimensional vector space $V$, and if $I : V \to V$ is the identity operator on $V$, then

$$P_{B \to B'} = [I]_{B',B}$$

and

$$P_{B' \to B} = [I]_{B,B'}$$

[DF*] The main problem in similarity:
If $B$ and $B'$ are two bases for a finite-dimensional vector space $V$, and if $T : V \to V$ is a linear operator, what relationship, if any, exists between the matrices $[T]_B$ and $[T]_{B'}$?

The answer can be obtained by considering the composition of three linear operators on $V$.

Let $\boldsymbol{v}$ be a vector in $V$. Then $\boldsymbol{v}$ is first mapped into itself by the identity operator, then $\boldsymbol{v}$ is mapped into $T(\boldsymbol{v})$ by $T$, and then $T(\boldsymbol{v})$ is mapped into itself by the identity operator.

All four vector spaces involved in the composition are the same (namely, $V$), but the bases for the sapces vary.

Since the starting vector is $\boldsymbol{v}$ and the final vector is $T(\boldsymbol{v})$, the composition produces the same result as applying $T$ directly; that is,

$$T = I \circ T \circ I \tag{23.271}$$

If the first and last vector spaces are assigned the basis $B'$ and the middle two spaces are assigned the basis $B$, then it follows that

$$[T]_{B',B'} = [I \circ T \circ I]_{B',B'} = [I]_{B',B}[T]_{B,B}[I]_{B,B'} \tag{23.272}$$

or, in simpler notation,

$$[T]_{B'} = [I]_{B',B}[T]_B[I]_{B,B'} \tag{23.273}$$

we can rewrite the formula as

$$[T]_{B'} = P_{B\to B'}[T]_B P_{B'\to B} \tag{23.274}$$

**Theorem 23.149: Changing Bases on Matrices of Linear Operators**

Let $T : V \to V$ be a linear operator on a finite-dimensional vector space $V$, and let $B$ and $B'$ be bases for $V$. Then

$$[T]_{B'} = P^{-1}[T]_B P \tag{23.275}$$

where $P = P_{B'\to B}$ and $P^{-1} = P_{B\to B'}$.

**Theorem 23.150: Similar Matrices**

Two matrices, $A$ and $B$, are similar if and only if they represent the same linear operator. Moreover, if $B = P^{-1}AP$, then $P$ is the transition matrix from the basis relative to matrix $B$ to the basis relative to matrix $A$.

[DF*] **Example:**
We know that the matrices

$$C = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix}, \quad D = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

represent the same linear operator $T : \mathbb{R}^2 \to \mathbb{R}^2$. Verify that these matrices are similar by finding a matrix $P$ for which $D = P^{-1}CP$.

**Solution:**
We need to find the transition matrix

$$P = P_{B' \to B} = \left[ [\boldsymbol{u}_1']_B | [\boldsymbol{u}_2']_B \right]$$

where $B' = \{\boldsymbol{u}_1', \boldsymbol{u}_2'\}$ is the basis for $\mathbb{R}^2$ and $B = \{\boldsymbol{e}_1, \boldsymbol{e}_2\}$ is the standard basis for $\mathbb{R}^2$. We see by inspection that

$$\begin{aligned} \boldsymbol{u}_1' &= \boldsymbol{e}_1 + \boldsymbol{e}_2 \\ \boldsymbol{u}_2' &= \boldsymbol{e}_1 + 2\boldsymbol{e}_2 \end{aligned}$$

from which it follows that

$$[\boldsymbol{u}_1']_B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad [\boldsymbol{u}_2']_B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Thus,

$$P = P_{B' \to B} = \left[ [\boldsymbol{u}_1']_B | [\boldsymbol{u}_2']_B \right] = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

We leave it for you to verify that

$$P^{-1} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

and hence that

$$D = P^{-1}CP$$

$$\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

[DF*] Recall that a property of a square matrix is called a similarity invariant if that property is shared by all similar matrices.

Two matrices are similar if and only if they represent the same linear operator $T : V \to V$, it follows that if $B$ and $B'$ are bases for $V$, then every similarity invariant property of $[T]_B$ is also a similarity invariant property of $[T]_{B'}$ for any other basis $B'$ for $V$. For example, for any two bases $B$ and $B'$ we must have

$$\det([T]_B) = \det([T]_{B'})$$

It follows from this equation that the value of the determinant depends on $T$, but not on the particular basis that is used to obtaite matrix for $T$. Thus, the determinant can be regarded as a property of the linear operator $T$; indeed, if $V$ is a finite-dimensional vector space, then we can define the determinant of the linear operator $T$ to be

$$\det(T) = \det([T]_B) \tag{23.276}$$

where $B$ is any basis for $V$.

[DF*] **Determinant of a Linear Operator**
At the beginning of this section we showed that the matrices

$$[T] = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix}, \quad [T]_{B'} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

represent the same linear operator relative to different bases, the first relative to the standard basis $B = \{\boldsymbol{e}_1, \boldsymbol{e}_2\}$ for $\mathbb{R}^2$ and the second relative to the basis $B' = \{\boldsymbol{u}'_1, \boldsymbol{u}'_2\}$ for which

$$\boldsymbol{u}'_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}'_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

This means that $[T]$ and $[T]_{B'}$ must be similar matrices and hence must have the same similarity invariant properties. In particular, they must have the same determinant.

$$\det[T] = \begin{vmatrix} 1 & 1 \\ -2 & 4 \end{vmatrix} = 6$$

and

$$\det[T]_{B'} = \begin{vmatrix} 2 & 0 \\ 0 & 3 \end{vmatrix} = 6$$

[DF*] **Example:**
Find the eigenvalues and bases for the eigenspaces of the linear operator $T : P_2 \to P_2$ defined by

$$T(a + bx + cx^2) = -2c + (a + 2b + c)x + (a + 3c)x^2$$

**Solution:**
We know that the matrix for $T$ with respect to the standard basis $B = \{1, x, x^2\}$ is

$$[T]_B = \begin{bmatrix} 0 & 0 & -2 \\ 1 & 2 & 1 \\ 1 & 0 & 3 \end{bmatrix}$$

The eigenvalues of $T$ are $\lambda = 1$ and $\lambda = 2$. The eigenspace of $[T]_B$ corresponding to $\lambda = 2$ has the basis $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

and the eigenspace of $[T]_B$ corresponding to $\lambda = 1$ has the basis $\{\boldsymbol{u}_3\}$, where

$$\boldsymbol{u}_3 = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

The matrices $\boldsymbol{u}_1, \boldsymbol{u}_2$, and $\boldsymbol{u}_3$ are the coordinate matrices relative to $B$ of

$$\begin{aligned} \boldsymbol{p}_1 &= -1 + x^2 \\ \boldsymbol{p}_2 &= x \\ \boldsymbol{p}_3 &= -2 + x + x^2 \end{aligned}$$

Thus, the eigenspace of $T$ corresponding to $\lambda = 2$ has the basis

$$\{\boldsymbol{p}_1, \boldsymbol{p}_2\} = \{-1 + x^2, x\}$$

and that corresponding to $\lambda = 1$ has the basis

$$\{\boldsymbol{p}_3\} = \{-2 + x + x^2\}$$

We can check with the given formula for $T$ to verify that

$$T(\boldsymbol{p}_1) = 2\boldsymbol{p}_1$$
$$T(\boldsymbol{p}_2) = 2\boldsymbol{p}_2$$
$$T(\boldsymbol{p}_3) = \boldsymbol{p}_3$$

[DF*] **Example:**
Let $\lambda$ be an eigenvalue of a linear operator $T : V \to V$. Prove that the eigenvectors of $T$ corresponding to $\lambda$ are the nonzero vectors in the kernel of $\lambda I - T$.

**Solution:**

If $\boldsymbol{v}$ is an eigenvector of $T$ corresponding to $\lambda$, then $\boldsymbol{v}$ is a nonzero vector which satisfies the equation

$$T(\boldsymbol{v}) = \lambda \boldsymbol{v}$$

or

$$(\lambda I - T)(\boldsymbol{v}) = \mathbf{0}$$

Thus $\boldsymbol{v}$ is in the kernel of $\lambda I - T$.

[DF*] **Example:**
Let $C$ and $D$ be $m \times n$ matrices, and let $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ be a basis for a vector space $V$. Show that if $C[\boldsymbol{x}]_B = D[\boldsymbol{x}]_B$ for all $\boldsymbol{x}$ in $V$, then $C = D$.

**Solution:**
Since $C[\boldsymbol{x}]_B = D[\boldsymbol{x}]_B$ for all $\boldsymbol{x}$ in $V$, then we can let $\boldsymbol{x} = \boldsymbol{v}_i$ for each of the basis vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ of $V$.

Since $[\boldsymbol{v}_i]_B = \boldsymbol{e}_i$ for each $i$ where $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$ is the standard basis for $\mathbb{R}^n$, this yields

$$C\boldsymbol{e}_i = D\boldsymbol{e}_i$$

for $i = 1, 2, \ldots, n$.

But $C\boldsymbol{e}_i$ and $D\boldsymbol{e}_i$ are the $i$th columns of $C$ and $D$, respectively. Since corresponding columns of $C$ and $D$ are all equal, $C = D$.

## i. C++ Computation: Computing with Images of Basis Vectors for $T : \mathbb{R}^3 \to \mathbb{R}^2$

Consider the basis $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ for $\mathbb{R}^3$, where

$$\boldsymbol{v}_1 = (1,1,1), \quad \boldsymbol{v}_2 = (1,1,0), \quad \boldsymbol{v}_3 = (1,0,0)$$

Let $T : \mathbb{R}^3 \to \mathbb{R}^2$ be the linear transformation for which

$$T(\boldsymbol{v}_1) = (1,0), \quad T(\boldsymbol{v}_2) = (2,-1), \quad T(\boldsymbol{v}_3) = (4,3)$$

Find a formula for $T(x_1, x_2, x_3)$, and then use that formula to compute $T(2,-3,5)$.

**Solution:**
We first need to express $\boldsymbol{x} = (x_1, x_2, x_3)$ as a linear combination of $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$. If we write

$$(x_1, x_2, x_3) = c_1(1,1,1) + c_2(1,1,0) + c_3(1,0,0)$$

then on equation corresponding components, we obtain

$$\begin{aligned} c_1 + c_2 + c_3 &= x_1 \\ c_1 + c_2 &= x_2 \\ c_1 &= x_3 \end{aligned}$$

which yields

$$\begin{aligned} c_1 &= x_3 \\ c_2 &= x_2 - x_3 \\ c_3 &= x_1 - x_2 \end{aligned}$$

so

$$\begin{aligned} (x_1, x_2, x_3) &= x_3(1,1,1) + (x_2 - x_3)(1,1,0) + (x_1 - x_2)(1,0,0) \\ &= x_3\boldsymbol{v}_1 + (x_2 - x_3)\boldsymbol{v}_2 + (x_1 - x_2)\boldsymbol{v}_3 \end{aligned}$$

Thus

$$\begin{aligned} T(x_1, x_2, x_3) &= x_3 T(\boldsymbol{v}_1) + (x_2 - x_3)T(\boldsymbol{v}_2) + (x_1 - x_2)T(\boldsymbol{v}_3) \\ &= x_3(1,0) + (x_2 - x_3)(2,-1) + (x_1 - x_2)(4,3) \\ &= (4x_1 - 2x_2 - x_3, 3x_1 - 4x_2 + x_3) \end{aligned}$$

From this formula we obtain

$$T(2,-3,5) = (9,23)$$

The C++ code to compute the solution for this problem will be using **GiNaC** because this library is very powerful for symbolic computation with matrix, and to substitute the variable values later on, they just come in handy.

```
// g++ main.cpp -o result -lginac -lcln
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
```

```
using namespace GiNaC;

int main()
{
        Digits = 5; // define maximum decimal digits
        symbol x1("x1"), x2("x2"), x3("x3"), c1("c1"), c2("c2"), c3("c3");

        // Store basis vectors as a 3x1 vector in GiNac
        matrix v1 = {{1,1,1}};
        matrix v2 = {{1,1,0}};
        matrix v3 = {{1,0,0}};

        matrix Tv1 = {{1,0}};
        matrix Tv2 = {{2,-1}};
        matrix Tv3 = {{4,3}};

        matrix A(3,3);
        A(0,0) = v1[0]; A(1,0) = v2[0]; A(2,0) = v3[0];
        A(0,1) = v1[1]; A(1,1) = v2[1]; A(2,1) = v3[1];
        A(0,2) = v1[2]; A(1,2) = v2[2]; A(2,2) = v3[2];

        ex c = lst_to_matrix(lst{lst{c1},lst{c2},lst{c3}});
        ex x = lst_to_matrix(lst{lst{x1},lst{x2},lst{x3}});
        ex d = (A*c).evalm();
        cout << "\nMatrix A consists of basis vector S ={v1, v2, v3} for R3"
            << endl;
        cout << "\n A = " << A << endl;
        cout << "\n c = " << c << endl;
        cout << "\n x = " << x << endl;
        cout << "\nLet T: R3 -> R2 be the linear transformation for which"
            << endl;
        cout << "\n T(v1) = " << Tv1 << endl;
        cout << "\n T(v2) = " << Tv2 << endl;
        cout << "\n T(v3) = " << Tv3 << endl;
        cout << "\n A*c = " << d << endl;

        lst eqns = {d[0] == x1, d[1] ==x2, d[2] == x3}; // list all
            equations need to be solved
        lst vars = {c1,c2,c3}; // list the variables we want to solve
        ex c1_ans = lsolve(eqns, vars)[0].rhs(); // accessing the right-
            hand side expression
        ex c2_ans = lsolve(eqns, vars)[1].rhs(); // accessing the right-
            hand side expression
        ex c3_ans = lsolve(eqns, vars)[2].rhs(); // accessing the right-
            hand side expression

        cout << "\nExpress x(x1,x2,x3) as a linear combination of v1, v2, v3
            "<< endl;
```

```
        cout << "then solve A*c=x will yields"<< endl;
        cout << "\n " << lsolve(eqns, vars)<< endl;

        ex dom = c1_ans*v1 + c2_ans*v2 + c3_ans*v3 ;
        ex image = c1_ans*Tv1 + c2_ans*Tv2 + c3_ans*Tv3 ;

        cout << "\n (x1,x2,x3) = " << dom << endl;
        cout << "\n T(x1,x2,x3) = " << image << endl;
        cout << " T(x1,x2,x3) = " << image.evalm() << endl;

        cout << "\n T(2,-3,5) = " << subs(subs(subs(image.evalm(), x1 == 2)
            , x2 == -3), x3 == 5) << endl;
        cout << endl;

        return 0;
    }
```

**C++ Code 229:** *main.cpp "Computing with Images of Basis Vectors"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In **GiNaC** we need to learn how to store a vector, it is saved as variable declared with anme of **matrix**, then to make the matrix $A$ from basis vectors $v_1, v_2, v_3$ we will have to declare the entries one by one.

```
        ...
        ...

        matrix v1 = {{1,1,1}};
        matrix v2 = {{1,1,0}};
        matrix v3 = {{1,0,0}};

        matrix Tv1 = {{1,0}};
        matrix Tv2 = {{2,-1}};
        matrix Tv3 = {{4,3}};

        matrix A(3,3);
        A(0,0) = v1[0]; A(1,0) = v2[0]; A(2,0) = v3[0];
        A(0,1) = v1[1]; A(1,1) = v2[1]; A(2,1) = v3[1];
        A(0,2) = v1[2]; A(1,2) = v2[2]; A(2,2) = v3[2];

        ...
```

```
...
```

- When computing

$$T(2,-3,5)$$

we have to use the command of fractal **subs**, a **subs** inside a **subs** for three times because this is array / list expresssion, we cannot just use one **subs** and define the variables value all at once, it needs to be done one at a time.

```
...
...

cout << "\n T(2,-3,5) = " << subs(subs(subs(image.evalm(), x1
    == 2), x2 == -3), x3 == 5) << endl;
```

```
Matrix A consists of basis vector S ={v1, v2, v3} for R3

 A = [[1,1,1],[1,1,0],[1,0,0]]

 c = [[c1],[c2],[c3]]

 x = [[x1],[x2],[x3]]

Let T: R3 -> R2 be the linear transformation for which

 T(v1) = [[1,0]]

 T(v2) = [[2,-1]]

 T(v3) = [[4,3]]

 A*c = [[c1+c2+c3],[c1+c2],[c1]]

Express x(x1,x2,x3) as a linear combination of v1, v2, v3
then solve A*c=x will yields

  {c1==x3,c2==-x3+x2,c3==x1-x2}

 (x1,x2,x3) = (x1-x2)*[[1,0,0]]+x3*[[1,1,1]]-[[1,1,0]]*(x3-x2)

 T(x1,x2,x3) = (x1-x2)*[[4,3]]+x3*[[1,0]]-(x3-x2)*[[2,-1]]
 T(x1,x2,x3) = [[-x3+4*x1-2*x2,x3+3*x1-4*x2]]

 T(2,-3,5) = [[9,23]]
```

**Figure 23.219:** *The computation to compute $T(2,3-3,5)$ the image of basis vectors $S = \{v_1, v_2, v_3\}$ for the linear transformation $T : \mathbb{R}^3 \to \mathbb{R}^2$.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Images of Basis Vectors/main.cpp)****.*

## ii. C++ Computation: Compute An Inverse Transformation for $T : \mathbb{R}^3 \to \mathbb{R}^3$

Let $T : \mathbb{R}^3 \to \mathbb{R}^3$ be the linear operator defined by the formula

$$T(x_1, x_2, x_3) = (3x_1 + x_2, -2x_1 - 4x_2 + 3x_3, 5x_1 + 4x_2 - 2x_3)$$

Determine whether $T$ is one-to-one; if so, find $T^{-1}(x_1, x_2, x_3)$.

**Solution:**

It follows that the standard matrix for $T$ is

$$[T] = \begin{bmatrix} 3 & 1 & 0 \\ -2 & -4 & 3 \\ 5 & 4 & -2 \end{bmatrix}$$

This matrix is invertible, and the standard matrix for $T^{-1}$ is

$$[T^{-1}] = [T]^{-1} = \begin{bmatrix} 4 & -2 & -3 \\ -11 & 6 & 9 \\ -12 & 7 & 10 \end{bmatrix}$$

It follows that

$$T^{-1}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = [T^{-1}]\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 & -2 & -3 \\ -11 & 6 & 9 \\ -12 & 7 & 10 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4x_1 - 2x_2 - 3x_3 \\ -11x_1 + 6x_2 + 9x_3 \\ -12x_1 + 7x_2 + 10x_3 \end{bmatrix}$$

Expressing this result in horizontal notation yields

$$T^{-1}(x_1, x_2, x_3) = (4x_1 - 2x_2 - 3x_3, -11x_1 + 6x_2 + 9x_3, -12x_1 + 7x_2 + 10x_3)$$

For the C++ codes, we are using **Armadillo** to compute the inverse of matrix $T$, and then we use **GiNaC** to perform symbolic matrix-vector multiplication.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
#include <ginac/ginac.h>

using namespace std;
using namespace arma;
using namespace GiNaC;

// Driver code
int main(int argc, char** argv)
{
       int N = 3;
       mat T;
       mat T_inv;
```

```
        T.load("matrixT.txt");

        mat I(3,3,fill::eye);
        cout <<"[T]:" << "\n" << T <<endl;

        cout << "det(T) = " << det(T) << endl;
        cout << "rank(T) = " << arma::rank(T) << endl;

        T_inv = inv(T);
        cout << "\n[T]^{-1} =\n " << T_inv << endl;

        // GiNac codes starts here
        Digits = 5; // define maximum decimal digits
        symbol x1("x1"), x2("x2"), x3("x3");

        matrix T_invginac = { { T_inv[0], T_inv[0+N], T_inv[0+2*N] },
               { T_inv[1], T_inv[1+N], T_inv[1+2*N]},
               { T_inv[2], T_inv[2+N], T_inv[2+2*N]} };
        matrix X = {{x1}, {x2}, {x3}};

        cout << "\n [T]^{-1} (ginac) = " << T_invginac << endl;
        cout << "\n x = " << X << endl;

        ex Tinv_image = (T_invginac*X).evalm();
        cout << "\n [T]^{-1} * x = " << Tinv_image << endl;

        cout << "\n
            ************************************************************
            "<< endl;
        cout << "\nIf we input (x1,x2,x3) with (1,2,3)"<< endl;
        cout << "then the inverse will be"<< endl;

        float a = 1;
        float b = 2;
        float c = 3;

        cout << "\n [T]^{-1}(1,2,3) = " << subs(subs(subs(Tinv_image.evalm
            (), x1 == a), x2 == b), x3 == c) << endl;

        return 0;
    }
```

**C++ Code 230:** *main.cpp "Compute an inverse transformation from R3 to R3"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lcln**
**./main**

or with Makefile, type:

**make**
**./main**

Explanation for the codes:

- In **Armadillo** the first index $T[0]$ stands for the entries at the first row, first column ($T_{11}$), the second index stands for the entries at the second row, first column ($T_{21}$), overall they all will look like this:

  1. $T[0]$ in Armadillo = entry in matrix $T_{11}$
  2. $T[1]$ in Armadillo = entry in matrix $T_{21}$
  3. $T[2]$ in Armadillo = entry in matrix $T_{31}$
  4. $T[3]$ in Armadillo = entry in matrix $T_{12}$
  5. $T[4]$ in Armadillo = entry in matrix $T_{22}$
  6. $T[5]$ in Armadillo = entry in matrix $T_{32}$
  7. $T[6]$ in Armadillo = entry in matrix $T_{13}$
  8. $T[7]$ in Armadillo = entry in matrix $T_{23}$
  9. $T[8]$ in Armadillo = entry in matrix $T_{33}$

  so we copy all the entries of matrix $[T]^{-1}$ from **Armadillo** into **GiNaC**′ matrix with this technique.

```
...
...

Digits = 5; // define maximum decimal digits
symbol x1("x1"), x2("x2"), x3("x3");

matrix T_invginac = { { T_inv[0], T_inv[0+N], T_inv[0+2*N] },
       { T_inv[1], T_inv[1+N], T_inv[1+2*N]},
       { T_inv[2], T_inv[2+N], T_inv[2+2*N]} };
matrix X = {{x1}, {x2}, {x3}};

...
```

```
[T]:
   3.0000   1.0000        0
  -2.0000  -4.0000   3.0000
   5.0000   4.0000  -2.0000

det(T) = -1
rank(T) = 3

[T]^{-1} =
     4.0000   -2.0000   -3.0000
  -11.0000    6.0000    9.0000
  -12.0000    7.0000   10.0000


 [T]^{-1} (ginac) = [[4.0,-2.0,-3.0],[-11.0,6.0,9.0],[-12.0,7.0,10.0]]

 x = [[x1],[x2],[x3]]

 [T]^{-1} * x = [[-(3.0)*x3+(4.0)*x1-(2.0)*x2],[(9.0)*x3-(11.0)*x1+(6.0)*x2],[(1
0.0)*x3-(12.0)*x1+(7.0)*x2]]

************************************************************

If we input (x1,x2,x3) with (1,2,3)
then the inverse will be

 [T]^{-1}(1,2,3) = [[-9.0],[28.0],[32.0]]
```

**Figure 23.220:** *The computation to compute the inverse transformation $[T]^{-1}(x_1, x_2, x_3)$ for the linear operator $T : \mathbb{R}^3 \to \mathbb{R}^3$ defined by the formula $T(x_1, x_2, x_3) = (3x_1 + x_2, -2x_1 - 4x_2 + 3x_3, 5x_1 + 4x_2 - 2x_3)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Inverse Transformation of Linear Operator with Armadillo and GiNaC/-main.cpp**).*

## iii. C++ Computation: Linear Transformation from $P_1$ to $P_2$, Finding $T(\boldsymbol{x})$ Indirectly

Let $T : P_1 \to P_2$ be the linear transformation, and use the three-step procedure to perform the computation

$$T(a + bx) = x(a + bx) = ax + bx^2$$

**Solution**

1. The coordinate matrix for $\boldsymbol{x} = a + bx$ relative to the basis $B = \{1, x\}$ is

$$[\boldsymbol{x}]_B = \begin{bmatrix} a \\ b \end{bmatrix}$$

2. Multiplying $[\boldsymbol{x}]_B$ by the matrix $[T]_{B',B}$, we obtain

$$[T]_{B',B}[\boldsymbol{x}]_B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 \\ a \\ b \end{bmatrix} = [T(\boldsymbol{x})]_{B'}$$

3. Reconstructing $T(\boldsymbol{x}) = T(a + bx)$ from $[T(\boldsymbol{x})]_{B'}$, we obtain

$$T(a + bx) = 0 + ax + bx^2 = ax + bx^2$$

The C++ code is using **GiNac** library to do the symbolic matrix and vector computation.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

// T(a+bx) = x(a+bx) = ax + bx^2
// Driver code
int main(int argc, char** argv)
{
       // GiNac codes starts here
       Digits = 5; // define maximum decimal digits
       symbol x("x"), xpow2("x^2"), a("a"), b("b");

       // Initialize vector T(u1) and T(u2) as zero vectors first
       matrix Tu1 = {{0},{0},{0}};
       matrix Tu2 = {{0},{0},{0}};
       ex u1 = x;
       ex u2 = xpow2;

       cout << "\n u1 = " << u1 << endl;
       cout << "\n u2 = " << u2 << endl;

       // If else conditional to compute [T(u1)]_{B'}
       if (u1 == 1)
       {
              Tu1 = {{1}, {0}, {0}};
       }
       else if (u1 == x)
       {
              Tu1 = {{0}, {1}, {0}};
       }
       else if (u1 == xpow2)
       {
              Tu1 = {{0}, {0}, {1}};
       }

       // If else conditional to compute [T(u2)]_{B'}
       if (u2 == 1)
       {
              Tu2 = {{1}, {0}, {0}};
       }
       else if (u2 == x)
       {
              Tu2 = {{0}, {1}, {0}};
```

```
        }
        else if (u2 == xpow2)
        {
            Tu2 = {{0}, {0}, {1}};
        }

        cout << "\n T(u1) = " << Tu1 << endl;
        cout << "\n T(u2) = " << Tu2 << endl;

        // Construct matrix [T]_{B',B}
        matrix TBBnew(3,2);
        TBBnew(0,0) = Tu1[0];
        TBBnew(0,1) = Tu2[0];
        TBBnew(1,0) = Tu1[1];
        TBBnew(1,1) = Tu2[1];
        TBBnew(2,0) = Tu1[2];
        TBBnew(2,1) = Tu2[2];

        matrix xB(2,1);
        xB(0,0) = a;
        xB(1,0) = b;

        // Construct a vector 1X3 as the new basis vector = {1,x,x^2}
        matrix newBasis(1,3);
        newBasis(0,0) = 1;
        newBasis(0,1) = x;
        newBasis(0,2) = xpow2;

        cout << "\n [T]_{B',B} = " << TBBnew << endl;
        cout << "\n [x]_{B} = " << xB << endl;

        ex TxBnew = (TBBnew*xB).evalm();
        cout << "\n [T]_{B',B} * [x]_{B}= " << TxBnew << endl;

        ex Tx = (newBasis*TxBnew).evalm();
        cout << "\n T(a+bx)= " << Tx << endl;

        return 0;
    }
```

**C++ Code 231:** *main.cpp "Finding T(x) indirectly"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We construct vector $B' = \{1, x, x^2\}$ as a $1 \times 3$ vector in **GiNac** so we can multiply it with $[T]_{B',B}[\boldsymbol{x}]_B$ to obtain $T(\boldsymbol{x}) = T(a + bx) = ax + bx^2$. There is no dot product function in **GiNac** yet, so this is how we can get through.

```
...
...

// Construct a vector 1 X 3 as the new basis vector = {1,x,x^2}
matrix newBasis(1,3);
newBasis(0,0) = 1;
newBasis(0,1) = x;
newBasis(0,2) = xpow2;

cout << "\n [T]_{B',B} = " << TBBnew << endl;
cout << "\n [x]_{B} = " << xB << endl;

ex TxBnew = (TBBnew*xB).evalm();
cout << "\n [T]_{B',B} * [x]_{B}= " << TxBnew << endl;

ex Tx = (newBasis*TxBnew).evalm();
cout << "\n T(a+bx)= " << Tx << endl;
```

```
u1 = x
u2 = x^2
T(u1) = [[0],[1],[0]]
T(u2) = [[0],[0],[1]]
[T]_{B',B} = [[0,0],[1,0],[0,1]]
[x]_{B} = [[a],[b]]
[T]_{B',B} * [x]_{B}= [[0],[a],[b]]
T(a+bx)= [[a*x+b*x^2]]
```

**Figure 23.221:** *The computation to compute $T(\boldsymbol{x}) = T(a + bx)$ with $T : P_1 \to P_2$ as the linear transformation. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Linear Transformation from P1 to P2 with GiNaC/main.cpp**).*

## iv. C++ Computation: Matrix for a Linear Transformation $T : \mathbb{R}^2 \to \mathbb{R}^3$

Let $T : \mathbb{R}^2 \to \mathbb{R}^3$ be the linear transformation defined by

$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_2 \\ -5x_1 + 13x_2 \\ -7x_1 + 16x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -5 & 13 \\ -7 & 16 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Find the matrix for the transformation $T$ with respect to the bases $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ for $\mathbb{R}^2$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ for $\mathbb{R}^3$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} -1 \\ 22 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

**Solution:**

From the formula for $T$,

$$T(\boldsymbol{u}_1) = \begin{bmatrix} 1 \\ -2 \\ -5 \end{bmatrix}, \quad T(\boldsymbol{u}_2) = \begin{bmatrix} 2 \\ 1 \\ -3 \end{bmatrix}$$

Expressing these vectors as linear combinations of $\boldsymbol{v}_1, \boldsymbol{v}_2$, and $\boldsymbol{v}_3$, we obtain

$$\begin{aligned} T(\boldsymbol{u}_1) &= \boldsymbol{v}_1 - 2\boldsymbol{v}_3 \\ T(\boldsymbol{u}_2) &= 3\boldsymbol{v}_1 + \boldsymbol{v}_2 - \boldsymbol{v}_3 \end{aligned}$$

Thus,

$$[T(\boldsymbol{u}_1)]_{B'} = \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}, \quad [T(\boldsymbol{u}_2)]_{B'} = \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix}$$

so

$$[T]_{B',B} = [[T(\boldsymbol{u}_1)]_{B'} | [T(\boldsymbol{u}_2)]_{B'}] = \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ -2 & -1 \end{bmatrix}$$

In this case the matrices

$$[T] = \begin{bmatrix} 0 & 1 \\ -5 & 13 \\ -7 & 16 \end{bmatrix}, \quad [T]_{B',B} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ -2 & -1 \end{bmatrix}$$

both represent the transformation $T$, the first relative to the standard bases for $\mathbb{R}^2$ and $\mathbb{R}^3$, the second relative to the bases $B$ and $B'$ stated in the example.

A fixed linear transformation generally has multiple representations, each depending on the bases chosen. In this case the matrices

$$[T] = \begin{bmatrix} 0 & 1 \\ -5 & 13 \\ -7 & 16 \end{bmatrix}$$

and

$$[T]_{B',B} = \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ -2 & -1 \end{bmatrix}$$

both represent the transformation $T$, the first relative to the standard bases for $\mathbb{R}^2$ and $\mathbb{R}^3$, the second relative to the bases $B$ and $B'$.

For the C++ codes, we started with **GiNac**, but then we decided to add **Armadillo** and put it above **GiNaC** code because it can compute $T(\boldsymbol{u}_1)$ and $T(\boldsymbol{u}_2)$ easily with **solve()** function from **Armadillo** library. Any reader who want to continue to finish the computation with **GiNac** are very much welcomed.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;
using namespace GiNaC;

// Linear transformation T: R^2 —> R^3
// Driver code
int main(int argc, char** argv)
{

       mat matrixT;
       matrixT.load("matrixT.txt");
       // column vectors are treated like a matrix with one column, the
           function is 'vec'
       vec u1 = { 3, 1 };
       vec u2 = { 5, 2 };
       vec v1 = { 1, 0, —1 };
       vec v2 = { —1, 2, 2 };
       vec v3 = { 0, 1, 2 };

       mat matrix_Tu1 = matrixT*u1 ;
       mat matrix_Tu2 = matrixT*u2 ;
       mat B(3,3,fill::zeros);
       B.col(0) = v1;
       B.col(1) = v2;
       B.col(2) = v3;

       cout << "Old basis B = {u1, u2}"<< endl;
       cout << "u1: \n" << u1 << endl;
       cout << "u2: \n" << u2 << endl;
```

```
cout << "New basis B' = {v1, v2, v3}"<< endl;
cout << "v1: \n" << v1 << endl;
cout << "v2: \n" << v2 << endl;
cout << "v3: \n" << v3 << endl;
cout << "T(u1): \n" << matrix_Tu1 << endl;
cout << "T(u2): \n" << matrix_Tu2 << endl;

// Compute [T(u1)]_{B'} and [T(u2)]_{B'}
mat Tu1_newBasis;
mat Tu2_newBasis;
Tu1_newBasis = solve(B,matrix_Tu1,solve_opts::force_approx);
Tu2_newBasis = solve(B,matrix_Tu2,solve_opts::force_approx);
// Rounding the entries to the nearest integer
Tu1_newBasis[0] = lround(Tu1_newBasis[0]);
Tu1_newBasis[1] = lround(Tu1_newBasis[1]);
Tu1_newBasis[2] = lround(Tu1_newBasis[2]);
Tu2_newBasis[0] = lround(Tu2_newBasis[0]);
Tu2_newBasis[1] = lround(Tu2_newBasis[1]);
Tu2_newBasis[2] = lround(Tu2_newBasis[2]);

cout <<"Matrix from new basis B':" << "\n" << B <<endl;
cout <<"Expressing T(u1) and T(u2) as the linear combinations of v1,
     v2, and v3:" <<endl;
cout <<"[T(u1)]_{B'}:" << "\n" << Tu1_newBasis <<endl;
cout <<"[T(u2)]_{B'}:" << "\n" << Tu2_newBasis <<endl;

mat TBnewBold(3,2,fill::zeros);
TBnewBold.col(0) = Tu1_newBasis ;
TBnewBold.col(1) = Tu2_newBasis ;
matrixT.print("[T]:\n");
TBnewBold.print("[T]_{B',B}:\n");


Digits = 5; // define maximum decimal digits
symbol x("x"), xpow2("x^2"), a("a"), b("b"), x1("x1"), x2("x2");

matrix T = {{0,1},{-5,13},{-7,16}};

// Create vector x=(x1, x2)
matrix vectorx(2,1);
vectorx(0,0) = x1;
vectorx(1,0) = x2;

cout << "\n [T] = " << T << endl;
cout << "\n [T] * x = " << (T*vectorx).evalm() << endl;
ex Tx = (T*vectorx).evalm() ;
matrix X = {{x1}, {x2}};
cout << "\n [T] * x = " << (T*X).evalm() << endl;
```

```
        // Create vector T(u1), with u1 = (3,1) and u2 = (5,2)
        ex Tu1 = subs(subs( (T*X).evalm(), x1==3), x2==1);
        ex Tu2 = subs(subs( (T*X).evalm(), x1==5), x2==2);
        cout << "\n T(u1) = " << Tu1 << endl;
        cout << "\n T(u2) = " << Tu2 << endl;

        // Construct matrix [T]_{B,B'}
        matrix TBBnew(3,2);
        TBBnew(0,0) = Tu1[0];
        TBBnew(0,1) = Tu2[0];
        TBBnew(1,0) = Tu1[1];
        TBBnew(1,1) = Tu2[1];
        TBBnew(2,0) = Tu1[2];
        TBBnew(2,1) = Tu2[2];

        matrix xB(2,1);
        xB(0,0) = a;
        xB(1,0) = b;

        //cout << "\n [T]_{B , B'} = " << TBBnew << endl;

        return 0;
    }
```

**C++ Code 232:** *main.cpp "Matrix for a linear transformation from R2 to R3"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
*********************************************************************
*************************Armadillo Codes*************************
*********************************************************************
Old basis B = {u1, u2}
u1:
   3.0000
   1.0000

u2:
   5.0000
   2.0000

New basis B' = {v1, v2, v3}
v1:
   1.0000
        0
  -1.0000

v2:
  -1.0000
   2.0000
   2.0000

v3:
        0
   1.0000
   2.0000

T(u1):
   1.0000
  -2.0000
  -5.0000

T(u2):
   2.0000
   1.0000
  -3.0000

Matrix from new basis B':
   1.0000  -1.0000        0
        0   2.0000   1.0000
  -1.0000   2.0000   2.0000
```

**Figure 23.222:** *The computation to compute $[T]_{B',B}$ with $T : \mathbb{R}^2 \to \mathbb{R}^3$ as the linear transformation. (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Linear Transformation from R2 to R3 with Armadillo and GiNaC/main.cpp**).*

```
Expressing T(u1) and T(u2) as the linear combinations of v1, v2, and v3:
[T(u1)]_{B'}:
   1.0000
        0
  -2.0000

[T(u2)]_{B'}:
   3.0000
   1.0000
  -1.0000

[T]:

         0    1.0000
   -5.0000   13.0000
   -7.0000   16.0000
[T]_{B',B}:

   1.0000   3.0000
        0   1.0000
  -2.0000  -1.0000
*********************************************************************
****************************GiNaC Codes******************************
*********************************************************************

 [T] = [[0,1],[-5,13],[-7,16]]

 [T] * x = [[x2],[-5*x1+13*x2],[-7*x1+16*x2]]

 [T] * x = [[x2],[-5*x1+13*x2],[-7*x1+16*x2]]

 T(u1) = [[1],[-2],[-5]]

 T(u2) = [[2],[1],[-3]]
```

**Figure 23.223:** *The computation to compute $[T]_{B',B}$ with $T : \mathbb{R}^2 \to \mathbb{R}^3$ as the linear transformation. (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Linear Transformation from R2 to R3 with Armadillo and GiNaC/main.cpp**).*

## v. C++ Computation: Linear Operator on $P_2$

Let $T : P_2 \to P_2$ be the linear operator defined by

$$T(p(x)) = p(3x - 5)$$

that is, $T(c_0 + c_1 x + c_2 x^2) = c_0 + c_1(3x - 5) + c_2(3x - 5)^2$.

(a) Find $[T]_B$ relatve to the basis $B = \{1, x, x^2\}$.

(b) Use the indirect procedure to compute $T(1 + 2x + 3x^2)$.

(c) Check the result in (b) by computing $T(1 + 2x + 3x^2)$ directly.

**Solution:**

(a) From the formula for $T$,

$$\begin{aligned} T(1) &= 1 \\ T(x) &= 3x - 5 \\ T(x^2) &= (3x - 5)^2 = 9x^2 - 30x + 25 \end{aligned}$$

so

$$[T(1)]_B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad [T(x)]_B = \begin{bmatrix} -5 \\ 3 \\ 0 \end{bmatrix}, \quad [T(x^2)]_B = \begin{bmatrix} 25 \\ -30 \\ 9 \end{bmatrix}$$

Thus,

$$[T]_B = \begin{bmatrix} 1 & -5 & 25 \\ 0 & 3 & -30 \\ 0 & 0 & 9 \end{bmatrix}$$

(b) The coordinate matrix for $\boldsymbol{p} = 1 + 2x + 3x^2$ relative to the basis $B = \{1, x, x^2\}$ is

$$[\boldsymbol{p}]_B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Multiplying $[\boldsymbol{p}]_B$ by the matrix $[T]_B$ found in part (a) we obtain

$$[T]_B[\boldsymbol{p}]_B = \begin{bmatrix} 1 & -5 & 25 \\ 0 & 3 & -30 \\ 0 & 0 & 9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 66 \\ -84 \\ 27 \end{bmatrix} = [T(\boldsymbol{p})]_B$$

Reconstructing $T(\boldsymbol{p}) = T(1 + 2x + 3x^2)$ from $[T(\boldsymbol{p})]_B$ we obtain

$$T(1 + 2x + 3x^2) = 66 - 84x + 27x^2$$

(c) By direct computation,

$$\begin{aligned} T(1 + 2x + 3x^2) &= 1 + 2(3x - 5) + 3(3x - 5)^2 \\ &= 1 + 6x - 10 + 27x^2 - 90x + 75 \\ &= 66 - 84x + 27x^2 \end{aligned}$$

which agrees with the result in (b).

The C++ code is using **GiNaC** library only, if you want a more sophisticated vector and matrix looks, you can copy the entries of vector, matrix and the polynomial coefficient from **GiNaC** to vector and matrix entries in **Armadillo**.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include <vector>

using namespace std;
using namespace GiNaC;

// Linear operator T: P_2 —> P_2
// Driver code
int main(int argc, char** argv)
{

       Digits = 5; // define maximum decimal digits
       symbol x("x"), xpow2("x^2"), a("a"), b("b"), x1("x1"), x2("x2");
```

```
        ex px = 3*x - 5;

        cout << "\n T(1) = " << 1 << endl;
        cout << "\n T(x) = " << px << endl;
        cout << "\n T(x^2) = " << pow(px,2).expand() << endl;

        matrix T1 = {{1},{0},{0}};
        matrix Tx = {{-5},{3},{0}};
        matrix Tx2 = {{25},{-30},{9}};
        matrix pB = {{1},{2},{3}};

        cout << "\n [T(1)]_{B} = " << T1 << endl;
        cout << "\n [T(x)]_{B} = " << Tx << endl;
        cout << "\n [T(x^2)]_{B} = " << Tx2 << endl;

        matrix TB(3,3);
        TB(0,0) = T1[0];
        TB(0,1) = Tx[0];
        TB(0,2) = Tx2[0];
        TB(1,0) = T1[1];
        TB(1,1) = Tx[1];
        TB(1,2) = Tx2[1];
        TB(2,0) = T1[2];
        TB(2,1) = Tx[2];
        TB(2,2) = Tx2[2];

        ex Tp = (1 + 2*px + 3*(pow(px,2))).expand();

        cout << "\n [T]_{B} = " << TB << endl;
        cout << "\n [p]B = " << pB << endl;
        cout << "\n [T(p)]_{B} = [T]_{B} * [p]B = " << (TB*pB).evalm() <<
            endl;

        cout << "\n T(1 + 2x + 3x^2) = " << Tp << endl;

        cout << "\n For T(1 + 2x + 3x^2) : " << endl;

        for (int i = Tp.ldegree(x); i <= Tp.degree(x); ++i)
        {
               cout << " The x^" << i << "-coefficient is " << Tp.coeff(x,i
                   ) << endl;
        }

        return 0;
    }
```

**C++ Code 233:** *main.cpp "Linear Operator on P2"*

To compile it, type:

**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- This **For** loop is used to obtain the coefficient of polynomial $T(\boldsymbol{p}) = T(1 + 2x + 3x^2) = 66 - 84x + 27x^2$.

```
...
...

for (int i = Tp.ldegree(x); i <= Tp.degree(x); ++i)
{
       cout << " The x^" << i << "—coefficient is " << Tp.coeff
           (x,i) << endl;
}
```

```
T(1) = 1

T(x) = -5+3*x

T(x^2) = 25-30*x+9*x^2

[T(1)]_{B} = [[1],[0],[0]]

[T(x)]_{B} = [[-5],[3],[0]]

[T(x^2)]_{B} = [[25],[-30],[9]]

[T]_{B} = [[1,-5,25],[0,3,-30],[0,0,9]]

[p]B = [[1],[2],[3]]

[T(p)]_{B} = [T]_{B} * [p]B = [[66],[-84],[27]]

T(1 + 2x + 3x^2) = 66-84*x+27*x^2

For T(1 + 2x + 3x^2) :
The x^0-coefficient is 66
The x^1-coefficient is -84
The x^2-coefficient is 27
```

**Figure 23.224:** *The computation to compute $[T]_B$ relative to the basis $B = \{1, x, x^2\}$ and $T(1 + 2x + 3x^2)$ from linear operator with $T : P_2 \to P_2$ defined by $T(p(x)) = p(3x - 5)$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Linear Operator on P2 with GiNaC/main.cpp**).*

## vi. C++ Computation: Composition of Linear Transformations

Let $T_1 : P_1 \to P_2$ be the linear transformation defined by

$$T_1(p(x)) = xp(x)$$

and let $T_2 : P_2 \to P_2$ be the linear operator defined by

$$T_2(p(x)) = p(3x - 5)$$

Then the composition $(T_2 \circ T_1) : P_1 \to P_2$ is given by

$$\begin{aligned}(T_2 \circ T_1)(p(x)) &= T_2(T_1(p(x))) \\ &= T_2(xp(x)) \\ &= (3x-5)p(3x-5)\end{aligned}$$

Thus, if $p(x) = c_0 + c_1 x$, then

$$\begin{aligned}(T_2 \circ T_1)(c_0 + c_1 x) &= (3x-5)(c_0 + c_1(3x-5)) \\ &= c_0(3x-5) + c_1(3x-5)^2\end{aligned}$$

In this example, $P_1$ plays the role of $U$ in theorem about matrices of compositions, and $P_2$ plays the role of both $V$ and $W$; thus we can take $B' = B''$ so that the formula simplifies to

$$[T_2 \circ T_1]_{B',B} = [T_2]_{B'}[T_1]_{B',B} \tag{23.277}$$

Let us choose $B = \{1, x\}$ to be the basis for $P_1$ and choose $B' = \{1, x, x^2\}$ to be the basis for $P_2$. Then we will have

$$[T_1]_{B',B} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad [T_2]_{B'} = \begin{bmatrix} 1 & -5 & 25 \\ 0 & 3 & -30 \\ 0 & 0 & 9 \end{bmatrix}$$

Thus

$$[T_2 \circ T_1]_{B',B} = \begin{bmatrix} 1 & -5 & 25 \\ 0 & 3 & -30 \\ 0 & 0 & 9 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -5 & 25 \\ 3 & -30 \\ 0 & 9 \end{bmatrix}$$

As a check, we will calculate $[T_2 \circ T_1]_{B',B}$ directly. Since $B = \{1, x\}$, with $\boldsymbol{u}_1 = 1$ and $\boldsymbol{u}_2 = x$, we will obtain

$$[T_2 \circ T_1]_{B',B} = [[(T_2 \circ T_1)(1)]_{B'} | [(T_2 \circ T_1)(x)]_{B'}] \tag{23.278}$$

hence

$$(T_2 \circ T_1)(1) = 3x - 5$$

$$(T_2 \circ T_1)(x) = (3x-5)^2 = 9x^2 - 30x + 25$$

From this and the fact that $B = \{1, x, x^2\}$, it follows that

$$[(T_2 \circ T_1)(1)]_{B'} = \begin{bmatrix} -5 \\ 3 \\ 0 \end{bmatrix}$$

and

$$[(T_2 \circ T_1)(x)]_{B'} = \begin{bmatrix} 25 \\ -30 \\ 9 \end{bmatrix}$$

Substituting will yields

$$[(T_2 \circ T_1)(x)]_{B',B} = \begin{bmatrix} -5 & 25 \\ 3 & -30 \\ 0 & 9 \end{bmatrix}$$

which agrees with the result previously.

The C++ code is using **GiNaC** for the symbolic computation and matrix computation.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include <vector>
#include <armadillo>

using namespace std;
using namespace GiNaC;
using namespace arma;

// Linear operator T_{1}: P_2 —> P_2
// Linear operator T_{2}: P_2 —> P_2
// Driver code
int main(int argc, char** argv)
{

       Digits = 5; // define maximum decimal digits
       symbol x("x"), xpow2("x^2"), a("a"), b("b"), x1("x1"), x2("x2");

       ex px = 3*x — 5;

       cout << "\n T(1) = " << 1 << endl;
       cout << "\n T(x) = " << px << endl;
       cout << "\n T(x^2) = " << pow(px,2).expand() << endl;

       matrix T2col1 = {{1},{0},{0}};
       matrix T2col2 = {{—5},{3},{0}};
       matrix T2col3 = {{25},{—30},{9}};

       matrix T1col1 = {{0},{1},{0}};
       matrix T1col2 = {{0},{0},{1}};

       matrix T2Bnew(3,3);
       T2Bnew(0,0) = T2col1[0];
       T2Bnew(0,1) = T2col2[0];
```

```
        T2Bnew(0,2) = T2col3[0];
        T2Bnew(1,0) = T2col1[1];
        T2Bnew(1,1) = T2col2[1];
        T2Bnew(1,2) = T2col3[1];
        T2Bnew(2,0) = T2col1[2];
        T2Bnew(2,1) = T2col2[2];
        T2Bnew(2,2) = T2col3[2];

        matrix T1BBnew(3,2);
        T1BBnew(0,0) = T1col1[0];
        T1BBnew(0,1) = T1col2[0];
        T1BBnew(1,0) = T1col1[1];
        T1BBnew(1,1) = T1col2[1];
        T1BBnew(2,0) = T1col1[2];
        T1BBnew(2,1) = T1col2[2];

        cout << "\n [T_{2}]_{B'} = " << T2Bnew << endl;
        cout << "\n [T_{1}]_{B',B} = " << T1BBnew << endl;
        cout << "\n [T_{2} o T_{1}]_{B', B} = " << (T2Bnew*T1BBnew).evalm()
            << endl;

        return 0;
    }
```

**C++ Code 234:** *main.cpp "Composition of linear transformations"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
*****************************************************************
***************************GiNaC Codes*****************************
*****************************************************************

T(1) = 1

T(x) = -5+3*x

T(x^2) = 25-30*x+9*x^2

[T_{2}]_{B'} = [[1,-5,25],[0,3,-30],[0,0,9]]

[T_{1}]_{B',B} = [[0,0],[1,0],[0,1]]

[T_{2} o T_{1}]_{B', B} = [[-5,25],[3,-30],[0,9]]
```

**Figure 23.225:** *The computation to compute the composition* $[T_2 \circ T_1]_{B',B}$. *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Composition Transformations with GiNaC/main.cpp**).*

## vii. C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Linear Transformation

Find the matrix $T$ relative to the basis $B$, and then compute matrix for $T$ relative to the basis $B'$.

Suppose that $T : \mathbb{R}^2 \to \mathbb{R}^2$ is defined by

$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} 3x_1 - x_2 \\ x_1 \end{bmatrix}$$

and $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

**Solution:**

$$T(\boldsymbol{u}_1) = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \quad T(\boldsymbol{u}_2) = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

so

$$[T]_B = \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix}$$

By inspection,

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{u}_1 - 3\boldsymbol{u}_2 \\ \boldsymbol{v}_2 &= 3\boldsymbol{u}_1 \end{aligned}$$

so the transition matrix from $B'$ to $B$ is

$$P = \begin{bmatrix} 1 & 3 \\ -3 & 0 \end{bmatrix}$$

Thus,

$$P_{B \to B'} = P^{-1} = \begin{bmatrix} 0 & -\frac{1}{3} \\ \frac{1}{3} & \frac{1}{9} \end{bmatrix}$$

Now, to compute $[T]_{B'}$ we will use the formula

$$[T]_{B'} = P^{-1}[T]_B P = \begin{bmatrix} -\frac{1}{3} & -1 \\ \frac{19}{9} & \frac{30}{9} \end{bmatrix}$$

The C++ code is using **Armadillo** library only for this problem.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>
```

```
using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       // column vectors are treated like a matrix with one column,
           the function is 'vec'
       vec u1 = { 1, 0 };
       vec u2 = { 0, 1 };
       vec v1 = { 1, -3 };
       vec v2 = { 3, 0 };

       mat T;
       T.load("matrixT.txt");

       mat Tu1 = T*u1;
       mat Tu2 = T*u2;

       cout <<"Matrix T:" << "\n" << T <<endl;
       cout <<"T(u1):" << "\n" << Tu1 <<endl;
       cout <<"T(u2):" << "\n" << Tu2 <<endl;

       mat A(2,2,fill::zeros);
       A.col(0) = u1;
       A.col(1) = u2;
       mat V1;
       mat V2;
       V1 = solve(A,v1,solve_opts::force_approx);
       V2 = solve(A,v2,solve_opts::force_approx);

       cout <<"Matrix B [u1 | u2]:" << "\n" << A <<endl;
       cout <<"v1_{B}:" << "\n" << V1 <<endl;
       cout <<"v2_{B}:" << "\n" << V2 <<endl;

       //cout << "det(T) = " << det(T) << endl;
       //cout << "rank(T) = " << arma::rank(T) << endl;

       mat P(2,2,fill::zeros);
       P.col(0) = V1;
       P.col(1) = V2;
       cout <<"Transition matrix P:" << "\n" << P <<endl;
       cout <<"P^{-1}:" << "\n" << inv(P) <<endl;
       cout <<"[T]_{B'} = P^{-1} * [T]_{B} * P:" << "\n" << inv(P)*
           T*P <<endl;

       return 0;
}
```

**C++ Code 235:** *main.cpp "Compute matrix T relative to the basis B' for linear transformation"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We are using Gaussian elimination (the function in **Armadillo** is **solve(A,b,option)** to solve for $\boldsymbol{x}$ in $A\boldsymbol{x} = \boldsymbol{b}$), thus the codes to obtained $[\boldsymbol{v}_1]_B$ and $[\boldsymbol{v}_2]_B$ are

```
...
...

mat A(2,2,fill::zeros);
A.col(0) = u1;
A.col(1) = u2;
mat V1;
mat V2;
V1 = solve(A,v1,solve_opts::force_approx);
V2 = solve(A,v2,solve_opts::force_approx);

...
...
```

```
Matrix T:
   3.0000  -1.0000
   1.0000        0

T(u1):
   3.0000
   1.0000

T(u2):
  -1.0000
        0

Matrix B [u1 | u2]:
   1.0000        0
        0   1.0000

v1_{B}:
   1.0000
  -3.0000

v2_{B}:
   3.0000
        0

Transition matrix P:
   1.0000   3.0000
  -3.0000        0

P^{-1}:
        0  -0.3333
   0.3333   0.1111

[T]_{B'} = P^{-1} * [T]_{B} * P:
  -0.3333  -1.0000
   2.1111   3.3333
```

**Figure 23.226:** *The computation to compute $[T]_{B'}$.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix T Relative to Basis B′ with Armadillo/main.cpp)****.*

## viii. C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Rotation

Find the matrix $T$ relative to the basis $B$, and then compute matrix for $T$ relative to the basis $B'$.

Suppose that $T : \mathbb{R}^2 \to \mathbb{R}^2$ is the rotation about the origin through an angle of $45^0$.

and $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

**Solution:**

Since $B$ is the standard basis for $\mathbb{R}^2$, we will have

$$[T]_B = \begin{bmatrix} \cos\ 45^0 & -\sin\ 45^0 \\ \sin\ 45^0 & \cos\ 45^0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Since the basis $B$ and $B'$ are the same with the previous section, then $P_{B' \to B}$ and $P_{B \to B'}$ will be the

same as well, so

$$[T]_{B'} = P_{B\to B'}[T]_{B'}P_{B'\to B} = \begin{bmatrix} 0 & -\frac{1}{3} \\ \frac{1}{3} & \frac{1}{9} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 3 \\ -3 & 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{3} & -\frac{1}{\sqrt{2}} \\ \frac{5\sqrt{2}}{3} & \frac{2\sqrt{2}}{3} \end{bmatrix}$$

The C++ code is also using **Armadillo** like the previous section.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
// Driver code
int main(int argc, char** argv)
{
        // column vectors are treated like a matrix with one column, the
            function is 'vec'
        vec u1 = { 1, 0 };
        vec u2 = { 0, 1 };
        vec v1 = { 1, -3 };
        vec v2 = { 3, 0 };
        int deg = 45;
        float theta = deg*DEGTORAD;
        vec t1 = { cos(theta), sin(theta) };
        vec t2 = { -sin(theta), cos(theta) };

        mat T(2,2,fill::zeros);
        T.col(0) = t1;
        T.col(1) = t2;

        mat Tu1 = T*u1;
        mat Tu2 = T*u2;

        cout <<"Matrix T:" << "\n" << T <<endl;
        cout <<"T(u1):" << "\n" << Tu1 <<endl;
        cout <<"T(u2):" << "\n" << Tu2 <<endl;

        mat A(2,2,fill::zeros);
        A.col(0) = u1;
        A.col(1) = u2;
        mat V1;
        mat V2;
        V1 = solve(A,v1,solve_opts::force_approx);
        V2 = solve(A,v2,solve_opts::force_approx);
```

```
        cout <<"Matrix B [u1 | u2]:" << "\n" << A <<endl;
        cout <<"v1_{B}:" << "\n" << V1 <<endl;
        cout <<"v2_{B}:" << "\n" << V2 <<endl;

        mat P(2,2,fill::zeros);
        P.col(0) = V1;
        P.col(1) = V2;
        cout <<"Transition matrix P:" << "\n" << P <<endl;
        cout <<"P^{-1}:" << "\n" << inv(P) <<endl;
        cout <<"[T]_{B'} = P^{-1} * [T]_{B} * P:" << "\n" << inv(P)*T*P <<
            endl;

        return 0;
    }
```

**C++ Code 236:** *main.cpp "Compute matrix T relative to the basis B′ for rotation"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix T:
   0.7071  -0.7071
   0.7071   0.7071

T(u1):
   0.7071
   0.7071

T(u2):
  -0.7071
   0.7071

Matrix B [u1 | u2]:
   1.0000        0
        0   1.0000

v1_{B}:
   1.0000
  -3.0000

v2_{B}:
   3.0000
        0

Transition matrix P:
   1.0000   3.0000
  -3.0000        0

P^{-1}:
        0  -0.3333
   0.3333   0.1111

[T]_{B'} = P^{-1} * [T]_{B} * P:
   0.4714  -0.7071
   0.7857   0.9428
```

**Figure 23.227:** *The computation to compute $[T]_{B'}$ for rotation linear operator in $\mathbb{R}^2$.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix T Relative to Basis B′ for Rotation Linear Operator with Armadillo/main.cpp)****.*

## ix. C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Orthogonal Projection on the $xy$-Plane

Find the matrix $T$ relative to the basis $B$, and then compute matrix for $T$ relative to the basis $B'$.

Suppose that $T : \mathbb{R}^2 \to \mathbb{R}^2$ is the orthogonal projection on the $xy$-plane and $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ and $B' = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_2\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

and

$$\boldsymbol{v}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix}1\\1\\0\end{bmatrix}, \quad \boldsymbol{v}_3 = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

**Solution:**

$$T(\boldsymbol{u}_1) = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad T(\boldsymbol{u}_2) = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad T(\boldsymbol{u}_3) = \begin{bmatrix}0\\0\\0\end{bmatrix}$$

so

$$[T]_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

By inspection,

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{u}_1 \\ \boldsymbol{v}_2 &= \boldsymbol{u}_1 + \boldsymbol{u}_2 \\ \boldsymbol{v}_3 &= \boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{u}_3 \end{aligned}$$

so the transition matrix from $B'$ to $B$ is

$$P = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus,

$$P_{B \to B'} = P^{-1} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, to compute $[T]_{B'}$ we will use the formula

$$[T]_{B'} = P^{-1}[T]_B P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

The C++ code is still using **Armadillo** library.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
        // column vectors are treated like a matrix with one column, the
            function is 'vec'
        vec u1 = { 1, 0, 0 };
        vec u2 = { 0, 1, 0 };
        vec u3 = { 0, 0, 1 };
        vec v1 = { 1, 0, 0 };
        vec v2 = { 1, 1, 0 };
        vec v3 = { 1, 1, 1 };

        mat T;
        T.load("matrixT.txt");
```

```
        mat Tu1 = T*u1;
        mat Tu2 = T*u2;
        mat Tu3 = T*u3;

        cout <<"Matrix T:" << "\n" << T <<endl;
        cout <<"T(u1):" << "\n" << Tu1 <<endl;
        cout <<"T(u2):" << "\n" << Tu2 <<endl;
        cout <<"T(u3):" << "\n" << Tu3 <<endl;

        mat A(3,3,fill::zeros);
        A.col(0) = u1;
        A.col(1) = u2;
        A.col(2) = u3;
        mat V1;
        mat V2;
        mat V3;
        V1 = solve(A,v1,solve_opts::force_approx);
        V2 = solve(A,v2,solve_opts::force_approx);
        V3 = solve(A,v3,solve_opts::force_approx);

        cout <<"Matrix B [u1 | u2 | u3]:" << "\n" << A <<endl;
        cout <<"v1_{B}:" << "\n" << V1 <<endl;
        cout <<"v2_{B}:" << "\n" << V2 <<endl;
        cout <<"v3_{B}:" << "\n" << V3 <<endl;

        mat P(3,3,fill::zeros);
        P.col(0) = V1;
        P.col(1) = V2;
        P.col(2) = V3;
        cout <<"Transition matrix P:" << "\n" << P <<endl;
        cout <<"P^{-1}:" << "\n" << inv(P) <<endl;
        cout <<"[T]_{B'} = P^{-1} * [T]_{B} * P:" << "\n" << inv(P)*T*P <<
            endl;

        return 0;
    }
```

**C++ Code 237:** *main.cpp "Compute matrix T relative to the basis B′ for orthogonal projection on xy-plane"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
T(u1):
   1.0000
        0
        0

T(u2):
        0
   1.0000
        0

T(u3):
        0
        0
        0

Matrix B [u1 | u2 | u3]:
   1.0000        0        0
        0   1.0000        0
        0        0   1.0000

v1_{B}:
   1.0000
        0
        0

v2_{B}:
   1.0000
   1.0000
        0

v3_{B}:
   1.0000
   1.0000
   1.0000

Transition matrix P:
   1.0000   1.0000   1.0000
        0   1.0000   1.0000
        0        0   1.0000

P^{-1}:
   1.0000  -1.0000        0
        0   1.0000  -1.0000
        0        0   1.0000

[T]_{B'} = P^{-1} * [T]_{B} * P:
   1.0000        0        0
        0   1.0000   1.0000
        0        0        0
```

**Figure 23.228:** *The computation to compute $[T]_{B'}$ for orthogonal projection on xy-plane linear operator in $\mathbb{R}^3$. (**DF-SimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix T Relative to Basis B′ for Orthogonal Projection on xy-Plane with Armadillo/main.cpp**).*

## x. C++ Computation: Compute the Matrix for $T$ Relative to the Basis $B'$ for Polynomial Linear Operator

Find the matrix $T$ relative to the basis $B$, and then compute matrix for $T$ relative to the basis $B'$.

Suppose that $T : P_1 \to P_1$ is defined by $T(a_0 + a_1 x) = a_0 + a_1(x+1)$, and $B = \{\boldsymbol{p}_1, \boldsymbol{p}_2\}$ and $B' = \{\boldsymbol{q}_1, \boldsymbol{q}_2\}$, where

$$\begin{aligned}
\boldsymbol{p}_1 &= 6 + 3x \\
\boldsymbol{p}_2 &= 10 + 2x \\
\boldsymbol{q}_1 &= 2 \\
\boldsymbol{q}_2 &= 3 + 2x
\end{aligned}$$

**Solution:**

$$\begin{aligned}
T(\boldsymbol{p}_1) &= 6 + 3(x+1) = 9 + 3x = \frac{2}{3}\boldsymbol{p}_1 + \frac{1}{2}\boldsymbol{p}_2 \\
T(\boldsymbol{p}_2) &= 10 + 2(x+1) = 12 + 2x = \frac{2}{9}\boldsymbol{p}_1 + \frac{4}{3}\boldsymbol{p}_2
\end{aligned}$$

How we obtain the coefficient above is by using Gaussian elimination, we first construct a matrix $A$ with entries like this

$$A = [\boldsymbol{p}_1 | \boldsymbol{p}_2] = \begin{bmatrix} 6 & 10 \\ 3 & 2 \end{bmatrix}$$

with the $A\boldsymbol{x} = \boldsymbol{b}$, we use $T(\boldsymbol{p}_1)$ as $\boldsymbol{b}$ thus

$$\begin{bmatrix} 9 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

with Gaussian Elimination we will obtain $b_1 = \frac{2}{3}$ and $b_2 = \frac{1}{2}$, and now when $T(\boldsymbol{p}_2)$ as $\boldsymbol{b}$ we will have

$$\begin{bmatrix} 9 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

we will obtain $b_1 = -\frac{2}{9}$ and $b_2 = \frac{4}{3}$.

Hence

$$[T]_B = \begin{bmatrix} \frac{2}{3} & -\frac{2}{9} \\ \frac{1}{2} & \frac{4}{3} \end{bmatrix}$$

$$\begin{aligned}
\boldsymbol{q}_1 &= -\frac{2}{9}\boldsymbol{p}_1 + \frac{1}{3}\boldsymbol{p}_2 \\
\boldsymbol{q}_2 &= \frac{7}{9}\boldsymbol{p}_1 - \frac{1}{6}\boldsymbol{p}_2
\end{aligned}$$

To obtain the coefficients above, we will use Gaussian Elimination again, we still use matrix $A$ since we want the linear combination of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ thus we need not to change the matrix $A$, but this time we use $\boldsymbol{q}_1$ as $\boldsymbol{b}$ thus

$$\begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

with Gaussian Elimination we will obtain $b_1 = -\frac{2}{9}$ and $b_2 = \frac{1}{3}$, and now when $\boldsymbol{q}_2$ as $\boldsymbol{b}$ we will have

$$\begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

we will obtain $b_1 = \frac{7}{9}$ and $b_2 = -\frac{1}{6}$.

So the transition matrix from $B'$ to $B$ is

$$P = \begin{bmatrix} -\frac{2}{9} & \frac{7}{9} \\ \frac{1}{3} & -\frac{1}{6} \end{bmatrix}$$

Thus

$$P_{B \to B'} = P^{-1} = \begin{bmatrix} \frac{3}{4} & \frac{7}{2} \\ \frac{3}{2} & 1 \end{bmatrix}$$

and

$$[T]_{B'} = P_{B \to B'} [T]_B P_{B' \to B} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

The C++ code is using **GiNaC** combined with **Armadillo** to get the best of both world symbolic computation and solving numerical matrix computation.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;
using namespace GiNaC;

// Linear operator T: P_1 —> P_1
// Driver code
int main(int argc, char** argv)
{

        Digits = 5; // define maximum decimal digits
        symbol x("x"), xpow2("x^2"), a("a"), b("b"), x1("x1"), x2("x2");

        ex Tx = x + 1;
        // p1 = 6+3x ; p2 = 10+2x
        ex p1 = 6 + 3*x;
        ex p2 = 10 + 2*x;

        cout << "\n T(p1) = " << p1.subs(x==Tx) << endl;
        cout << "\n T(p2) = " << p2.subs(x==Tx) << endl;

        ex Tp1 = p1.subs(x==Tx);
```

```
ex Tp2 = p2.subs(x==Tx);

cout << "\n Coefficient of x^0 for T(p1) = " << Tp1.coeff(x,0) <<
    endl;
cout << "\n Coefficient of x^1 for T(p1) = " << Tp1.coeff(x,1) <<
    endl;
cout << "\n Coefficient of x^0 for T(p2) = " << Tp2.coeff(x,0) <<
    endl;
cout << "\n Coefficient of x^1 for T(p2) = " << Tp2.coeff(x,1) <<
    endl;

// To convert GiNaC:: ex to Double
double tp11 = ex_to<numeric>(Tp1.coeff(x,0)).to_double();
double tp12 = ex_to<numeric>(Tp1.coeff(x,1)).to_double();
double tp21 = ex_to<numeric>(Tp2.coeff(x,0)).to_double();
double tp22 = ex_to<numeric>(Tp2.coeff(x,1)).to_double();

vec bp1 = { tp11, tp12 };
vec bp2 = { tp21, tp22 };
vec v1 = { 6, 3 }; // from p1 = 6 + 3x
vec v2 = { 10, 2 }; // from p2 = 10 + 2x
mat A(2,2,fill::zeros);
A.col(0) = v1;
A.col(1) = v2;

mat X1;
mat X2;

X1 = solve(A,bp1,solve_opts::force_approx);
X2 = solve(A,bp2,solve_opts::force_approx);

mat T_B(2,2,fill::zeros);
T_B.col(0) = X1;
T_B.col(1) = X2;

cout << "\n [T]_{B} = \n" << T_B << endl;

// q1 = 2 ; q2 = 3 + 2x
ex q1 = 2;
ex q2 = 3 + 2*x;

// To convert GiNaC:: ex to Double
double q11 = ex_to<numeric>(q1.coeff(x,0)).to_double();
double q12 = ex_to<numeric>(q1.coeff(x,1)).to_double();
double q21 = ex_to<numeric>(q2.coeff(x,0)).to_double();
double q22 = ex_to<numeric>(q2.coeff(x,1)).to_double();

vec bq1 = { q11, q12 };
```

```
        vec bq2 = { q21, q22 };

        mat X_Bnew1;
        mat X_Bnew2;

        X_Bnew1 = solve(A,bq1,solve_opts::force_approx);
        X_Bnew2 = solve(A,bq2,solve_opts::force_approx);

        mat P(2,2,fill::zeros);
        P.col(0) = X_Bnew1;
        P.col(1) = X_Bnew2;

        cout << "\n Transition matrix from B' to B is P" << endl;
        cout << "\n P = \n" << P << endl;
        cout << "\n P^{-1} = \n" << arma::inv(P) << endl;

        mat T_Bnew;
        T_Bnew = arma::inv(P)*T_B*P;

        T_Bnew(0) = lround(T_Bnew(0));
        T_Bnew(1) = lround(T_Bnew(1));
        T_Bnew(2) = lround(T_Bnew(2));
        T_Bnew(3) = lround(T_Bnew(3));

        cout << "\n [T]_{B'} = P^{-1} * [T]_{B} * P = \n" << T_Bnew<< endl;

        return 0;
    }
```

**C++ Code 238:** *main.cpp "Compute matrix T relative to the basis B' for polynomial linear operator"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We can convert **ex** (expression / function in **GiNaC**) into double so we can then save the coefficient of polynomial into double as entries in vector or matrix in **Armadillo**.

```
        ...
        ...

        double tp11 = ex_to<numeric>(Tp1.coeff(x,0)).to_double();
        double tp12 = ex_to<numeric>(Tp1.coeff(x,1)).to_double();
        double tp21 = ex_to<numeric>(Tp2.coeff(x,0)).to_double();
```

```
        double tp22 = ex_to<numeric>(Tp2.coeff(x,1)).to_double();

        vec bp1 = { tp11, tp12 };
        vec bp2 = { tp21, tp22 };

        ...
```

```
*********************************************************************
********************GiNaC & Armadillo Codes**********************
*********************************************************************

T(p1) = 9+3*x

T(p2) = 12+2*x

Coefficient of x^0 for T(p1) = 9

Coefficient of x^1 for T(p1) = 3

Coefficient of x^0 for T(p2) = 12

Coefficient of x^1 for T(p2) = 2

[T]_{B} =
  0.6667  -0.2222
  0.5000   1.3333


Transition matrix from B' to B is P

P =
 -0.2222   0.7778
  0.3333  -0.1667


P^{-1} =
  0.7500   3.5000
  1.5000   1.0000


[T]_{B'} = P^{-1} * [T]_{B} * P =
  1.0000   1.0000
       0   1.0000
```

**Figure 23.229:** *The computation to compute $[T]_{B'}$ for polynomial $P_1$ linear operator.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Matrix T Relative to the Basis B′ for Linear Operator on P1 with GiNaC and Armadillo/main.cpp)****.*

## xi. C++ Computation: Find a Basis for $\mathbb{R}^2$ Relative to Which the Matrix $[T]_{B'}$ is Diagonal

Find a basis for $\mathbb{R}^2$ relative to which the matrix for $T$ is diagonal

$$T\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} 3x_1 - 2x_2 \\ -x_1 + 4x_2 \end{bmatrix}$$

**Solution:**

Let $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$, where

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$T(\boldsymbol{u}_1) = \begin{bmatrix} 3 \\ -1 \end{bmatrix}, \quad T(\boldsymbol{u}_2) = \begin{bmatrix} -2 \\ 4 \end{bmatrix}$$

The matrix for $T$ relative to the standard basis $B$ is

$$[T]_B = \begin{bmatrix} 3 & -2 \\ -1 & 4 \end{bmatrix}$$

The eigenvalues of $[T]_B$ are

$$\lambda_1 = 2$$
$$\lambda_2 = 5$$

with corresponding eigenvectors

$$\begin{bmatrix} -\frac{2\sqrt{5}}{5} \\ -\frac{\sqrt{5}}{5} \end{bmatrix}, \quad \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}$$

Then for

$$P = \begin{bmatrix} -\frac{2\sqrt{5}}{5} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{5}}{5} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

we have

$$P^{-1} = \begin{bmatrix} -\frac{\sqrt{5}}{3} & -\frac{\sqrt{5}}{3} \\ \frac{\sqrt{2}}{3} & \frac{2\sqrt{2}}{3} \end{bmatrix}$$

and

$$[T]_{B'} = P^{-1} * [T]_B * P = \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix}$$

Since $P$ represents the transition matrix from the basis $B'$ to the standard basis $B$, then

$$B' = \left(\begin{bmatrix} -\frac{2\sqrt{5}}{5} \\ -\frac{\sqrt{5}}{5} \end{bmatrix}, \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}\right)$$

is a basis for which $[T]_{B'}$ is diagonal.

The C++ code is only using **Armadillo** library for this section.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       // Define the standard basis vector B = {u1, u2}
       vec u1 = {1, 0};
       vec u2 = {0, 1};
       mat TB(2,2,fill::zeros);
       TB.col(0) = {3*u1(0) - 2*u1(1), -u1(0) + 4*u1(1)};
       TB.col(1) = {3*u2(0) - 2*u2(1), -u2(0) + 4*u2(1)};
       cx_mat eigvec;
       cx_vec eigval;


       mat I(2,2,fill::eye);
       cout <<"Matrix [T]_{B}:" << "\n" << TB <<endl;

       eig_gen(eigval, eigvec, TB); // Eigen decomposition of dense general
            square matrix

       cout << "Eigenvalues:" << endl;

       int n = 2;
       for (int i = 1; i <= n; i++)
       {
              cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                  imag(eigval[i-1]) << "i" << endl;
       }

       cout << endl;

       cout << "Eigenvectors:" << endl;
       for (int i = 1; i <= n; i++)
       {
              cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                  endl;
       }

       cout << "det([T]_{B}) = " << det(TB) << endl;
       cout << "rank([T]_{B}) = " << arma::rank(TB) << endl;
```

```
        mat P(2,2,fill::zeros);
        P.col(0) = real(eigvec.col(0));
        P.col(1) = real(eigvec.col(1));
        mat T_Bnew;
        T_Bnew = arma::inv(P)*TB*P;

        T_Bnew(0) = lround(T_Bnew(0));
        T_Bnew(1) = lround(T_Bnew(1));
        T_Bnew(2) = lround(T_Bnew(2));
        T_Bnew(3) = lround(T_Bnew(3));

        cout << "\nP = \n" << P << endl;
        cout << "P^{-1} = \n" << inv(P) << endl;
        cout << "P^{-1} * [T]_{B} * P = \n" << T_Bnew << endl;

        return 0;
    }
```

**C++ Code 239:** *main.cpp "Find a basis for R2 relative to matrix T that is diagonal"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We define the standard basis vectors $B = \{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ and then we construct a matrix $2 \times 2$, named $TB$ with **Armadillo** to represent $[T]_B$. We then input the formula for the first column and second column of $[T]_B$ that can be subtituted with the standard basis vectors $B$.

```
        ...

        vec u1 = {1, 0};
        vec u2 = {0, 1};
        mat TB(2,2,fill::zeros);
        TB.col(0) = {3*u1(0) - 2*u1(1), -u1(0) + 4*u1(1)};
        TB.col(1) = {3*u2(0) - 2*u2(1), -u2(0) + 4*u2(1)};

        ...
```

```
Matrix [T]_{B}:
   3.0000  -2.0000
  -1.0000   4.0000

Eigenvalues:
λ_1 = 2 + 0i
λ_2 = 5 + 0i

Eigenvectors:
x[λ_1] :
   -0.8944
  -0.4472

x[λ_2] :
    0.7071
  -0.7071

det([T]_{B}) = 10
rank([T]_{B}) = 2

P =
  -0.8944   0.7071
  -0.4472  -0.7071

P^{-1} =
  -0.7454  -0.7454
   0.4714  -0.9428

P^{-1} * [T]_{B} * P =
   2.0000        0
        0   5.0000
```

**Figure 23.230:** *The computation to compute $[T]_{B'}$ that will be diagonal in $\mathbb{R}^2$.* ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Basis for R2 Relative to make Matrix T Diagonal/main.cpp)****.*

## xii. C++ Computation: Find Eigenvalues and Bases for Eigenspaces for Polynomial $P_2$ Linear Operator

Let $T : P_2 \to P_2$ be defined by

$$T(a_0 + a_1x + a_2x^2) = (2a_0 - a_1 + 3a_2) + (4a_0 - 5a_1)x + (a_1 + 2a_2)x^2$$

(a) Find the eigenvalues of $T$.

(b) Find bases for the eigenspaces of $T$.

**Solution:**

(a) The matrix for $T$ with respect to the standard basis $B = \{1, x, x^2\}$ is

$$[T]_B = \begin{bmatrix} 2 & -1 & 3 \\ 4 & -5 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

The eigenvalues of $T$ are

$$\lambda_1 = -4$$
$$\lambda_2 = 0$$
$$\lambda_3 = 3$$

(b) The eigenspace of $[T]_B$ corresponding to $\lambda_1 = -4$ is

$$\boldsymbol{u}_1 = \begin{bmatrix} 0.2394 \\ 0.9577 \\ -0.1596 \end{bmatrix}$$

the eigenspace of $[T]_B$ corresponding to $\lambda_2 = 0$ is

$$\boldsymbol{u}_2 = \begin{bmatrix} -0.7454 \\ -0.5963 \\ 0.2981 \end{bmatrix}$$

the eigenspace of $[T]_B$ corresponding to $\lambda_3 = 3$ is

$$\boldsymbol{u}_3 = \begin{bmatrix} 0.8165 \\ 0.4082 \\ 0.4082 \end{bmatrix}$$

The C++ code is using **Armadillo** and **GiNaC** so we can gain the results in polynomial term and able to compute eigenvalues and eigenvectors easily.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <ginac/ginac.h>
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;
using namespace GiNaC;

// Linear operator T: P_2 —> P_2
// Driver code
int main(int argc, char** argv)
{

       Digits = 5; // define maximum decimal digits
       symbol x("x"), a("a"), b("b"), c("c");

       ex Tx = (2*a—b+3*c) + (4*a—5*b)*x + (b+2*c)*pow(x,2);

       cout << "\n T(x) = " << Tx << endl;

       for (int i = Tx.ldegree(x); i <= Tx.degree(x); ++i)
       {
              cout << " The x^" << i << "—coefficient is " << Tx.coeff(x,i
                  ) << endl;
       }

       // To obtain the coefficient of a,b,c for x^0, x^1 and x^2.
       ex Tx0 = Tx.coeff(x,0);
       ex Tx1 = Tx.coeff(x,1);
       ex Tx2 = Tx.coeff(x,2);

       /*
```

```
cout << "\n x^0 coeff a = " << Tx0.coeff(a,1) <<endl;
cout << "\n x^0 coeff b = " << Tx0.coeff(b,1) <<endl;
cout << "\n x^0 coeff c = " << Tx0.coeff(c,1) <<endl;

cout << "\n x^1 coeff a = " << Tx1.coeff(a,1) <<endl;
cout << "\n x^1 coeff b = " << Tx1.coeff(b,1) <<endl;
cout << "\n x^1 coeff c = " << Tx1.coeff(c,1) <<endl;

cout << "\n x^2 coeff a = " << Tx2.coeff(a,1)<<endl;
cout << "\n x^2 coeff b = " << Tx2.coeff(b,1)<<endl;
cout << "\n x^2 coeff c = " << Tx2.coeff(c,1)<<endl;
*/

// Convert the expression to double
double tb11 = ex_to<numeric>(Tx0.coeff(a,1)).to_double();
double tb12 = ex_to<numeric>(Tx0.coeff(b,1)).to_double();
double tb13 = ex_to<numeric>(Tx0.coeff(c,1)).to_double();
double tb21 = ex_to<numeric>(Tx1.coeff(a,1)).to_double();
double tb22 = ex_to<numeric>(Tx1.coeff(b,1)).to_double();
double tb23 = ex_to<numeric>(Tx1.coeff(c,1)).to_double();
double tb31 = ex_to<numeric>(Tx2.coeff(a,1)).to_double();
double tb32 = ex_to<numeric>(Tx2.coeff(b,1)).to_double();
double tb33 = ex_to<numeric>(Tx2.coeff(c,1)).to_double();

// Create matrix [T]_{B} with Armadillo
vec tb1 = {tb11, tb21, tb31};
vec tb2 = {tb12, tb22, tb32};
vec tb3 = {tb13, tb23, tb33};
arma::mat TB(3,3,fill::zeros);
TB.col(0) = tb1;
TB.col(1) = tb2;
TB.col(2) = tb3;

cout << "\n [T]_{B} = \n" << TB << endl;

// Compute the eigenvalues and eigenvectors with Armadillo
cx_mat eigvec;
cx_vec eigval;
eig_gen(eigval, eigvec, TB); // Eigen decomposition of dense general
     square matrix
cout << "Eigenvalues:" << endl;

int n = 3;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}
```

```
        cout << endl;
        cout << "Eigenvectors / Basis for the eigenspace of [T]_{B}
            correspond to certain l:" << endl;
        for (int i = 1; i <= n; i++)
        {
               cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                   endl;
        }
        cout << "(The eigenspace of [T]_{B} corresponding to l has the basis
             as the eigenvector corresponding to that l)" << endl;

        vec u1 = real(eigvec.col(0));
        vec u2 = real(eigvec.col(1));
        vec u3 = real(eigvec.col(2));

        cout << "\n u_{1} = \n" << u1 << endl;
        cout << "\n u_{2} = \n" << u2 << endl;
        cout << "\n u_{3} = \n" << u3 << endl;

        // Construct the basis B' = { p_{1}, p_{2}, p_{3} } with GiNaC
        ex p1 = 1*u1(0) + x*u1(1) + pow(x,2)*u1(2);
        ex p2 = 1*u2(0) + x*u2(1) + pow(x,2)*u2(2);
        ex p3 = 1*u3(0) + x*u3(1) + pow(x,2)*u3(2);

        cout << "\n p_{1} = " << p1 << endl;
        cout << "\n p_{2} = " << p2 << endl;
        cout << "\n p_{3} = " << p3 << endl;

        cout << "\n The eigenspace of T corresponding to l = " << real(
            eigval[0]) << " has the basis " << endl;
        cout << "\n p_{1} = " << p1 << endl;

        cout << "\n The eigenspace of T corresponding to l = " << real(
            eigval[1]) << " has the basis " << endl;
        cout << "\n p_{2} = " << p2 << endl;

        cout << "\n The eigenspace of T corresponding to l = " << real(
            eigval[2]) << " has the basis " << endl;
        cout << "\n p_{3} = " << p3 << endl;

        return 0;
    }
```

**C++ Code 240:** *main.cpp "Find Eigenvalues and Bases for Eigenspaces for Polynomial Linear Operator"*

To compile it, type:
**g++ -o main main.cpp -larmadillo -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
 T(x) = -b-(5*b-4*a)*x+3*c+x^2*(b+2*c)+2*a
 The x^0-coefficient is -b+3*c+2*a
 The x^1-coefficient is -5*b+4*a
 The x^2-coefficient is b+2*c

 [T]_{B} =
   2.0000  -1.0000   3.0000
   4.0000  -5.0000        0
        0   1.0000   2.0000

Eigenvalues:
λ_1 = -4 + 0i
λ_2 = 4.31946e-16 + 0i
λ_3 = 3 + 0i

Eigenvectors / Basis for the eigenspace of [T]_{B} correspond to certain λ:
x[λ_1] :
    0.2394
   0.9577
  -0.1596

x[λ_2] :
   -0.7454
  -0.5963
   0.2981

x[λ_3] :
    0.8165
   0.4082
   0.4082
```

**Figure 23.231:** *The computation (part 1) to compute eigenvalues and bases for eigenspaces of the linear operator $T : P_2 \to P_2$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Bases for Eigenspaces for Linear Operator on P2 with GiNaC and Armadillo Exercise 13/main.cpp**).*

```
(The eigenspace of [T]_{B} corresponding to λ has the basis as the eigenvector corresponding to that λ)

 u_{1} =
   0.2394
   0.9577
  -0.1596


 u_{2} =
  -0.7454
  -0.5963
   0.2981


 u_{3} =
   0.8165
   0.4082
   0.4082


 p_{1} = 0.23942606-(0.15961738)*x^2+(0.95770425)*x

 p_{2} = -0.74535596+(0.2981424)*x^2-(0.5962848)*x

 p_{3} = 0.8164966+(0.4082483)*x^2+(0.4082483)*x

 The eigenspace of T corresponding to λ = -4 has the basis

 p_{1} = 0.23942606-(0.15961738)*x^2+(0.95770425)*x

 The eigenspace of T corresponding to λ = 4.31946e-16 has the basis

 p_{2} = -0.74535596+(0.2981424)*x^2-(0.5962848)*x

 The eigenspace of T corresponding to λ = 3 has the basis

 p_{3} = 0.8164966+(0.4082483)*x^2+(0.4082483)*x
```

**Figure 23.232:** *The computation (part 2) to compute eigenvalues and bases for eigenspaces of the linear operator $T : P_2 \to P_2$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Eigenvalues and Bases for Eigenspaces for Linear Operator on P2 with GiNaC and Armadillo Exercise 13/main.cpp**).*

# XVI. Vandermonde Matrix

Fix a sequence of numbers $\{x_1, x_2, \ldots, x_m\}$. If $p$ and $q$ are polynomials of degree $< n$ and $\alpha$ is a scalar, then $p + q$ and $\alpha p$ are also polynomials of degree $< n$. Moreover, the values of these polynomials at the points $x_i$ satisfy the following linearity properties

$$(p+q)(x_i) = p(x_i) + q(x_i)$$
$$(\alpha p)(x_i) = \alpha(p(x_i))$$

Thus the map from vectors of coefficients of polynomials $p$ of degree $< n$ to vectors $(p(x_1), p(x_2), \ldots, p(x_m))$ of sampled polynomial values is linear. Any linear map can be expressed as multiplication by a matrix. It is expressed by an $m \times n$ Vandermonde matrix

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^{n-1} \end{bmatrix}$$

If $c$ is the column vector of coefficients of $p$,

$$c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}, \quad p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}$$

then the product of $Ac$ gives the sampled polynomial values. That is, for each $i$ from 1 to $m$, we have

$$(Ac)_i = c_0 + c_1 x_i + c_2 x_i^2 + \cdots + c_{n-1} x_i^{n-1} = p(x_i) \tag{23.279}$$

## i. C++ Computation: Polynomial Interpolation with Vandermonde Matrix and Gauss-Jordan Elimination

If we have several points in $\mathbb{R}^2$ that we want to interpolate, the Vandermonde matrix can be used for this kind of problem.

Suppose you have conducted an experiment and have data like this

$$(x_1, y_1),\ (x_2, y_2),\ \ldots,\ (x_n, y_n)$$

with $i = 1, 2, \ldots, n$, and $x_i$ represents the independent variable, $y_i$ is the dependent variable.

Every physics phenomena or engineering system is described by its underlying mathematical model (equation describing the relation between the independent and dependent variable).

We can fit a polynomial curve through all the data points to find the relation between the independent and dependent variable. Once you get the polynomial function for the curve you can calculate the value of the dependent variable at any data point.

The procedure for getting the function is called polynomial curve-fitting. When you calculate the value of the function between two data points it is called polynomial interpolation.

**Theorem 23.151: Polynomial Interpolation**

Given any $n$ points in the $xy$-plane ($\mathbb{R}^2$) that have distinct $x$-coordinates, there is a unique polynomial of degree $n-1$ or less whose graph passes through those points.

In general the polynomial function can be defined as

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \tag{23.280}$$

To obtain the function, we need to compute the coefficients of the polynomial, that is $a_0, a_1, \ldots, a_n$. For this we will create a system of $n$ linear equations:

$$\begin{array}{llllll}
a_0 & + a_1 x_1 & + a_2 x_1^2 & + \ldots & + a_{n-1} x_1^{n-1} & = y_1 \\
a_0 & + a_1 x_2 & + a_2 x_2^2 & + \ldots & + a_{n-1} x_2^{n-1} & = y_2 \\
a_0 & + a_1 x_3 & + a_2 x_3^2 & + \ldots & + a_{n-1} x_3^{n-1} & = y_3 \\
\vdots & & & & & \\
a_0 & + a_1 x_n & + a_2 x_n^2 & + \ldots & + a_{n-1} x_n^{n-1} & = y_n
\end{array}$$

In matrix form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The coefficient matrix in the above is known as the Vandermonde Matrix, with the elements in the row of a Vandermonde Matrix are in geometric progression.

The solution of the above linear system will give us the coefficients of the polynomial function $(a_0, a_1, \ldots, a_{n-1})$.

We can find the solution with two methods:

1. Find the inverse of the Vandermonde Matrix then multiply it with the vector of the dependent variable.
2. Make an augmented matrix containing the Vandermonde Matrix and the last column will be the vector of the dependent variable, then use Gauss-Jordan elimination to obtain the reduced row echelon form.

If we want to fit polynomial of higher degree we will need a larger matrix, but a larger matrix will lead to accumulation of round-off errors and the matrix will become ill conditioned. Henve, this method of polynomial interpolation with Vandermonde Matrix is not used beyond cubic polynomials.

Now for the example:
Find a cubic polynomial whose graph passes through the points

$$(1,3),\ (2,-2),\ (3,-5),\ (4,0)$$

Since there ar efour points, we will use an interpolating polynomial of degree $n = 3$. Denote this polynomial by

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

and denote the $x$- and $y$-coordinates of the given points by

$$\begin{aligned}(x_1, y_1) &= (1, 3)\\ (x_2, y_2) &= (2, -2)\\ (x_3, y_3) &= (3, -5)\\ (x_4, y_4) &= (4, 0)\end{aligned}$$

Thus, the augmented matrix for the linear system in the unknowns $a_0, a_1, a_2, a_3$ is

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & y_1 \\ 1 & x_2 & x_2^2 & x_2^3 & y_2 \\ 1 & x_3 & x_3^2 & x_3^3 & y_3 \\ 1 & x_4 & x_4^2 & x_4^3 & y_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 3 \\ 1 & 2 & 4 & 8 & -2 \\ 1 & 3 & 9 & 27 & -5 \\ 1 & 4 & 16 & 64 & 0 \end{bmatrix}$$

The reduced echelon form of the augmented matrix above is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & -5 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

it follows that $a_0 = 4,\ a_1 = 3,\ a_2 = -5,\ a_3 = 1$. Thus, the interpolating polynomial is

$$p(x) = 4 + 3x - 5x^2 + x^3$$

```
#include <iostream>
#include <bits/stdc++.h>
#include "gnuplot-iostream.h"

using namespace std;

#define M 10
const int N = 4;

// Function to print the matrix
void PrintMatrix(float a[][M], int n)
{
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j <= n; j++)
                cout << a[i][j] << " ";
                cout << endl;
        }
}

// function to reduce matrix to reduced
```

```
// row echelon form.
int PerformOperation(float a[][M], int n)
{
	int i, j, k = 0, c, flag = 0, m = 0;
	float pro = 0;

	// Performing elementary operations
	for (i = 0; i < n; i++)
	{
		if (a[i][i] == 0)
		{
			c = 1;
			while ((i + c) < n && a[i + c][i] == 0)
			c++;
			if ((i + c) == n)
			{
				flag = 1;
				break;
			}
			for (j = i, k = 0; k <= n; k++)
			swap(a[j][k], a[j+c][k]);
		}

		for (j = 0; j < n; j++)
		{
			// Excluding all i == j
			if (i != j)
			{
				// Converting Matrix to reduced row
				// echelon form(diagonal matrix)
				float pro = a[j][i] / a[i][i];
				for (k = 0; k <= n; k++)
				a[j][k] = a[j][k] - (a[i][k]) * pro;
			}
		}
	}
	return flag;
}

// Function to print the desired result
// if unique solutions exists, otherwise
// prints no solution or infinite solutions
// depending upon the input given.
void PrintResult(float a[][M], int n, int flag)
{
	cout << "The solution/s : ";

	if (flag == 2)
```

```
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "x(" << i << ") = "<< a[i][n] / a[i][i] << ", ";
        }
}

// To check whether infinite solutions
// exists or no solution exists
int CheckConsistency(float a[][M], int n, int flag)
{
        int i, j;
        float sum;

        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < n; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + a[i][j];
                if (sum == a[i][j])
                flag = 2;
        }
        return flag;
}

// Driver code
int main()
{
        float x1 = 1;
        float x2 = 2;
        float x3 = 3;
        float x4 = 4;
        float y1 = 3;
        float y2 = -2;
        float y3 = -5;
```

```
        float y4 = 0;

        float a[M][M] = {{ 1, x1, x1*x1,x1*x1*x1, y1 },
               { 1, x2, x2*x2, x2*x2*x2, y2 },
               { 1, x3, x3*x3, x3*x3*x3, y3 },
               { 1, x4, x4*x4, x4*x4*x4, y4 }};

        // Printing matrix A
        cout << "Matrix A : " << endl;
        int n1 = 4;
        for (int i = 0; i < n1; ++i)
        {
               for (int j = 0; j < n1; ++j)
               {
                      cout << a[i][j]<< ' ';
               }
               cout << endl;
        }

        // Order of Matrix(n)
        int n = 4, flag = 0;

        // Performing Matrix transformation
        flag = PerformOperation(a, n);

        if (flag == 1)
        flag = CheckConsistency(a, n, flag);

        // Printing Final Matrix
        cout << endl;
        cout << "Final Augmented Matrix is : " << endl;
        PrintMatrix(a, n);
        cout << endl;

        // Printing Solutions(if exist)
        cout << "Solutions for Matrix A : " << endl;
        PrintResult(a,n, flag);
        cout << endl;

        // Plot the points and the polynomial interpolation with Gnuplot
        std::vector<std::pair<double, double>> xy_pts;
        float B[N];
        for (int i = 0; i < N; ++i)
        {
               for (int j = 0; j < N; ++j)
               {
                      B[i] = a[i][n] / a[i][i] ;
               }
```

```
                cout << B[i] << endl;
                xy_pts.emplace_back(i, B[i]);
        }
        Gnuplot gp;

        gp << "set xrange [-1:5]\n";
        gp << "f(x) = a + b*x + c*x*x + d*x*x*x\n";
        gp << "fit f(x) '-' via a,b,c,d\n";
        gp.send1d(xy_pts);
        gp << "plot '-' with points title 'input', f(x) with lines title '
            fit'\n";
        gp.send1d(xy_pts);

        return 0;
    }
```

**C++ Code 241:** *main.cpp "Polynomial interpolation with Vandermonde Matrix and Gauss-Jordan Elimination"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- Under the **int main()** you can replace the value for $x_1, x_2, x_3, x_4$ and its corresponding dependent variable $y_1, y_2, y_3, y_4$ for cubic polynomial, for quadratic polynomial you only need to make a few changes for the code below.
  Afterwards, we will first we assign the **xy_pts** vector with the variable of the solution **a[i][n] / a[i][i]**, then call gnuplot to plot the points along with the polynomial interpolation / the curve fitting.

```
int main()
{
        float x1 = 1;
        float x2 = 2;
        float x3 = 3;
        float x4 = 4;
        float y1 = 3;
        float y2 = -2;
        float y3 = -5;
        float y4 = 0;

        float a[M][M] = {{ 1, x1, x1*x1,x1*x1*x1, y1 },
                { 1, x2, x2*x2, x2*x2*x2, y2 },
                { 1, x3, x3*x3, x3*x3*x3, y3 },
                { 1, x4, x4*x4, x4*x4*x4, y4 }};
```

```
// Printing matrix A
cout << "Matrix A : " << endl;
int n1 = 4;
for (int i = 0; i < n1; ++i)
{
       for (int j = 0; j < n1; ++j)
       {
              cout << a[i][j]<< ' ';
       }
       cout << endl;
}

// Order of Matrix(n)
int n = 4, flag = 0;

// Performing Matrix transformation
flag = PerformOperation(a, n);

if (flag == 1)
flag = CheckConsistency(a, n, flag);

// Printing Final Matrix
cout << endl;
cout << "Final Augmented Matrix is : " << endl;
PrintMatrix(a, n);
cout << endl;

// Printing Solutions(if exist)
cout << "Solutions for Matrix A : " << endl;
PrintResult(a,n, flag);
cout << endl;

// Plot the points and the polynomial interpolation with
     Gnuplot
std::vector<std::pair<double, double>> xy_pts;
float B[N];
for (int i = 0; i < N; ++i)
{
       for (int j = 0; j < N; ++j)
       {
              B[i] = a[i][n] / a[i][i] ;
       }
       cout << B[i] << endl;
       xy_pts.emplace_back(i, B[i]);
}
Gnuplot gp;
```

```
                    gp << "set xrange [-1:5]\n";
                    gp << "f(x) = a + b*x + c*x*x + d*x*x*x\n";
                    gp << "fit f(x) '-' via a,b,c,d\n";
                    gp.send1d(xy_pts);
                    gp << "plot '-' with points title 'input', f(x) with
                        lines title 'fit'\n";
                    gp.send1d(xy_pts);

                    return 0;
            }
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Polynomial Interpolation ]# ./a.out
Matrix A :
1 1 1 1
1 2 4 8
1 3 9 27
1 4 16 64

Final Augmented Matrix is :
1 0 0 0 4
0 1 0 0 3
0 0 2 0 -10
0 0 0 6 6

Solutions for Matrix A :
The solution/s : x(0) = 4,  x(1) = 3,  x(2) = -5,  x(3) = 1,
```

**Figure 23.233:** *The computation to find reduced row echelon form with Gauss-Jordan elimination for polynomial interpolation using Vandermonde Matrix with C++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Polynomial Interpolation/main.cpp**).*

**Figure 23.234:** *The graph of the polynomial $p(x) = 4 + 3x - 5x^2 + x^3$ that is plotted with Gnuplot from the C++ code directly (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Polynomial Interpolation/main.cpp**).*

# XVII. Gaussian Elimination with Backward Substitution

Gaussian Elimination takes a very important role in the Linear Algebra world, we want to take a deeper insight into this to gain better understanding after all the previous sections. The Gaussian Elimination algorithm is majorly about performing a sequence of operations on the rows of the matrix. What we would like to keep in mind while performing these operations is that we want to convert the matrix into an upper triangular matrix in row echelon form. Gaussian elimination is the best for computing determinant of a square matrix. It is simpler than Gauss-Jordan elimination. The operations can be:

1. Swapping two rows
2. Multiplying a row by a non-zero scalar
3. Adding to one row a multiple of another

The process:

1. Forward elimination: reduction to row echelon form. Using it one can tell whether there are no solutions, or unique solution, or infinitely many solutions.
2. Back substitution: further reduction to reduced row echelon form.

Algorithm:

1. Partial pivoting: Find the $k$th pivot by swapping rows, to move the entry with the largest absolute value to the pivot position. This imparts computational stability to the algorithm.
2. For each row below the pivot, calculate the factor $f$ which makes the $k$th entry zero, and for every element in the row subtract the $f$th multiple of the corresponding element in the $k$th row.
3. Repeat above steps for each unknown. We will be left with a partial reduced echelon form matrix.

Consider this linear system

$$\begin{array}{lllllll} E_1: & a_{11}x_1 & +\,a_{12}x_2 & +\dots & +a_{1n}x_n & = b_1 \\ E_2: & a_{21}x_1 & +\,a_{22}x_2 & +\dots & +\,a_{2n}x_n & = b_2 \\ & \vdots & \vdots & \dots & \vdots & \vdots \\ E_n: & a_{n1}x_1 & +\,a_{n2}x_2 & +\dots & +\,a_{nn}x_n & = b_n \end{array}$$

for $x_1, x_2, \dots, x_n$, given the constants $a_{ij}$, for each $i, j = 1, 2, \dots, n$ and $b_i$ for each $i = 1, 2, \dots, n$.

First, form the augmented matrix $\tilde{A}$

$$\tilde{A} = [A, \boldsymbol{b}] = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n,n+1} \end{bmatrix}$$

where $A$ denotes the matrix formed by the coefficients. The entries in the $(n+1)$st column are the values of $\boldsymbol{b}$, that is

$$a_{i,n+1} = b_i$$

for each $i = 1, 2, \ldots, n$.

Provided $a_{11} \neq 0$, the operations corresponding to

$$\left(E_j - \frac{a_{j1}}{a_{11}} E_1\right) \to (E_j)$$

are performed for each $j = 2, 3, \ldots, n$ to eliminate the coefficient of $x_1$ in each of these rows. Although the entries in rows $2, 3, \ldots, n$ are expected to change, for ease of notation we again denote the entry in the $i$th row and the $j$th column by $a_{ij}$.

We folow a sequential procedure for $i = 2, 3, \ldots, n-1$ and perform operation

$$\left(E_j - \frac{a_{ji}}{a_{ii}} E_i\right) \to (E_j)$$

for each $j = i+1, i+2, \ldots, n$, provided $a_{ii} \neq 0$.

This eliminates $x_i$ in each row below the $i$th for all values of $i = 1, 2, \ldots, n-1$. The resulting matrix has the form

$$\tilde{A}' = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} & a_{1,n+1} \\ 0 & a_{22} & \ldots & a_{2n} & a_{2,n+1} \\ \vdots & \ddots & & \vdots & \vdots \\ 0 & \ldots & 0 & a_{nn} & a_{n,n+1} \end{bmatrix}$$

where, except in the first row, the values of $a_{ij}$ are not expected to agree with those in the original matrix $\tilde{A}$. The matrix $\tilde{A}'$ represnts a linear system with the same solution set as the original system $\tilde{A}$. SInce the new linear system is triangular

$$\begin{array}{llll} a_{11}x_1 & + a_{12}x_2 & + \ldots & + a_{1n}x_n & = a_{1,n+1} \\ & + a_{22}x_2 & + \ldots & + a_{2n}x_n & = a_{2,n+1} \\ & \ddots & & \vdots & \vdots \\ & & & a_{nn}x_n & = a_{n,n+1} \end{array}$$

backward substitution can be performed. Solving the $n$th equation for $x_n$ gives

$$x_n = \frac{a_{n,n+1}}{a_{nn}}$$

Solving the $(n-1)$st equation for $x_{n-1}$ and using $x_n$ yields

$$x_{n-1} = \frac{a_{n-1,n+1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

Continuing this process, we obtain

$$x_i = \frac{a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}}$$

for each $i = n-1, n-2, \ldots, 2, 1$.

The Gaussian elimination procedure can be presented more precisely, although more intricately, by forming a sequence of augmented matrices $\tilde{A}^1, \tilde{A}^2, \ldots, \tilde{A}^n$, where $\tilde{A}^1$ is the matrix $\tilde{A}$ and $\tilde{A}^{(k)}$, for each $k = 2, 3, \ldots, n$ has entries $a_{ij}^{(k)}$, where:

$$a_{ij}^{(k)} = \begin{cases} a_{ij}^{(k-1)}, & \text{when } i = 1, 2, \ldots, k-1 \text{ and } j = 1, 2, \ldots, n+1 \\ 0, & \text{when } i = k, k+1, \ldots, n \text{ and } j = 1, 2, \ldots, k-1 \\ a_{ij}^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} a_{k-1,j}^{(k-1)}, & \text{when } i = k, k+1, \ldots, n \text{ and } j = k, k+1, \ldots, n+1 \end{cases}$$

Thus,

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1,k-1}^{(1)} & a_{1,k}^{(1)} & \cdots & a_{1n}^{(1)} & a_{1,n+1}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} & a_{2,n+1}^{(2)} \\ \vdots & \ddots & \ddots & & \vdots & \vdots & & \vdots & \vdots \\ \vdots & & \ddots & \ddots & \vdots & \vdots & & \vdots & \vdots \\ \vdots & & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} & a_{k-1,n+1}^{(k-1)} \\ \vdots & & & & 0 & a_{k,k}^{(k)} & \cdots & a_{k,n}^{(k)} & a_{k,n+1}^{(k)} \\ \vdots & & & & \vdots & \vdots & & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} & a_{n,n+1}^{(k)} \end{bmatrix}$$

represents the equivalent linear system for which the variable $x_{k-1}$ has just been eliminated from equations $E_k, E_{k+1}, \ldots, E_n$.

The procedure will fail if one of the elements $a_{11}^{(1)}, a_{22}^{(2)}, a_{33}^{(3)}, \ldots, a_{n-1,n-1}^{(n-1)}, a_{nn}^{(n)}$ is zero because the step

$$\left( E_i - \frac{a_{i,k}^{(k)}}{a_{kk}^{(k)}} E_k \right) \to E_i$$

either cannot be performed (this occurs if one of $a_{11}^{(1)}, a_{22}^{(2)}, a_{33}^{(3)}, \ldots, a_{n-1,n-1}^{(n-1)}$ is zero), or the backward substitution cannot be accomplished (in the case $a_{nn}^{(n)} = 0$). The system may still have a solution, but the technique for finding the solution must be altered.

## i. C++ Computation: Gaussian Elimination with Backward Substitution

Taken from exercise set 6.1 number 2 part (b) from [2].

Use Gaussian elimination with backward substitution to solve the following linear systems

$$\begin{array}{rrrrl} 4x_1 & +\,x_2 & +\,2x_3 & = & 9 \\ 2x_1 & +\,4x_2 & -\,x_3 & = & -5 \\ x_1 & +\,x_2 & -\,3x_3 & = & -9 \end{array}$$

The exact solution is $x_1 = 1,\ x_2 = -1,\ x_3 = 3$. We are going to compute with C++ and confirm this solution.

The code is taken from:
**https://www.geeksforgeeks.org/gaussian-elimination/**
as usual we modify a bit.

```
#include<bits/stdc++.h>
using namespace std;

#define N 3 // Number of unknowns

// function to reduce matrix to r.e.f. Returns a value to
// indicate whether matrix is singular or not
int forwardElim(double mat[N][N+1]);

// function to calculate the values of the unknowns
void backSub(double mat[N][N+1]);

// function to get matrix content
void gaussianElimination(double mat[N][N+1])
{
       /* reduction into r.e.f. */
       int singular_flag = forwardElim(mat);

       /* if matrix is singular */
       if (singular_flag != -1)
       {
              printf("Singular Matrix.\n");

              /* if the RHS of equation corresponding to
              zero row is 0, * system has infinitely
              many solutions, else inconsistent*/
              if (mat[singular_flag][N])
              printf("Inconsistent System.");
              else
              printf("May have infinitely many solutions.");

              return;
```

```
        }

        /* get solution to system and print it using
        backward substitution */
        backSub(mat);
}

// function for elementary operation of swapping two rows
void swap_row(double mat[N][N+1], int i, int j)
{
        //printf("Swapped rows %d and %d\n", i, j);

        for (int k=0; k<=N; k++)
        {
                double temp = mat[i][k];
                mat[i][k] = mat[j][k];
                mat[j][k] = temp;
        }
}

// function to print matrix content at any stage
void print(double mat[N][N+1])
{
        for (int i=0; i<N; i++, printf("\n"))
        {
                for (int j=0; j<=N; j++)
                {
                        printf("%lf ", mat[i][j]);
                }
                printf("\n");
        }
}

void PrintMatrix(double mat[N][N+1], int n)
{
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j <= n; j++)
                cout << setw(6) << mat[i][j] << "\t";
                cout << endl;
        }
        cout<<endl;
}

// function to reduce matrix to r.e.f.
int forwardElim(double mat[N][N+1])
{
        for (int k=0; k<N; k++)
```

```
        {
                // Initialize maximum value and index for pivot
                int i_max = k;
                int v_max = mat[i_max][k];


                /* find greater amplitude for pivot if any */
                for (int i = k+1; i < N; i++)
                {
                        if (abs(mat[i][k]) > v_max)
                        v_max = mat[i][k], i_max = i;
                }
                /* if a principal diagonal element is zero,
                * it denotes that matrix is singular, and
                * will lead to a division-by-zero later. */
                if (!mat[k][i_max])
                return k; // Matrix is singular

                /* Swap the greatest value row with current row */
                if (i_max != k)
                swap_row(mat, k, i_max);

                for (int i=k+1; i<N; i++)
                {
                        /* factor f to set current row kth element to 0,
                        * and subsequently remaining kth column to 0 */
                        double f = mat[i][k]/mat[k][k];

                        /* subtract fth multiple of corresponding kth
                        row element*/
                        for (int j=k+1; j<=N; j++)
                        {
                                mat[i][j] -= mat[k][j]*f;
                                /* filling lower triangular matrix with zeros*/
                        }
                        mat[i][k] = 0;
                }
                PrintMatrix(mat,N); //for matrix state
        }
        //print(mat); //for matrix state
        return -1;
}

// function to calculate the values of the unknowns
void backSub(double mat[N][N+1])
{
        double x[N]; // An array to store solution
```

```
        /* Start calculating from last equation up to the
        first */
        for (int i = N-1; i >= 0; i--)
        {
                /* start with the RHS of the equation */
                x[i] = mat[i][N];

                /* Initialize j to i+1 since matrix is upper
                triangular*/
                for (int j=i+1; j<N; j++)
                {
                        /* subtract all the lhs values
                        * except the coefficient of the variable
                        * whose value is being calculated */
                        x[i] -= mat[i][j]*x[j];
                }

                /* divide the RHS by the coefficient of the
                unknown being calculated */
                x[i] = x[i]/mat[i][i];
        }

        printf("\nSolution for the system:\n");
        for (int i=0; i<N; i++)
        printf("%lf\n", x[i]);
}

// Driver program
int main()
{
        /* input matrix */
        double mat[N][N+1] = {{4.0, 1.0,2.0, 9.0},
                {2.0, 4.0, -1.0, -5.0},
                {1.0, 1.0, -3.0, -9.0}};

        //print(mat);
        cout << endl;
        cout << "Matrix A:" << endl;
        PrintMatrix(mat,N);
        cout << endl;
        cout << "The Gaussian Elimination process:" << endl;
        gaussianElimination(mat);

        return 0;
}
```

**C++ Code 242:** *main.cpp "Gaussian Elimination"*

To compile it, type:

**g++ -o result main.cpp**
**./result**

or with the Makefile, type:
**make**
**./main**

To check the compiling time you can type:
**time make**

Explanation for the codes:

- The function **void backSub(double mat[N][N+1])** is to do the backward substitution to obtain the solution for the linear system
- We use **void PrintMatrix(double mat[N][N+1], int n)** to make the matrix printed more nicely with better margin between each indices / entries.

The time to compile the source code into binary file is 10 seconds with Dell Precision laptop, CPU Intel Core i7 CPU Q 720 @ 1.60$GHz \times 8$ and RAM of 16 GB.

```
mbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination ]# time make
g++    -c -o main.o main.cpp
g++ -o main -ggdb main.o -lstdc++

real    0m10.686s
user    0m10.208s
sys     0m0.478s
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination ]# ./main

Matrix A:
     4       1       2       9
     2       4      -1      -5
     1       1      -3      -9


The Gaussian Elimination process:
     4       1       2       9
     0     3.5      -2    -9.5
     0    0.75    -3.5  -11.25

     4       1       2       9
     0     3.5      -2    -9.5
     0       0  -3.07143        -9.21429

     4       1       2       9
     0     3.5      -2    -9.5
     0       0  -3.07143        -9.21429


Solution for the system:
1.000000
-1.000000
3.000000
```

**Figure 23.235:** *The computation to get the solution for $A\boldsymbol{x} = \boldsymbol{b}$ with Gaussian Elimination with backward substitution (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination/main.cpp**).*

## ii. C++ Computation: Gaussian Elimination for Symbolic Matrix

Taken from exercise set 4.2 number 13 from [11].

Determine whether the following polynomials span $P_2$

$$\boldsymbol{p}_1 = 1 - x + 2x^2, \quad \boldsymbol{p}_2 = 3 + x$$

$$\boldsymbol{p}_3 = 5 - x + 4x^2, \quad \boldsymbol{p}_4 = -2 - 2x + 2x^2$$

**Solution:**
Let $\boldsymbol{p} = a_0 + a_1 x + a_2 x^2$ be an arbitrary polynomial in $P_2$. If $k_1\boldsymbol{p}_1 + k_2\boldsymbol{p}_2 + k_3\boldsymbol{p}_3 + k_4\boldsymbol{p}_4 = \boldsymbol{p}$, then

$$(k_1 + 3k_2 + 5k_3 - 2k_4) + (-k_1 + k_2 - k_3 - 2k_4)x + (2k_1 + 4k_3 + 2k_4)x^2 = a_0 + a_1 x + a_2 x^2$$

or

$$\begin{array}{rrrrl} k_1 & +\,3k_2 & +\,5k_3 & -\,2k_4 & = a_0 \\ -k_1 & +\,3k_2 & -\,k_3 & -\,2k_4 & = a_1 \\ 2k_1 & + & +\,4k_3 & +\,2k_4 & = a_2 \end{array}$$

In matrix form:

$$\begin{bmatrix} 1 & 3 & 5 & -2 & a_0 \\ -1 & 1 & -1 & -2 & a_1 \\ 2 & 0 & 4 & 2 & a_2 \end{bmatrix}$$

Reducing the matrix above to echelon form leads to In matrix form:

$$\begin{bmatrix} 1 & 3 & 5 & -2 & a_0 \\ 0 & 1 & 1 & -1 & \frac{1}{4}a_0 + \frac{1}{4}a_1 \\ 0 & 0 & 0 & 0 & -\frac{1}{2}a_0 + \frac{3}{2}a_1 + a_2 \end{bmatrix}$$

The system has a solution if

$$-\frac{1}{2}a_0 + \frac{3}{2}a_1 + a_2 = 0$$

or

$$a_0 = 3a_1 + 2a_2$$

This restriction means that the span of $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$, and $\boldsymbol{p}_4$ is not all of $P_3$, thus they do not span $P_2$.

In C++, we are used to do Gaussian Elimination for numerical matrix, the matrix entries are all numerical value / numbers only, but this time, what if there are symbolic entries inside? Thus in this case we are going to use the **SymbolicC++** library to help us do the symbolic computation.

```
#include<bits/stdc++.h>
#include<iostream>
#include "symbolicc++.h"
#include<vector>
using namespace std;

#define R 3 // number of rows
#define C 5 // number of columns

// Driver program
```

```
int main()
{
        Symbolic a0("a0");
        Symbolic a1("a1");
        Symbolic a2("a2");

        // Construct a symbolic matrix of size 3 X 5
        Matrix<Symbolic> B_mat(3,5);
        B_mat[0][0] = 1;  B_mat[0][1] = 3; B_mat[0][2] = 5; B_mat[0][3] =
            -2; B_mat[0][4] = a0;
        B_mat[1][0] = -2;  B_mat[1][1] = 1; B_mat[1][2] = -1; B_mat[1][3] =
             -2; B_mat[1][4] = a1;
        B_mat[2][0] = 2;  B_mat[2][1] = 0; B_mat[2][2] = 5; B_mat[2][3] = 2;
             B_mat[2][4] = a2;
        cout << "B:\n" << B_mat <<endl;
        //cout << "det(B):\n" << B_mat.determinant() <<endl;
        //cout << "inv(B):\n" << B_mat.inverse() <<endl;
        cout << endl;

        // Perform Gaussian Elimination / Forward elimination
        for (int k=0; k<R; k++)
        {
                int i_max = k;
                Symbolic v_max = B_mat[i_max][k];

                if (i_max != k)

                for (int m=0; m<C; k++)
                {
                        Symbolic temp = B_mat[k][m];
                        B_mat[k][m] = B_mat[i_max][m];
                        B_mat[i_max][m] = temp;
                }

                for (int i=k+1; i<R; i++)
                {
                        // factor f to set current row kth element to 0 and
                            subsequently remaining kth column to 0
                        Symbolic f = B_mat[i][k]/B_mat[k][k];

                        // subtract fth multiple of corresponding kth row
                            element
                        for (int j=k+1; j<C; j++)
                        {
                                B_mat[i][j] -= B_mat[k][j]*f;
                                //B_mat[i][j] = B_mat[i][j]/B_mat[i][i];
                                // filling lower triangular matrix with zeros
                        }
```

```
                        B_mat[i][k] = 0;
                }
        }

        cout << "B (in reduced row form) :\n" << B_mat <<endl;

        for (int i = 0; i < R; i++)
        {
                Symbolic f2 = B_mat[i][i];
                for (int j = 0; j < C; j++)
                {
                        B_mat[i][j] = B_mat[i][j]/f2;
                }

        }
        cout << "B (in row reduced echelon form) :\n" << B_mat <<endl;
        cout<<endl;
        return 0;
}
```

**C++ Code 243:** *main.cpp "Gaussian Elimination for Symbolic Matrix"*

To compile it, type:
**g++ -o result main.cpp -lsymbolicc++**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- After we get a reduced row form, the next step is to all entries of the matrix with the diagonal / $B_mat[i][i]$ for that current row. So we can have a leading one / the entry is 1 for that row and only 0 on its left and below.

```
        for (int i = 0; i < R; i++)
        {
                Symbolic f2 = B_mat[i][i];
                for (int j = 0; j < C; j++)
                {
                        B_mat[i][j] = B_mat[i][j]/f2;
                }

        }
```

  Placing **Symbolic f2 = B_mat[i][i]** before the second **for** loop is crucial as it will save the value of the diagonal entry first, and the whole row will make the change we wanted / the computation will be correct.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian
 for Symbolic Matrix with SymbolicC++  ]# ./main
B:
[ 1  3  5 -2 a0]
[-2  1 -1 -2 a1]
[ 2  0  5  2 a2]


B (in reduced row form) :
[        1               3               5              -2              a0       ]
[        0               7               9              -6            a1+2*a0    ]
[        0               0             19/7             6/7       a2-2/7*a0+6/7*a1]

B (in row reduced echelon form) :
[          1                      3                      5                      -2                      a0          ]
[          0                      1                     9/7                    -6/7              1/7*a1+2/7*a0     ]
[          0                      0                      1                     6/19          7/19*a2-2/19*a0+6/19*a1]
```

**Figure 23.236:** *The computation for Gaussian Elimination for symbolic matrix with C++ and SymbolicC++ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination for Symbolic Matrix with SymbolicC++/main.cpp**).*

## XVIII. Best Approximation; Least Squares

[DF*] Suppose that $A\boldsymbol{x} = \boldsymbol{b}$ is an inconsistent linear system of $m$ equations in $n$ unknowns, the inconsistency could be caused by measurement errors in the coefficients of $A$.
Since no exact solution is possible, we will look for a vector $\boldsymbol{x}$ that comes as "close as possible" to being a solution in the sense that it minimizes $||\boldsymbol{b} - A\boldsymbol{x}||$ with respect to the Euclidean inner product on $\mathbb{R}^m$. You can think of $A\boldsymbol{x}$ as an approximation to $\boldsymbol{b}$ and $||\boldsymbol{b} - A\boldsymbol{x}||$ as the error in that approximation, the smaller the error, the better the approximation.

[DF*] In short:

- $A\boldsymbol{x} = b$ is a linear system of $m$ equations and $n$ unknowns.
- Vector $\boldsymbol{x}$ is the least squares solution of the system that minimizes $||\boldsymbol{b} - A\boldsymbol{x}||$.
- $\boldsymbol{b} - A\boldsymbol{x}$ is the least squares error vector.
- $||\boldsymbol{b} - A\boldsymbol{x}||$ is the least squares error.

[DF*] Suppose the matrix form of $\boldsymbol{b} - A\boldsymbol{x}$ is

$$\boldsymbol{b} - A\boldsymbol{x} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

The term "least squares solution" results from the fact that minimizing $||\boldsymbol{b} - A\boldsymbol{x}||$ also minimizes

$$||\boldsymbol{b} - A\boldsymbol{x}||^2 = e_1^2 + e_2^2 + \cdots + e_m^2$$

**Theorem 23.152: Best Approximation Theorem**

If $W$ is a finite-dimensional subspace of an inner product space $V$, and if $\boldsymbol{b}$ is a vector in $V$, then $\text{proj}_W\boldsymbol{b}$ is the best approximation to $\boldsymbol{b}$ from $W$ in the sense that

$$\boldsymbol{b} - \text{proj}_W\boldsymbol{b}|| < ||\boldsymbol{b} - \boldsymbol{W}||$$

for every vector $\boldsymbol{w}$ in $W$ that is different from $\text{proj}_W\boldsymbol{b}$.

[DF*] For every vector $\boldsymbol{w}$ in $W$, we can write

$$\boldsymbol{b} - \boldsymbol{w} = (\boldsymbol{b} - \text{proj}_W \boldsymbol{b}) + (\text{proj}_W \boldsymbol{b} - \boldsymbol{w}) \tag{23.281}$$

Since $\boldsymbol{b} - \text{proj}_W \boldsymbol{b}$ is orthogonal to $W$, the two terms on the right side of the equation are orthogonal. Thus, it follows from the Theorem of Pythagoras that

$$||\boldsymbol{b} - \boldsymbol{w}||^w = ||\boldsymbol{b} - \text{proj}_W \boldsymbol{b}||^2 + ||\text{proj}_W \boldsymbol{b} - \boldsymbol{w}||^2$$

Since $\boldsymbol{w} \neq \text{proj}_W \boldsymbol{b}$, hence

$$||\boldsymbol{b} - \text{proj}_W \boldsymbol{b}||^2 < ||\boldsymbol{b} - \boldsymbol{w}||^2$$

Since norms are nonnegative, it follows that

$$||\boldsymbol{b} - \text{proj}_W \boldsymbol{b}|| < ||\boldsymbol{b} - \boldsymbol{w}||$$

**Theorem 23.153: Least Squares Solutions of Linear Systems**

For every linear system $A\boldsymbol{x} = \boldsymbol{b}$, the associated normal system

$$A^T A\boldsymbol{x} = A^T \boldsymbol{b} \tag{23.282}$$

is consistent, and all solutions are least squares solutions of $A\boldsymbol{x} = \boldsymbol{b}$. Moreover, if $W$ is the column space of $A$, and $\boldsymbol{x}$ is any least squares solution of $A\boldsymbol{x} = \boldsymbol{b}$, then the orthogonal projection of $\boldsymbol{b}$ on $W$ is

$$\text{proj}_W \boldsymbol{b} = A\boldsymbol{x} \tag{23.283}$$

[DF*] In general, least squares solutions of linear systems are not unique. If the linear system turned out to have a unique least squares solution, then the coefficient matrix of the system happened to satisfy certain conditions that guarantee uniqueness.

**Theorem 23.154: Linearly Independent and Invertible**

If $A$ is an $m \times n$ matrix, then the following are equivalent.

(a) $A$ has linearly independent column vectors.

(b) $A^T A$ is invertible.

**Theorem 23.155: Formula for the Least Squares Solution**

If $A$ is an $m \times n$ matrix with linearly independent column vectors, then for every $m \times 1$ matrix $\boldsymbol{b}$, the linear system $A\boldsymbol{x} = \boldsymbol{b}$ has a unique least squares solution. This solution is given by

$$\boldsymbol{x} = (A^T A)^{-1} A^T \boldsymbol{b} \tag{23.284}$$

Moreover, if $W$ is the column space of $A$, then the orthogonal projection of $\boldsymbol{b}$ on $W$ is

$$\text{proj}_W \boldsymbol{b} = A\boldsymbol{x} = A(A^T A)^{-1} A^T \boldsymbol{b} \tag{23.285}$$

[DF*] In practice, least squares solutions of $A\boldsymbol{x} = \boldsymbol{b}$ are typically found by using some variation of Gaussian elimination to solve the normal equations or by using $QR$-decomposition.

**Theorem 23.156: $QR$-Decomposition in Least Squares Problems**

If $A$ is an $m \times n$ matrix with linearly independent column vectors, and if $A = QR$ is a $QR$-decomposition of $A$, then for each $\boldsymbol{b}$ in $\mathbb{R}^m$ the system $A\boldsymbol{x} = \boldsymbol{b}$ has a unique least squares solution given by

$$\boldsymbol{x} = R^{-1}Q^T\boldsymbol{b} \tag{23.286}$$

**Definition 23.69: Orthogonal Projections on Subspaces of $\mathbb{R}^m$**

If $W$ is a subspace of $\mathbb{R}^m$, then the linear transformation $P : \mathbb{R}^m \to W$ that maps each vector $\boldsymbol{x}$ in $\mathbb{R}^m$ into its orthogonal projection $\text{proj}_W\boldsymbol{x}$ in $W$ is called the orthogonal projection of $\mathbb{R}^m$ on $W$.

[DF*] The standard matrix for the transformation $P$ is

$$[P] = A(A^TA)^{-1}A^T \tag{23.287}$$

where $A$ is constructed using any basis for $W$ as its column vectors.

[DF*] The null space and the row space of an $m \times n$ matrix $A$ are orthogonal complements, as are the null space of $A^T$ and the column space of $A$.

Thus, given a linear system $A\boldsymbol{x} = \boldsymbol{b}$ in which $A$ is an $m \times n$ matrix, the Projection Theorem tells us that the vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ can each be decomposed into sums of orthogonal terms as

$$\begin{aligned}\boldsymbol{x} &= \boldsymbol{x}_{\text{row}(A)} + \boldsymbol{x}_{\text{null}(A)}\\ \boldsymbol{b} &= \boldsymbol{b}_{\text{null}(A^T)} + \boldsymbol{b}_{\text{col}(A)}\end{aligned}$$

where $\boldsymbol{x}_{\text{row}(A)}$ and $\boldsymbol{x}_{\text{null}(A)}$ are the orthogonal projections of $\boldsymbol{x}$ on the row space of $A$ and the null space of $A$, and the vectors $\boldsymbol{b}_{\text{null}(A^T)}$ and $\boldsymbol{b}_{\text{col}(A)}$ are the orthogonal projections of $\boldsymbol{b}$ on the null space of $A^T$ and the column space of $A$.

To comprehend this it is better if we use example, let $A$ be a matrix $3 \times 2$ ($m \times n$)

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

then for $A\boldsymbol{x} = \boldsymbol{b}$ we will have

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

In short we know that

$$\begin{aligned}\text{row}(A) &\in \mathbb{R}^n\\ \text{null}(A) &\in \mathbb{R}^n\\ \text{col}(A) &\in \mathbb{R}^m\\ \text{null}(A^T) &\in \mathbb{R}^m\end{aligned}$$

Sometimes if people forget, just remember that null$(A)$ is the solution for $A\boldsymbol{x} = \mathbf{0}$, since $\boldsymbol{x}$ is a vector in $\mathbb{R}^n$, this explains why null$(A) \in \mathbb{R}^n$. To always re-read and remember the theorem from previous sections will help to boost your linear algebra' knowledge.

The illustration below will show the least squares solutions of $A\boldsymbol{x} = \boldsymbol{b}$ are the exact solutions of the equation $A\boldsymbol{x} = \boldsymbol{b}_{\text{col}(A)}$. This, of course, is only pictorial since the fundamental spaces

**Figure 23.237:** *The fundamental spaces of $A$ by perpendicular lines in $\mathbb{R}^n$ and $\mathbb{R}^m$ on which we indicated the orthogonal projections of $\boldsymbol{x}$ and $\boldsymbol{b}$. It shows $A\boldsymbol{x}$ as a point in the column space of $A$ and conveys that $\boldsymbol{b}_{col(A)}$ is the point in col$(A)$ that is closest to $\boldsymbol{b}$.*

need not be one-dimensional.

[DF*] In the chapter of General Vector Spaces we learned that

$$P_\theta \begin{bmatrix} \cos^2\theta & \sin\,\theta\,\cos\,\theta \\ \sin\,\theta\,\cos\,\theta & \sin^2\theta \end{bmatrix}$$

is the standard matrix for the orthogonal projection on the line $W$ through the origin of $\mathbb{R}^2$ that makes an angle $\theta$ with the positive $x$-axis.

With least squares we can obtain $P_\theta$ easily. The column vectors of $A$ can be formed from any basis for $W$. Since $W$ is one-dimensional, we can take $\boldsymbol{w} = (\cos\,\theta, \sin\,\theta)$ as the basis vector, so

$$A = \begin{bmatrix} \cos\,\theta \\ \sin\,\theta \end{bmatrix}$$

$A^TA$ is the $1 \times 1$ identity matrix. Thus,

$$[P] = A(A^TA)^{-1}A^T = AA^T$$

$$AA^T = \begin{bmatrix} \cos\,\theta \\ \sin\,\theta \end{bmatrix} \begin{bmatrix} \cos\,\theta & \sin\,\theta \end{bmatrix} = \begin{bmatrix} \cos^2\theta & \sin\,\theta\,\cos\,\theta \\ \sin\,\theta\,\cos\,\theta & \sin^2\theta \end{bmatrix} = P_\theta$$

[DF*] This theorem of equivalent statements relates all of the major topics that have been studied until this section, under this section we add another one.

**Theorem 23.157: Equivalent Statements for Square Matrix**

If $A$ is an $n \times n$ matrix, then the following statements are equivalent

(a) $A$ is invertible.

(b) $A\boldsymbol{x} = \mathbf{0}$ has only the trivial solution.

(c) The reduced row echelon form of $A$ is $I_n$.

(d) $A$ is expressible as a product of elementary matrices.

(e) $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for every $n \times 1$ matrix $\boldsymbol{b}$.

(f) $A\boldsymbol{x} = \boldsymbol{b}$ has exactly one solution for every $n \times 1$ matrix $\boldsymbol{b}$.

(g) $\det(A) \neq 0$.

(h) The column vectors of $A$ are linearly independent.

(i) The row vectors of $A$ are linearly independent.

(j) The column vectors of $A$ span $\mathbb{R}^n$.

(k) The row vectors of $A$ span $\mathbb{R}^n$.

(l) The column vectors of $A$ form a basis for $\mathbb{R}^n$.

(m) The row vectors of $A$ form a basis for $\mathbb{R}^n$.

(n) $A$ has rank $n$.

(o) $A$ has nullity 0.

(p) The orthogonal complement of the null space of $A$ is $\mathbb{R}^n$.

(q) The orthogonal complement of the row space of $A$ is $\{\mathbf{0}\}$.

(r) The range of $T_A$ is $\mathbb{R}^n$.

(s) $T_A$ is one-to-one.

(t) $\lambda = 0$ is not an eigenvalue of $A$.

(u) $A^T A$ is invertible

[DF*] **Example:**
Show that if $\boldsymbol{w} = (a, b, c)$ is a nonzero vector, then the standard matrix for the orthogonal projection of $\mathbb{R}^3$ on the line span$\{\boldsymbol{w}\}$ is

$$P = \frac{1}{a^2 + b^2 + c^2} \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix}$$

**Solution:**

We can take $\boldsymbol{w} = (a, b, c)$ as the basis vector, so

$$A = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$A^T A$ is

$$A^T A = \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = a^2 + b^2 + c^2$$

Thus,

$$[P] = A(A^T A)^{-1} A^T = AA^T$$

$$A(a^2+b^2+c^2)^{-1}A^T = \frac{1}{a^2+b^2+c^2} AA^T = \frac{1}{a^2+b^2+c^2} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} a & b & c \end{bmatrix} = \frac{1}{a^2+b^2+c^2} \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix}$$

[DF*] **Example:**
In $\mathbb{R}^3$, consider the line $l$ given by the equations

$$x = -1, \quad y = t, \quad z = -t$$

and the line $m$ given by the equations

$$x = 2s, \quad y = 1 + s, \quad z = s$$

Let $P$ be a point on $l$, and let $Q$ be a point on $m$. Find the values of $t$ and $s$ that minimize the distance between the lines by minimizing the squared distance $||P - Q||^2$.
**Solution:**
The minimal distance can be obtained if there exists a point that lies on both line $l$ and line $m$, so it will fulfill these conditions

$$\begin{aligned} (x_l, y_l, z_l) &= (x_m, y_m, z_m) \\ (-1, t, -t) &= (2s, 1+s, s) \\ -1 &= 2s \\ t &= 1 + s \\ -t &= s \end{aligned}$$

we obtain

$$\begin{aligned} s &= -\frac{1}{2} \\ t &= \frac{1}{2} \end{aligned}$$

the point $\left(-1, \frac{1}{2}, -\frac{1}{2}\right)$ lies on line $m$ and $l$, or we can saya that $P = Q$. Thus since $||P - Q|| = 0$, these are the values of $s$ and $t$ that minimize the distance between the lines.

## i. C++ Computation: Least Squares Solution of a Linear System

Find all least squares solutions of the linear system

$$\begin{aligned} x_1 - x_2 &= 4 \\ 3x_1 + 2x_2 &= 1 \\ -2x_1 + 4x_2 &= 3 \end{aligned}$$

Find the error vector and the error.

**Solution:**

It will be convenient to express the system in the matrix form $A\boldsymbol{x} = \boldsymbol{b}$, where

$$A = \begin{bmatrix} 1 & -1 \\ 3 & 2 \\ -2 & 4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix}$$

It follows that

$$A^T A = \begin{bmatrix} 1 & 3 & -2 \\ -1 & 2 & 4 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 3 & 2 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 14 & -3 \\ -3 & 21 \end{bmatrix}$$

$$A^T \boldsymbol{b} = \begin{bmatrix} 1 & 3 & -2 \\ -1 & 2 & 4 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \end{bmatrix}$$

so the normal system $A^T A\boldsymbol{x} = A^T \boldsymbol{b}$ is

$$\begin{bmatrix} 14 & -3 \\ -3 & 21 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \end{bmatrix}$$

Solving this yields a unique least squares solution, namely

$$x_1 = \frac{17}{95}, \quad x_2 = \frac{143}{285}$$

The error vector is

$$\boldsymbol{b} - A\boldsymbol{x} = \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 1 & -1 \\ 3 & 2 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} \frac{17}{95} \\ \frac{143}{285} \end{bmatrix} = \begin{bmatrix} \frac{1232}{285} \\ -\frac{154}{285} \\ \frac{4}{3} \end{bmatrix}$$

remember that the error vector is actually the component of $\boldsymbol{b}$ that is orthogonal to the column space of $A$, and the error is

$$||\boldsymbol{b} - A\boldsymbol{x}|| \approx 4.556$$

The C++ code is using **Armadillo**, and the plot is using **gnuplot**. We take the input of matrix $A$ and vector $\boldsymbol{b}$ from textfiles. The computation from **Armadillo** differs a bit from the book [11] solution of Example 1 chapter 6.4 about Best Approximation, probably because of roundoff error.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include <boost/tuple/tuple.hpp>
#include <cmath>
#include <utility>

#include "gnuplot-iostream.h"

using namespace std;
using namespace arma;

const int N = 5;
const int R = 3;
const int C = 2;

// Driver code
int main(int argc, char** argv)
{
       arma::mat A(R,C,fill::zeros); //declare matrix A with size of R X C
           all have 0 entries with Armadillo.
       A.load("matrixA.txt"); // load matrixA.txt with Armadillo

       arma::mat b(R,1,fill::zeros);
       b.load("vectorb.txt");

       cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;
       cout <<"Vector b:" << "\n" << b <<endl;

       arma::mat A_transpose(C,R,fill::zeros);
       A_transpose = A.t();

       arma::mat AtA(C,C,fill::zeros);
       arma::mat Atb(C,1,fill::zeros);
       AtA = A_transpose*A;
       Atb = A_transpose*b;

       cout <<"A^T:" << "\n" << A_transpose <<endl;
       cout <<"A^T * A:" << "\n" << AtA <<endl;
       cout <<"(A^T) * b:" << "\n" << Atb <<endl;

       // Find the solution for x from A^T * A * x = A^T * b
       mat X;
       mat bprojW;
```

```
X = solve(AtA,Atb,solve_opts::force_approx);
bprojW = A*X;
cout <<"The least squares solution " << endl;
cout <<"x :" << "\n" << X <<endl;
cout <<"Ax :" << "\n" << bprojW <<endl;

mat error_vector;
error_vector = b-bprojW;
cout <<"The error vector " << endl;
cout <<"b - Ax :" << "\n" << error_vector <<endl;

cout <<"The error" << endl;
cout <<"|| b - Ax || :" << norm(error_vector) <<endl;

cout <<"\n< Ax, b - Ax > :" << dot(bprojW,error_vector) <<endl;

Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_z;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_B_dz;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_z;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_C_dz;

float o = 0;

// Create a vector b in R^3
pts_A_x .push_back(o);
pts_A_y .push_back(o);
pts_A_z .push_back(o);
pts_A_dx.push_back(b[0]);
pts_A_dy.push_back(b[1]);
pts_A_dz.push_back(b[2]);
// Create a vector projection of b on W in R^3
```

```
            pts_B_x .push_back(o);
            pts_B_y .push_back(o);
            pts_B_z .push_back(o);
            pts_B_dx.push_back(bprojW[0]);
            pts_B_dy.push_back(bprojW[1]);
            pts_B_dz.push_back(bprojW[2]);
            // Create a vector component of b orthogonal to W in R^3
            pts_C_x .push_back(o);
            pts_C_y .push_back(o);
            pts_C_z .push_back(o);
            pts_C_dx.push_back(error_vector[0]);
            pts_C_dy.push_back(error_vector[1]);
            pts_C_dz.push_back(error_vector[2]);

            // Don't forget to put "\n" at the end of each line!
            gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
                persist\n"; // To make the plot interactive
            gp << "set xrange [-5:5]\nset yrange [-5:5]\nset zrange [-5:5]\n";
            // '-' means read from stdin. The send1d() function sends data to
                gnuplot's stdin.
            gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
                axis'\n";
            // the splot '++' to create the plane W spanned by the vector
                columns of A / column space of A
            gp << "splot '++' using ($1-$2):($1*3 + $2*2):($2*4 + $1*-2) with
                lines lw 2 lc rgb 'gray' title 'W = c1(1,3,-2) + c2(-1,2,4)',
                \
            '-' with vectors lw 3 lc rgb 'cyan' title 'b', '-' with vectors lw
                3 lc rgb 'skyblue' title 'proj_{W} b ', \
            '-' with vectors lw 3 lc rgb 'navy' title 'b - proj_{W} b '\n";
            gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
                pts_A_dy, pts_A_dz));
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
                pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
                pts_C_dy, pts_C_dz));

            return 0;
        }
```

**C++ Code 244:** *main.cpp "Least squares solution of an inconsistent linear system"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- The column space of $A$ (the plane where the vector $\text{proj}_W \boldsymbol{b}$ resides), or we can also say the column space $W$ of the matrix $A$, is denoted by

$$W = c_1(1, 3, -2) + c_2(-1, 2, 4)$$

with $c_1, c_2$ scalars, the command to plot such plane in **gnuplot** is:
**splot '++' using (**$1-$**2):(**$1*3+$**2*2):(**$2*4+$**1*-2) with lines**

```
...
...

gp << "splot '++' using ($1-$2):($1*3 + $2*2):($2*4 + $1*-2)
    with lines lw 2 lc rgb 'gray' title 'W = c1(1,3,-2) + c2
    (-1,2,4)', \
'-' with vectors lw 3 lc rgb 'cyan' title 'b', '-' with
    vectors lw 3 lc rgb 'skyblue' title 'proj_{W} b ', \
'-' with vectors lw 3 lc rgb 'navy' title 'b - proj_{W} b '\n"
    ;

...
...
```

```
Matrix A (Coefficient Matrix):
   1.0000  -1.0000
   3.0000   2.0000
  -2.0000   4.0000

Vector b:
   4.0000
   1.0000
   3.0000

A^T:
   1.0000   3.0000  -2.0000
  -1.0000   2.0000   4.0000

A^T * A:
   14.0000   -3.0000
   -3.0000   21.0000

(A^T) * b:
    1.0000
   10.0000

The least squares solution
x :
   0.1789
   0.5018

Ax :
  -0.3228
   1.5404
   1.6491

The error vector
b - Ax :
   4.3228
  -0.5404
   1.3509

The error
|| b - Ax || :4.56109

< Ax, b - Ax > :8.88178e-16
```

**Figure 23.238:** *The computation to obtain the least squares solution of inconsistent linear system $A\boldsymbol{x} = \boldsymbol{b}$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Least Square Solution for Inconsistent System with Armadillo/main.cpp**).*

**Figure 23.239:** *The plot of vector* $\boldsymbol{b}$ *the orthogonal projection of* $\boldsymbol{b}$ *on W,* $proj_W\boldsymbol{b}$ *on the column space W of the matrix A, and the vector component of* $\boldsymbol{b}$ *orthogonal to W, which is* $\boldsymbol{b} - proj_W\boldsymbol{b}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Least Square Solution for Inconsistent System with Armadillo/main.cpp**).*

## ii. C++ Computation: Orthogonal Projection on a Subspace

Find the orthogonal projection of the vector $\boldsymbol{u} = (-3, -3, 8, 9)$ on the subspace $\mathbb{R}^4$ spanned by the vectors

$$\boldsymbol{u}_1 = (3, 1, 0, 1), \quad \boldsymbol{u}_2 = (1, 2, 1, 1), \quad \boldsymbol{u}_3 = (-1, 0, 2, -1)$$

**Solution:**

The subspace $W$ of $\mathbb{R}^4$ spanned by $\boldsymbol{u}_1, \boldsymbol{u}_2$, and $\boldsymbol{u}_3$ is the column space of the matrix

$$A = \begin{bmatrix} 3 & 1 & -1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \\ 1 & 1 & -1 \end{bmatrix}$$

Thus, if $\boldsymbol{u}$ is expressed as a column vector, we can find the orthogonal projection of $\boldsymbol{u}$ on $W$ by finding a least squares solution of the system $A\boldsymbol{x} = \boldsymbol{u}$ and then calculating

$$\text{proj}_W \boldsymbol{u} = A\boldsymbol{x}$$

from the least squares solution. The computations are as follows: The system $A\boldsymbol{x} = \boldsymbol{u}$ is

$$\begin{bmatrix} 3 & 1 & -1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \\ 8 \\ 9 \end{bmatrix}$$

so

$$A^T A = \begin{bmatrix} 3 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ -1 & 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} 3 & 1 & -1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \\ 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 11 & 6 & -4 \\ 6 & 7 & 0 \\ -4 & 0 & 6 \end{bmatrix}$$

$$A^T \boldsymbol{u} = \begin{bmatrix} 3 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ -1 & 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} -3 \\ -3 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} -3 \\ 8 \\ 10 \end{bmatrix}$$

The normal system $A^T A\boldsymbol{x} = A^T \boldsymbol{u}$ in this case is

$$A^T \boldsymbol{u} = \begin{bmatrix} 11 & 6 & -4 \\ 6 & 7 & 0 \\ -4 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -3 \\ 8 \\ 10 \end{bmatrix}$$

Solving the system yields

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}$$

as the least squares solution of $A\boldsymbol{x} = \boldsymbol{u}$, so

$$\text{proj}_W \boldsymbol{u} = A\boldsymbol{x} = \begin{bmatrix} 3 & 1 & -1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ 4 \\ 0 \end{bmatrix}$$

or, in comma-delimited notation, $\text{proj}_W \boldsymbol{u} = (-2, 3, 4, 0)$.

The C++ code only uses **Armadillo**, since the matrix computation here is not that difficult to be handled by this library. We use 4 textfiles as vectors input: **vectorU.txt**, **vectorU1.txt**, **vectorU2.txt**, **vectorU3.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

#define R 4
#define C 3

// Driver code
int main(int argc, char** argv)
{
       arma::mat U(R,1,fill::zeros);
       arma::mat U1(R,1,fill::zeros);
       arma::mat U2(R,1,fill::zeros);
       arma::mat U3(R,1,fill::zeros);
       arma::mat A(R,C,fill::zeros);
       U.load("vectorU.txt");
       U1.load("vectorU1.txt");
       U2.load("vectorU2.txt");
       U3.load("vectorU3.txt");

       A.col(0) = U1;
       A.col(1) = U2;
       A.col(2) = U3;

       arma::mat A_transpose(C,R,fill::zeros);
       A_transpose = A.t();

       arma::mat X(C,1,fill::zeros);
       X = solve(A_transpose*A,A_transpose*U,solve_opts::force_approx);

       cout <<"Matrix A :" << "\n" << A <<endl;
       cout <<"Vector u :" << "\n" << U <<endl;
       cout <<"Vector u1 :" << "\n" << U1 <<endl;
       cout <<"Vector u2 :" << "\n" << U2 <<endl;
       cout <<"Vector u3 :" << "\n" << U3 <<endl;

       cout <<"Matrix A^{T}:" << "\n" << A_transpose <<endl;
```

```
        cout <<"Matrix A^{T} * A:" << "\n" << A_transpose*A <<endl;
        cout <<"Matrix A^{T} * u:" << "\n" << A_transpose*U <<endl;

        cout <<"\n Solution of the normal system A^{T} *A * x = A^{T} * u:"
            <<endl;
        cout <<"x: " << "\n" << X <<endl;

        cout <<"************************************" <<endl;
        cout <<"************************************" <<endl;
        cout <<"The least squares solution of Ax = u: " <<endl;
        cout <<"proj_{W} u = Ax : " << "\n" << A*X <<endl;

    }
```

**C++ Code 245:** *main.cpp "Orthogonal projection on a subspace"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Vector u1 :
   3.0000
   1.0000
        0
   1.0000

Vector u2 :
   1.0000
   2.0000
   1.0000
   1.0000

Vector u3 :
  -1.0000
        0
   2.0000
  -1.0000

Matrix A^{T}:
   3.0000   1.0000        0   1.0000
   1.0000   2.0000   1.0000   1.0000
  -1.0000        0   2.0000  -1.0000

Matrix A^{T} * A:
   11.0000    6.0000   -4.0000
    6.0000    7.0000         0
   -4.0000         0    6.0000

Matrix A^{T} * u:
   -3.0000
    8.0000
   10.0000


 Solution of the normal system A^{T} *A * x = A^{T} * u:
x:
  -1.0000
   2.0000
   1.0000

*************************************
*************************************
The least squares solution of Ax = u:
proj_{W} u = Ax :
  -2.0000e+00
   3.0000e+00
   4.0000e+00
  -3.3307e-16
```

**Figure 23.240:** *The computation to obtain the least squares solution of* $proj_W \boldsymbol{u} = A\boldsymbol{x}$ *(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Orthogonal Projection on a Subspace with Armadillo/main.cpp**).*

## iii. C++ Plot and Computation: Basis and Standard Matrix for the Orthogonal Projection on a Plane

Let $W$ be the plane with equation $3x - 4y + z = 0$.

1. Find a basis for $W$.
2. Find the standard matrix for the orthogonal projection on $W$.
3. Use the matrix obtained in part (b) to find the orthogonal projection of a point $P_0(x_0, y_0, z_0)$ on $W$.
4. Find the distance between the point $P_0(2, 1, -1)$ and the plane $W$.

**Solution:**

1. If $x = s$ and $y = t$, then
$$z = 4y - 3x = 4t - 3s$$
a point on the plane $W$ can be represented by
$$(s, t, 4t - 3s) = s(1, 0, -3) + t(0, 1, 4)$$
thus
$$\boldsymbol{w}_1 = \begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix}, \quad \boldsymbol{w}_2 = \begin{bmatrix} 0 \\ 1 \\ 4 \end{bmatrix}$$
are basis for $W$.

2. We can form matrix $A$ from $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -3 & 4 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 4 \end{bmatrix}$$
$$A^T A = \begin{bmatrix} 10 & -12 \\ -12 & 17 \end{bmatrix}$$
$$\det(A^T A) = 26$$
$$(A^T A)^{-1} = \frac{1}{26} \begin{bmatrix} 17 & 12 \\ 12 & 10 \end{bmatrix}$$
Then
$$[P] = A(A^T A)^{-1} A^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -3 & 4 \end{bmatrix} \frac{1}{26} \begin{bmatrix} 17 & 12 \\ 12 & 10 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 4 \end{bmatrix} = \frac{1}{26} \begin{bmatrix} 17 & 12 & -3 \\ 12 & 10 & 4 \\ -3 & 4 & 25 \end{bmatrix}$$
is the standard matrix for the orthogonal projection on $W$.

3. The orthogonal projection of a point $P_0(x_0, y_0, z_0)$ on $W$ is

$$[P]\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \frac{1}{26}\begin{bmatrix} 17 & 12 & -3 \\ 12 & 10 & 4 \\ -3 & 4 & 25 \end{bmatrix}\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} \frac{17x_0+12y_0-3z_0}{26} \\ \frac{12x_0+10y_0+4z_0}{26} \\ \frac{-3x_0+4y_0+25z_0}{26} \end{bmatrix}$$

The orthogonal projection in comma-delimited form is

$$\left(\frac{17x_0+12y_0-3z_0}{26}, \frac{12x_0+10y_0+4z_0}{26}, \frac{-3x_0+4y_0+25z_0}{26}\right)$$

4. The orthogonal projection of a point $P_0(2, 1, -1)$ on $W$ is

$$\left(\frac{17(2)+12(1)-3(-1)}{26}, \frac{12(2)+10(1)+4(-1)}{26}, \frac{-3(2)+4(1)+25(-1)}{26}\right) = \left(\frac{49}{26}, \frac{30}{26}, \frac{-27}{26}\right)$$

The distance between the point $P_0(2, 1, -1)$ and the plane $W$ is

$$d = \sqrt{\left(2-\left(\frac{49}{26}\right)\right)^2 + \left(1-\left(\frac{30}{26}\right)\right)^2 + \left(-1-\left(\frac{-27}{26}\right)\right)^2} = 0.196116$$

We will compare the result for $d$ with the formula from theorem we learned from Euclidean Vector Spaces section in Orthogonality for $\mathbb{R}^3$, the distance $D$ between the point $P_0(2, 1, -1)$ and the plane $ax + by + cz + d = 0$ has formula of

$$D = \frac{|ax_0 + by_0 + cz_0|}{\sqrt{a^2+b^2+c^2}}$$

we now substitute the value

$$D = \frac{|3(2) + -4(1) + 1(-1)|}{\sqrt{3^2+(-4)^2+1^2}} = 0.196116$$

the results of both formula of $d$ and $D$ are the same.

The C++ code is using **Armadillo** for the computation and **gnuplot** for the plotting part. We compute by hand the basis for $W$, then we use the basis as the column vector for matrix $A$ that will be loaded from textfile **matrixA.txt**, and the point $P_0(2, 1, -1)$ will be loaded from textfile **vectorp0.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>
#include <boost/tuple/tuple.hpp>
#include <cmath>
#include <utility>

#include "gnuplot-iostream.h"
```

```
using namespace std;
using namespace arma;

const int N = 5;
const int R = 3;
const int C = 2;

// Driver code
int main(int argc, char** argv)
{
       // Plane W has equation of 3x — 4y + z = 0
       // We put the basis for W as column vector for matrix A
       arma::mat A(R,C,fill::zeros); //declare matrix A with size of R X C
           all have 0 entries with Armadillo.
       A.load("matrixA.txt"); // load matrixA.txt with Armadillo

       arma::mat p0(R,1,fill::zeros);
       p0.load("vectorp0.txt");

       cout <<"Matrix A (Coefficient Matrix):" << "\n" << A <<endl;

       arma::mat A_transpose(C,R,fill::zeros);
       A_transpose = A.t();

       arma::mat AtA(C,C,fill::zeros);
       arma::mat AtA_inv(C,C,fill::zeros);
       AtA = A_transpose*A;
       AtA_inv = inv(AtA);

       cout <<"A^T:" << "\n" << A_transpose <<endl;
       cout <<"A^T * A:" << "\n" << AtA <<endl;
       cout <<"det(A^T * A):" << det(AtA) <<endl;
       cout <<"(A^T * A)^{—1}:" << "\n" << AtA_inv <<endl;

       arma::mat P(C,C,fill::zeros);
       P = A * inv(AtA) * A_transpose;
       cout <<"[P]:" << "\n" << P <<endl;

       // Find the orthogonal projection of a point P0(2,1,—1) on plane W
       arma::mat Pp0(R,1,fill::zeros);
       Pp0 = P*p0;
       cout <<"The orthogonal projection of a point P0 (2,1,—1) on plane W
            with equation 3x—4y+z=0 is " << endl;
       cout <<"[P] * P0 :" << "\n" << Pp0 <<endl;

       cout <<"The distance between the point P0 (2,1,—1) and the plane W
           with equation 3x—4y+z=0 is " << endl;
       // The general plane equation is ax+by+cz+d=0
```

```
float a = 3;
float b = -4;
float c = 1;
float d = 0;
float x0 = 2;
float y0 = 1;
float z0 = -1;

float distance = sqrt( pow(x0-Pp0[0],2) + pow(y0-Pp0[1],2) + pow(z0
    -Pp0[2],2));
float D = abs(a*x0 + b*y0 + c*z0) / ( sqrt(pow(a,2) + pow(b,2) + pow
    (c,2)) );

cout <<"d :" << distance <<endl;
cout <<"D :" << D <<endl;

Gnuplot gp;

// We use a separate container for each column, like so:
std::vector<double> pts_A_x;
std::vector<double> pts_A_y;
std::vector<double> pts_A_z;
std::vector<double> pts_A_dx;
std::vector<double> pts_A_dy;
std::vector<double> pts_A_dz;
std::vector<double> pts_B_x;
std::vector<double> pts_B_y;
std::vector<double> pts_B_z;
std::vector<double> pts_B_dx;
std::vector<double> pts_B_dy;
std::vector<double> pts_B_dz;
std::vector<double> pts_C_x;
std::vector<double> pts_C_y;
std::vector<double> pts_C_z;
std::vector<double> pts_C_dx;
std::vector<double> pts_C_dy;
std::vector<double> pts_C_dz;
std::vector<double> pts_D_x;
std::vector<double> pts_D_y;
std::vector<double> pts_D_z;
std::vector<double> pts_D_dx;
std::vector<double> pts_D_dy;
std::vector<double> pts_D_dz;
std::vector<double> pts_E_x;
std::vector<double> pts_E_y;
std::vector<double> pts_E_z;
std::vector<double> pts_E_dx;
std::vector<double> pts_E_dy;
```

```
std::vector<double> pts_E_dz;

float o = 0;

// Create a vector of a point P0 (2,1,-1)
pts_A_x .push_back(o);
pts_A_y .push_back(o);
pts_A_z .push_back(o);
pts_A_dx.push_back(p0[0]);
pts_A_dy.push_back(p0[1]);
pts_A_dz.push_back(p0[2]);
// Create a vector projection of P0 on W in R^3
pts_B_x .push_back(o);
pts_B_y .push_back(o);
pts_B_z .push_back(o);
pts_B_dx.push_back(Pp0[0]);
pts_B_dy.push_back(Pp0[1]);
pts_B_dz.push_back(Pp0[2]);
// Create a distance vector between P0 and the vector projection of
    P0 on W in R^3
pts_C_x .push_back(p0[0]);
pts_C_y .push_back(p0[1]);
pts_C_z .push_back(p0[2]);
pts_C_dx.push_back(Pp0[0]-p0[0]);
pts_C_dy.push_back(Pp0[1]-p0[1]);
pts_C_dz.push_back(Pp0[2]-p0[2]);
// Create the first basis vector for W
pts_D_x .push_back(o);
pts_D_y .push_back(o);
pts_D_z .push_back(o);
pts_D_dx.push_back(1);
pts_D_dy.push_back(0);
pts_D_dz.push_back(-3);
// Create the second basis vector for W
pts_E_x .push_back(o);
pts_E_y .push_back(o);
pts_E_z .push_back(o);
pts_E_dx.push_back(0);
pts_E_dy.push_back(1);
pts_E_dz.push_back(4);

// Don't forget to put "\n" at the end of each line!
gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
    persist\n"; // To make the plot interactive
gp << "set xrange [-1:3]\nset yrange [-1:3]\nset zrange [-5:5]\n";
// '-' means read from stdin. The send1d() function sends data to
    gnuplot's stdin.
gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n set zlabel 'z-
```

```
                axis'\n";
            // the splot '++' to create the plane W spanned by the vector
                columns of A / column space of A
            gp << "splot '++' using 1:2:($1*-3 + $2*4) with lines lw 2 lc rgb '
                gray' title 'W = c1(1,0,-3) + c2(0,1,4)', \
            '-' with vectors lw 3 lc rgb 'cyan' title 'P_{0} (2,1,-1)', '-'
                with vectors lw 3 lc rgb 'skyblue' title 'proj_{W} P_{0} ', \
            '-' with vectors lw 3 lc rgb 'navy' title 'd', '-' with vectors lw
                3 lc rgb 'salmon' title '(1,0,-3)', '-' with vectors lw 3 lc
                rgb 'dark-red' title '(0,1,4)'\n";
            gp.send1d(boost::make_tuple(pts_A_x, pts_A_y, pts_A_z, pts_A_dx,
                pts_A_dy, pts_A_dz));
            gp.send1d(boost::make_tuple(pts_B_x, pts_B_y, pts_B_z, pts_B_dx,
                pts_B_dy, pts_B_dz));
            gp.send1d(boost::make_tuple(pts_C_x, pts_C_y, pts_C_z, pts_C_dx,
                pts_C_dy, pts_C_dz));
            gp.send1d(boost::make_tuple(pts_D_x, pts_D_y, pts_D_z, pts_D_dx,
                pts_D_dy, pts_D_dz));
            gp.send1d(boost::make_tuple(pts_E_x, pts_E_y, pts_E_z, pts_E_dx,
                pts_E_dy, pts_E_dz));

            return 0;
        }
```

**C++ Code 246:** *main.cpp "Basis and standard matrix for an orthogonal projection on a plane"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
Matrix A (Coefficient Matrix):
   1.0000        0
        0   1.0000
  -3.0000   4.0000

A^T:
   1.0000        0  -3.0000
        0   1.0000   4.0000

A^T * A:
   10.0000  -12.0000
  -12.0000   17.0000

det(A^T * A):26
(A^T * A)^{-1}:
   0.6538   0.4615
   0.4615   0.3846

[P]:
   0.6538   0.4615  -0.1154
   0.4615   0.3846   0.1538
  -0.1154   0.1538   0.9615

The orthogonal projection of a point P0 (2,1,-1) on plane W with equation 3x-4y+z=0 is
[P] * P0 :
   1.8846
   1.1538
  -1.0385

The  distance between the point P0 (2,1,-1) and the plane W with equation 3x-4y+z=0 is
d :0.196116
D :0.196116
```

**Figure 23.241:** *The computation to obtain the standard matrix for the orthogonal projection on W, the orthogonal projection of a point $P_0(2,1,-1)$ on W, and the distance between the orthogonal projection on W and point $P_0$ itself (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection on a Plane from a Point and Distance Computation/main.cpp**).*

## XIX. Least Squares Fitting to Data

[DF*] A common problem in experimental work is to obtain a mathematical relationship $y = f(x)$ between two variables $x$ and $y$ by "fitting" a curve to points in the plane corresponding to various experimentally determined values of $x$ and $y$, say

$$(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$$

On the basis of theoretical considerations or simply after observing the pattern of the points, the experimenter decides on the general form of the curve $y = f(x)$ to be fitted. Some possibilities are

(a) A straight line

$$y = a + bx$$

(b) A quadratic polynomial

$$y = a + bx + cx^2$$

(c) A cubic polynomial

$$y = a + bx + cx^2 + dx^3$$

Because the points are obtained experimentally, there is often some measurement "error" in the data, making it impossible to find a curve of the desired form that passes through all the points. Thus, the idea is to choose the curve (by determining its coefficients) that "best" fits the data.

**Figure 23.242:** *The plot for plane W with the basis, point $P_0(2, 1, -1)$, the orthogonal projection of a point $P_0(2, 1, -1)$ on W, and the distance between the orthogonal projection on W and point $P_0$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/3D Plot Orthogonal Projection on a Plane from a Point and Distance Computation/main.cpp**).*

[DF*] Suppose we want to fit a straight line $y = a + bx$ to the experimentally determined points

$$(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$$

If the data were collinear, the line would pass through all $n$ points, and the unknown coefficients $a$ and $b$ would satisfy the equations

$$\begin{aligned} y_1 &= a + bx_1 \\ y_2 &= a + bx_2 \\ &\vdots \\ y_n &= a + bx_n \end{aligned}$$

we can write this system in matrix form as

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

or more compactly as

$$M\boldsymbol{v} = \boldsymbol{y} \tag{23.288}$$

where

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} a \\ b \end{bmatrix}$$

If the data points are not collinear, then it is impossible to find coefficients $a$ and $b$ that satisfy system $M\boldsymbol{v} = \boldsymbol{y}$ exactly; that is, the system is inconsistent. In this case we will look for a least square solution

$$\boldsymbol{v} = \boldsymbol{v}* = \begin{bmatrix} a^* \\ b^* \end{bmatrix}$$

we call a line $y = a^* + b^*x$ whose coefficients come from a least squares solution a regression line or a least squares straight line fit to the data. To explain this terminology, recall that a least squares solution of $M\boldsymbol{v} = \boldsymbol{y}$ minimizes

$$||\boldsymbol{y} - M\boldsymbol{v}|| \tag{23.289}$$

if we express the square of the equation above in terms of components, we obtain

$$||\boldsymbol{y} - M\boldsymbol{v}||^2 = (y_1 - a - bx_1)^2 + (y_2 - a - bx_2)^2 + \cdots + (y_n - a - bx_n)^2 \tag{23.290}$$

if we now let

$$d_1 = |y_1 - a - bx_1|, \quad |y_2 - a - bx_2|, \ldots \quad |y_n - a - bx_n|$$

then we can write

$$||\boldsymbol{y} - M\boldsymbol{v}||^2 = d_1^2 + d_2^2 + \cdots + d_n^2 \tag{23.291}$$

the number $d_i$ can be interpreted as the vertical distance between the line $y = ax + b$ and the data point $(x_i, y_i)$. This distance is a measure of the "error" at the point $(x_i, y_i)$ resulting

from the inexact fit of $y = a + bx$ to the data points, the assumption being that the $x_i$ are known exactly and that all the error is in the measurement of the $y_i$.
Since $||\boldsymbol{y} - M\boldsymbol{v}||$ and $||\boldsymbol{y} - M\boldsymbol{v}||^2$ are minimized by the same vector $\boldsymbol{v}*$, the least squares straight line fit minimizes the sum of the squares of the estimated errors $d_i$, hence the name least squares straight line fit.

**Figure 23.243:** *The illustration for least squares straight line fit, $d_i$ measures the vertical error in the least squares straight line fit.*

[DF*] The least squares solutions of

$$M\boldsymbol{v} = \boldsymbol{y}$$

can be obtained by soving the associated normal system

$$M^T M\boldsymbol{v} = M^T\boldsymbol{y}$$

the equations of which are called the normal equations. The least squares solution is unique and is given by

$$\boldsymbol{v}^* = (M^T M)^{-1} M^T \boldsymbol{y}$$

**Theorem 23.158: Uniqueness of the Least Squares Solution**

Let $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ be a set of two or more data points, not all lying on a vertical line, and let we can write this system in matrix form as

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Then there is a unique least squares straight line fit

$$y = a^* + b^* x$$

to the data points. Moreover,

$$\boldsymbol{v}* = \begin{bmatrix} a^* \\ b^* \end{bmatrix}$$

is given by the formula

$$\boldsymbol{v}^* = (M^T M)^{-1} M^T \boldsymbol{y} \tag{23.292}$$

which expresses the fact that $\boldsymbol{v} = \boldsymbol{v}*$ is the unique solution of the normal equations

$$M^T M \boldsymbol{v} = M^T \boldsymbol{y} \tag{23.293}$$

[DF*] The technique described for fitting a straight line to data points can be generalized to fitting a polynomial of specified degree to data points. Let us attempt to fit a polynomial of fixed degree $m$

$$y = a_0 + a_1 x + \ldots a_m x^m \tag{23.294}$$

to $n$ points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

Substituting these $n$ values of $x$ and $y$ into $y$ yields the $n$ equations

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 + \ldots a_m x_1^m \\ y_1 &= a_0 + a_1 x_1 + \ldots a_m x_1^m \\ &\vdots \\ y_1 &= a_0 + a_1 x_1 + \ldots a_m x_1^m \end{aligned}$$

or, in matrix form,

$$\boldsymbol{y} = M\boldsymbol{v}$$

where

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad M = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^m \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix}$$

As before, the solutions of the normal equations

$$M^T M \boldsymbol{v} = M^T \boldsymbol{y}$$

determine the coefficients of the polynomial, and the vector $\boldsymbol{v}$ minimizes

$$\boldsymbol{y} - M\boldsymbol{v}$$

If $M^T M$ is invertible, then the normal equations have a unique solution

$$\boldsymbol{v} = \boldsymbol{v}*$$

which is given by

$$\boldsymbol{v}^* = (M^T M)^{-1} M^T \boldsymbol{y} \tag{23.295}$$

## i. C++ Plot and Computation: Least Squares Straight Line Fit

After we do some observation or experimentation and obtain the observation data of $x$ and $y$, such as

$$(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$$

we can construct a straight line that can fit the data obtained with minimum error, it is called least square fit with straight line. The straight line has the general form of

$$y = a + bx \tag{23.296}$$

We are going to use OpenGL, GLEW and GLUT, as it is really helpful and we can do a lot of manipulation on the resulting plot, like changing from line to points or move the screen, zoom in or out, we can even use texture and make the plot more colorful. Compared to Gnuplot, we have more freedom with more codes to write, it is like creating the backend of something like Gnuplot ourselves.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>

#include <GL/glew.h>
#include <GL/glut.h>

#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "../common/shader_utils.h"

#include "res_texture.c"

GLuint program;
GLint attribute_coord2d;
GLint uniform_offset_x;
GLint uniform_offset_y;
GLint uniform_scale_x;
GLint uniform_scale_y;
GLint uniform_sprite;
GLuint texture_id;
GLint uniform_mytexture;

float offset_x = 0.0;
```

```
float offset_y = 0.0;
float scale_x = 1.0;
float scale_y = 1.0;
int mode = 0;

struct point {
	GLfloat x;
	GLfloat y;
};

const int N = 3;

GLuint vbo;

using namespace std;

// Function to calculate b
double calculateB(int x[], int y[], int n)
{
	// sum of array x
	int sx = accumulate(x, x + n, 0);
	// sum of array y
	int sy = accumulate(y, y + n, 0);

	// for sum of product of x and y
	int sxsy = 0;

	// sum of square of x
	int sx2 = 0;
	for(int i = 0; i < n; i++)
	{
		sxsy += x[i] * y[i];
		sx2 += x[i] * x[i];
	}
	double b = (double)(n * sxsy - sx * sy) /
	(n * sx2 - sx * sx);

	return b;
}

double calculateA(int X[], int Y[], int n)
{
	double b = calculateB(X, Y, n);

	int meanX = accumulate(X, X + n, 0) / n;
	int meanY = accumulate(Y, Y + n, 0) / n;

	// Calculating a
```

```
        double a = meanY — b * meanX;

        return a;
}

// Function to find the least regression line
void leastRegLine(int X[], int Y[], int n)
{
        // Finding b
        double b = calculateB(X, Y, n);

        int meanX = accumulate(X, X + n, 0) / n;
        int meanY = accumulate(Y, Y + n, 0) / n;

        // Calculating a
        double a = meanY — b * meanX;

        // Printing regression line
        cout << ("Regression line:") << endl;
        cout << ("Y = ");
        printf("%.3f + ", a);
        printf("%.3f *X", b);
}

int* vecx() {
        static int x[N];
        std::fstream in("vectorX.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
        return x;
}

int* vecy() {
        static int y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}
```

```
int init_resources() {
        program = create_program("graph.v.glsl", "graph.f.glsl");
        if (program == 0)
        return 0;

        attribute_coord2d = get_attrib(program, "coord2d");
        uniform_offset_x = get_uniform(program, "offset_x");
        uniform_offset_y = get_uniform(program, "offset_y");
        uniform_scale_x = get_uniform(program, "scale_x");
        uniform_scale_y = get_uniform(program, "scale_y");
        uniform_sprite = get_uniform(program, "sprite");
        uniform_mytexture = get_uniform(program, "mytexture");

        if (attribute_coord2d == -1 || uniform_offset_y == -1 ||
            uniform_offset_x == -1 || uniform_scale_y == -1 ||
            uniform_scale_x == -1 || uniform_sprite == -1 ||
            uniform_mytexture == -1)
        return 0;

        /* Enable blending */
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        /* Enable point sprites (not necessary for true OpenGL ES 2.0) */
        #ifndef GL_ES_VERSION_2_0
        glEnable(GL_POINT_SPRITE);
        glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
        #endif

        /* Upload the texture for our point sprites */
        glActiveTexture(GL_TEXTURE0);
        glGenTextures(1, &texture_id);
        glBindTexture(GL_TEXTURE_2D, texture_id);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, res_texture.width,
            res_texture.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, res_texture.
            pixel_data);

        // Create the vertex buffer object
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);

        // Take the vector data from textfile as pointer
        int* ptrvecx;
        ptrvecx = vecx();
        int* ptrvecy;
        ptrvecy = vecy();
```

```
        int X[N]; // declare an array name X with size of N
        int Y[N];

        for (int i = 0; i < N; ++i)
        {
                X[i] = ptrvecx[i];
                Y[i] = ptrvecy[i];
        }
        int n = N;
        double a = calculateA(X, Y, n);
        double b = calculateB(X, Y, n);

        // Create our own temporary buffer
        //point graph[2000]; // to draw regression line

        // Fill it in just like an array, to draw the points
        // to draw the regression line
        for (int i = 0; i < 2000; i++) {
                float x = (i -1000) / 10.0;
                // graph[i].x = x;
                // graph[i].y = a + b*x;
        }

        point graph[N]; // to draw points
        for (int i = 0; i < N; i++) {
                graph[i].x = ptrvecx[i];
                graph[i].y = ptrvecy[i];
        }


        // Tell OpenGL to copy our array to the buffer object
        glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph, GL_STATIC_DRAW);

        return 1;
}

void display() {
        glUseProgram(program);
        glUniform1i(uniform_mytexture, 0);

        glUniform1f(uniform_offset_x, offset_x);
        glUniform1f(uniform_offset_y, offset_y);
        glUniform1f(uniform_scale_x, scale_x);
        glUniform1f(uniform_scale_y, scale_y);

        glClearColor(0.0, 0.0, 0.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);
```

```
        /* Draw using the vertices in our vertex buffer object */
        glBindBuffer(GL_ARRAY_BUFFER, vbo);

        glEnableVertexAttribArray(attribute_coord2d);
        glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0,
            0);

        /* Push each element in buffer_vertices to the vertex shader */
        switch (mode) {
                case 0:
                glUniform1f(uniform_sprite, 0);
                glDrawArrays(GL_LINE_STRIP, 0, 2000);
                break;
                case 1:
                glUniform1f(uniform_sprite, 1);
                glDrawArrays(GL_POINTS, 0, 2000);
                break;
                case 2:
                glUniform1f(uniform_sprite, res_texture.width);
                glDrawArrays(GL_POINTS, 0, 2000);
                break;
        }

        glutSwapBuffers();
}

void special(int key, int x, int y) {
        switch (key) {
                case GLUT_KEY_F1:
                mode = 0;
                printf("Now drawing using lines.\n");
                break;
                case GLUT_KEY_F2:
                mode = 1;
                printf("Now drawing using points.\n");
                break;
                case GLUT_KEY_F3:
                mode = 2;
                printf("Now drawing using point sprites.\n");
                break;
                case GLUT_KEY_LEFT:
                offset_x += 0.1;
                break;
                case GLUT_KEY_RIGHT:
                offset_x -= 0.1;
                break;
                case GLUT_KEY_UP:
```

```
                offset_y -= 0.1;
                break;
                case GLUT_KEY_DOWN:
                offset_y += 0.1;
                break;
                case GLUT_KEY_F4:
                scale_x += 0.1;
                break;
                case GLUT_KEY_F5:
                scale_x -= 0.1;
                break;
                case GLUT_KEY_F6:
                scale_y += 0.1;
                break;
                case GLUT_KEY_F7:
                scale_y -= 0.1;
                break;
                case GLUT_KEY_HOME:
                offset_x = 0.0;
                offset_y = 0.0;
                scale_x = 0.1;
                scale_y = 0.1;
                break;
        }

        glutPostRedisplay();
}

void free_resources() {
        glDeleteProgram(program);
}

int main(int argc, char *argv[]) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB);
        glutInitWindowSize(1024, 600);
        glutCreateWindow("Least Square Regression Line");

        GLenum glew_status = glewInit();

        if (GLEW_OK != glew_status) {
                fprintf(stderr, "Error: %s\n", glewGetErrorString(glew_status
                    ));
                return 1;
        }

        if (!GLEW_VERSION_2_0) {
                fprintf(stderr, "No support for OpenGL 2.0 found\n");
```

```
        return 1;
    }

    int* ptrvecx;
    ptrvecx = vecx();
    int* ptrvecy;
    ptrvecy = vecy();

    cout << "Vector / Data X: " << setw(30) << "Vector / Data Y: " <<
        endl;
    for (int i = 0; i < N; ++i)
    {
        cout <<ptrvecx[i]<< "\t \t \t \t" << ptrvecy[i] << endl;
    }
    cout << endl;
    // Statistical data
    int X[N]; // declare an array name X with size of N
    int Y[N];

    for (int i = 0; i < N; ++i)
    {
        X[i] = ptrvecx[i];
        Y[i] = ptrvecy[i];
    }
    int n = N; //int n = sizeof(X) / sizeof(X[0]);
    double a = calculateA(X, Y, n);
    cout << "a = " << a << endl;
    cout << endl;
    double b = calculateB(X, Y, n);
    cout << "b = " << b << endl;
    cout << endl;
    leastRegLine(X, Y, n);
    cout << endl;

    GLfloat range[2];

    glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, range);
    if (range[1] < res_texture.width)
    fprintf(stderr, "WARNING: point sprite range (%f, %f) too small\n",
        range[0], range[1]);

    printf("Use left/right to move horizontally.\n");
    printf("Use up/down to move vertically.\n");
    printf("Press home to reset the position and scale.\n");
    printf("Press F1 to draw lines.\n");
    printf("Press F2 to draw points.\n");
    printf("Press F3 to draw point sprites.\n");
    printf("Press F4 to zoom in /F5 to zoom out the horizontal scale.\n"
```

```
                );
        printf("Press F6 to zoom in /F7 to zoom out the vertical scale.\n");

        if (init_resources()) {
                glutDisplayFunc(display);
                glutSpecialFunc(special);
                glutMainLoop();
        }

        free_resources();
        return 0;
}
```

**C++ Code 247:** *main.cpp "Least Square Straight Line Fit"*

To compile it, type:
**make**
**./main**

Explanation for the codes:

- Inside the function **int init_resources()** we draw the points from the vector $X$ and vector $Y$ data, if you want to draw the regression line / straight line fit then comment the second **for** loop and uncomment the first **for** loop, then compile and run the program again.

```
        int* ptrvecx;
        ptrvecx = vecx();
        int* ptrvecy;
        ptrvecy = vecy();

        int X[N]; // declare an array name X with size of N
        int Y[N];

        for (int i = 0; i < N; ++i)
        {
                X[i] = ptrvecx[i];
                Y[i] = ptrvecy[i];
        }
        int n = N;
        double a = calculateA(X, Y, n);
        double b = calculateB(X, Y, n);

        // Create our own temporary buffer
        //point graph[2000]; // to draw regression line

        // Fill it in just like an array, to draw the points
        // to draw the regression line
        for (int i = 0; i < 2000; i++) {
                float x = (i -1000) / 10.0;
                // graph[i].x = x;
```

```
            // graph[i].y = a + b*x;
        }

        point graph[N]; // to draw points
        for (int i = 0; i < N; i++) {
            graph[i].x = ptrvecx[i];
            graph[i].y = ptrvecy[i];
        }
```

**Figure 23.244:** *The computation and plot for the data points taken from textfile* ***vectorX.txt*** *and* ***vectorY.txt*** *with OpenGL you can move the screen, zoom in and zoom out horizontally or vertically, furthermore you can even change into viewing line or points. (****DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Gaussian Elimination/main.cpp****).*

## ii. C++ Computation: Least Squares Straight Line Fit with Armadillo

This example is taken from chapter 6.5 of [11].
Hooke's law in physics states that the length $x$ of a uniform spring is a linear function of the force $y$ applied to it. We express this relationship as

$$y = ax + b$$

the coefficient $b$ is called the spring constant. Suppose a particular unstretched spring has a measured length of 6.1 inches (i.e. $x = 6.1$ when $y = 0$). Forces of 2 pounds, 4 pounds, and 6 pounds are then applied to the spring, and the corresponding lengths are found to be 7.6 inches, 8.7 inches, and 10.4 inches. Find the spring constant.
**Solution:**
We have

$$M = \begin{bmatrix} 1 & 6.1 \\ 1 & 7.6 \\ 1 & 8.7 \\ 1 & 10.5 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 0 \\ 2 \\ 4 \\ 6 \end{bmatrix}$$

and the solution is

$$\boldsymbol{v}* = \begin{bmatrix} a^* \\ b^* \end{bmatrix} = (M^T M)^{-1} M^T \boldsymbol{y} \approx \begin{bmatrix} -8.6430 \\ 1.4199 \end{bmatrix}$$

The least square straight line fit will be

$$y = -8.643 + 1.42x$$

Thus, the estimated value of the spring constant is $b* \approx 1.4199$ pounds/inch.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

const int N = 4;
const int C1 = 2;
const int C2 = 4;

float* vecx() {
       static float x[N];
       std::fstream in("vectorX.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
              in >> vectortiles[i];
              x[i] =vectortiles[i];
```

```
        }
        return x;
}

float* vecy() {
        static float y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}

void displayfloat(float mat[N][C1], int row, int col)
{
        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < col; j++)
                cout<<setw(8) << fixed << setprecision(3) << mat[i][j]<<"\t";

                cout<<endl;
        }
        cout<<endl;
}

// Driver code
int main(int argc, char** argv)
{
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat M(N,C1,fill::zeros);
        arma::mat M_transpose(C1,C2,fill::zeros);
        arma::mat Y(N,1,fill::zeros);
        arma::mat V(N,1,fill::zeros);

        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C1; ++j)
                {
                        M[i+j*N] = pow(ptrvecx[i],j);
                }
        }
```

```
        M_transpose = M.t();

        for (int i = 0; i < N; ++i)
        {
            Y[i] = ptrvecy[i];
        }
        V = arma::inv(M_transpose * M)*M_transpose * Y ;

        M.print("M:");
        cout << endl;
        M_transpose.print("M^T:");
        cout << endl;
        Y.print("y:");
        cout << endl;

        // determinant and inverse from Armadillo
        cout << "det(M^T * M): " << det(M_transpose*M) << endl;
        cout << endl;
        cout << "inv(M^T * M): " << endl << arma::inv(M_transpose * M) <<
            endl;
        cout << "inv(M^T * M) * M^T: " << endl << arma::inv(M_transpose * M)
            *M_transpose << endl;
        cout << "v* = inv(M^T * M) * M^T * y : " << endl << arma::inv(
            M_transpose * M)*M_transpose * Y << endl;
        cout << "Least Square Straight Line Fit :" << endl;
        cout << ("y = ");
        printf("%.3f + ", V[0]);
        printf("%.3f x ", V[1]);
        cout <<endl;
    }
```

**C++ Code 248:** *main.cpp "Least Squares Straight Line Fit with Armadillo"*

To compile it, type:
**g++ -o result main.cpp -larmadillo**
**./result**

or with the Makefile, type:
**make**
**./main**

```
xterm
mbolicC++/ch23-Numerical Linear Algebra/Least Square Straight Line (Not yet, use
 Armadillo) ]# ./main
M:
    1.0000    6.1000
    1.0000    7.6000
    1.0000    8.7000
    1.0000   10.4000

M^T:
    1.0000    1.0000    1.0000    1.0000
    6.1000    7.6000    8.7000   10.4000

y:
         0
   2.0000
   4.0000
   6.0000

det(M^T * M): 39.44

inv(M^T * M):
   7.0695  -0.8316
  -0.8316   0.1014

inv(M^T * M) * M^T:
   1.9965   0.7490  -0.1658  -1.5796
  -0.2130  -0.0609   0.0507   0.2231

v* = inv(M^T * M) * M^T * y :
  -8.6430
   1.4199

Least Square Straight Line Fit :
y = -8.643 + 1.420 x
```

**Figure 23.245:** *The computation to obtain least squares straight line fit with Armadillo and C++. (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Least Square Straight Line with Armadillo/main.cpp**).*

## iii. C++ Computation: Least Squares Quadratic Polynomial Fit

Consider that we obtain the observation data of $x$ and $y$, such as

$$(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$$

we can construct a quadratic polynomial (a polynomial of degree at most 2) that can fit the data that we obtained with minimum error, it is called least squares fit with quadratic polynomial. The quadratic polynomial has the general form of

$$y = a + bx + cx^2 \tag{23.297}$$

For the test case, we are going to use example 3 from chapter 6.5 of [11].
According to Newton's second law of motion, a body near the Earth's surface falls vertically downward according to the equation

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

where $s$ = vertical displacement downward relative to some fixed point
$s_0$ = initial displacement at time $t = 0$
$v_0$ = initial velocity at time $t = 0$
$g$ = acceleration of gravity at the Earth's surface

from the equation above by releasing a weight with unknown initial displacement and velocity and measuring the distance it has fallen at certain times relative to a fixed reference point.

Suppose that a laboratory experiment is performed to evaluate $g$. Suppose it is found that at times $t = 0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5$ seconds the weight has fallen $s = -0.18,\ 0.31,\ 1.03,\ 2.48,\ 3.73$ feet, respectively, from the reference point. Find an approximate value of $g$ using these data.

**Solution:**

The mathematical problem is to fit a quadratic curve

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

to the five data points:

$$(0.1, -0.18), \quad (0.2, 0.32), \quad (0.3, 1.03), \quad (0.4, 2.48), \quad (0.5, 3.73)$$

With the appropriate adjustments in notation, the matrices $M$ and $\boldsymbol{y}$ are

$$M = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \end{bmatrix} = \begin{bmatrix} 1 & 0.1 & 0.01 \\ 1 & 0.2 & 0.04 \\ 1 & 0.3 & 0.09 \\ 1 & 0.4 & 0.16 \\ 1 & 0.5 & 0.25 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix} = \begin{bmatrix} -0.18 \\ 0.31 \\ 1.03 \\ 2.48 \\ 3.73 \end{bmatrix}$$

and the solution is

$$\boldsymbol{v}* = \begin{bmatrix} a_0^* \\ a_1^* \\ a_2^* \end{bmatrix} = (M^T M)^{-1} M^T \boldsymbol{y} \approx \begin{bmatrix} -0.3980 \\ 0.3471 \\ 16.0714 \end{bmatrix}$$

The least square quadratic polynomial fit will be

$$y = -0.3980 + 0.3471x + 16.0714x^2$$

From the equation at the beginning, we have set $a_2^* = 2g$, so the estimated value of $g$ is

$$g = 2a_2^* = 2(16.0714) = 32.2$$

the estimated gravity, $g$ is 32.2 $feet/second^2$. We can also estimate the initial displacement and initial velocity of the weight:

$$s_0 = a_0* = -0.3980$$
$$v_0 = a_1* = 0.3471$$

the initial displacement estimation is $-0.3980$ $feet$, and the initial velocity is 0.3471 $feet/second$.

A bit of improvement from the previous section. This time we are going to use **Armadillo** in the C++ code for this section, since on January 26th, 2024 we compare the algorithm created in website as function in C++ cannot compute inverse as good as Armadillo computing inverse, when you take a look at chapter 1.9 of [11] and try to compute the inverse of the matrix $(I - C)$ with entries of decimals and negative float numbers, **Armadillo** able to compute it precisely like the answer on the book, while a self-made function made by volunteer on website unable to obtain the inverse correctly.

This will come in handy as **Armadillo** can be the 'de facto' C++ library for Linear Algebra, the same as 'odeint' used to be the 'de facto' C++ library for ordinary differential equation

```
M:
   0.5000  -0.1000  -0.1000
  -0.2000   0.5000  -0.3000
  -0.1000  -0.3000   0.6000

det(M): 0.079

inv(M):
   2.6582   1.1392   1.0127
   1.8987   3.6709   2.1519
   1.3924   2.0253   2.9114

root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot Sy
mbolicC++/Armadillo ]#
```

```
mbolicC++/ch23-Numerical Linear Algeb
ain
The input matrix is :
   0.5    -0.1    -0.1
  -0.2     0.5    -0.3
  -0.1    -0.3     0.6

The matrix determinant is :
0

The Inverse is :
can't find its inverse
root [ ~/latex/DianFreya Math Physics
mbolicC++/ch23-Numerical Linear Algeb
```

**Figure 23.246:** *The comparison between Armadillo computing inverse of matrix $(I - C)$ from chapter 1.9 of [11] (on the left), and the other is a self-made function made by volunteer on some website in the internet (on the right).*

computation.

We prepare the data input as two vectors, which are $X$ (the independent variable, the time the weight has fallen, $t$) and $Y$ (the dependent variable, the distance the weight has fallen, $s$), that we obtain from observation or experimentation. These vectors are saved as texfile named **vectorX.txt** and **vectorY.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h>
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

const int N = 5;
const int C1 = 3;
const int C2 = 5;

float* vecx() {
       static float x[N];
       std::fstream in("vectorX.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
              in >> vectortiles[i];
              x[i] =vectortiles[i];
       }
       return x;
}

float* vecy() {
```

```
        static float y[N];
        std::fstream in("vectorY.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}

// Driver code
int main(int argc, char** argv)
{
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat M(N,C1,fill::zeros);
        arma::mat M_transpose(C1,C2,fill::zeros);
        arma::mat Y(N,1,fill::zeros);
        arma::mat V(N,1,fill::zeros);

        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C1; ++j)
                {
                        M[i+j*N] = pow(ptrvecx[i],j);
                }
        }

        M_transpose = M.t();

        // Fill vector y from Armadillo from loading textfile vectorY.txt
        for (int i = 0; i < N; ++i)
        {
                Y[i] = ptrvecy[i];
        }
        V = arma::inv(M_transpose * M)*M_transpose * Y ;

        M.print("M:");
        cout << endl;
        M_transpose.print("M^T:");
        cout << endl;
        Y.print("y:");
        cout << endl;
```

```
            // determinant and inverse from Armadillo
            cout << "det(M^T * M): " << det(M_transpose*M) << endl;
            cout << endl;
            cout << "inv(M^T * M): " << endl << arma::inv(M_transpose * M) <<
                endl;
            cout << "inv(M^T * M) * M^T: " << endl << arma::inv(M_transpose * M)
                *M_transpose << endl;
            cout << "v* = inv(M^T * M) * M^T * y : " << endl << arma::inv(
                M_transpose * M)*M_transpose * Y << endl;
            cout << "Least Square Quadratic Polynomial Fit :" << endl;
            cout << ("y = ");
            printf("%.3f + ", V[0]);
            printf("%.3f x + ", V[1]);
            printf("%.3f x^2 ", V[2]);
            cout <<endl;
        }
```

**C++ Code 249:** *main.cpp "Least Squares Quadratic Polynomial Fit"*

To compile it, type:
**g++ -o result main.cpp -larmadillo**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- We are taking the vector $X$ from textfile and make a vandermonde matrix of size $5 \times 3$ out of it, we declare the matrix as **arma::mat M(N,C1,fill::zeros);** , a 2 dimensional array.
  Then, we create the transpose for **M**, we declare the transpose as **arma::mat M_transpose(C1,C2,fill::zeros)**.

```
            arma::mat M(N,C1,fill::zeros);
            arma::mat M_transpose(C1,C2,fill::zeros);
            arma::mat Y(N,1,fill::zeros);
            arma::mat V(N,1,fill::zeros);
            // Fill matrix M from Armadillo with matrix mat from loading
                textfile
            for (int i = 0; i < N; ++i)
            {
                for(int j = 0; j<C1; ++j)
                {
                    M[i+j*N] = pow(ptrvecx[i],j);
                }
            }

            M_transpose = M.t();
```

- From the previously created array, we fill the matrix class of Armadillo, here are the matrices created from Armadillo library:

1. Matrix $M$ as **arma::mat M**
2. Matrix $M^T$ as **arma::mat M_transpose**
3. Matrix $Y$ as **arma::mat Y**, it is a vector declared as matrix with column of 1. We fill the data directly from the textfile input of **vectorY.txt**
4. Matrix $V$ as **arma::mat V**, it is a vector declared as matrix with column of 1. This is the computation for the solution vector. We follow this equation:

$$\boldsymbol{v}* = (M^T M)^{-1} M^T \boldsymbol{y}$$

we will obtain the least squares solution that will become the coefficients for the quadratic polynomial

$$\boldsymbol{v}* = \begin{bmatrix} a^* \\ b^* \\ c^* \end{bmatrix}$$

the quadratic polynomial that fit the experimental data is

$$y = a^* + b^* x + c^* x^2$$

```
M:
   1.0000   0.1000   0.0100
   1.0000   0.2000   0.0400
   1.0000   0.3000   0.0900
   1.0000   0.4000   0.1600
   1.0000   0.5000   0.2500

M^T:
   1.0000   1.0000   1.0000   1.0000   1.0000
   0.1000   0.2000   0.3000   0.4000   0.5000
   0.0100   0.0400   0.0900   0.1600   0.2500

y:
  -0.1800
   0.3100
   1.0300
   2.4800
   3.7300

det(M^T * M): 0.0007

inv(M^T * M):
   4.6000e+00  -3.3000e+01   5.0000e+01
  -3.3000e+01   2.6714e+02  -4.2857e+02
   5.0000e+01  -4.2857e+02   7.1429e+02

inv(M^T * M) * M^T:
   1.8000e+00  -3.3634e-08  -8.0000e-01  -6.0000e-01   6.0000e-01
  -1.0571e+01   3.2857e+00   8.5714e+00   5.2857e+00  -6.5714e+00
   1.4286e+01  -7.1429e+00  -1.4286e+01  -7.1429e+00   1.4286e+01

v* = inv(M^T * M) * M^T * y :
   -0.3980
    0.3471
   16.0714

Least Square Quadratic Polynomial Fit :
y = -0.398 + 0.347 x + 16.071 x^2
```

**Figure 23.247:** *The computation for least squares fit of a quadratic polynomial with C++ and Armadillo. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Least Square Quadratic Polynomial with Armadillo/main.cpp**).*

## iv. C++ Plot and Computation: Least Squares Quadratic Polynomial Fit

This is continuing from the previous section, we create the plot after we compute the least squares quadratic polynomial that fit the data points.

$$y = a^* + b^* x + c^* x^2$$

the five data points are:

$$(0.1, -0.18), \quad (0.2, 0.32), \quad (0.3, 1.03), \quad (0.4, 2.48), \quad (0.5, 3.73)$$

we already know that the least square quadratic polynomial fit will be

$$y = -0.3980 + 0.3471x + 16.0714x^2$$

The computation stays the same, with Armadillo, but remember the placement of the computations that are calling Armadillo should be placed in the correct function. The plot will be using OpenGL, GLUT and GLEW. The plus points of using OpenGL and its siblings is we can do more with the plot, such as zooming in, zooming out, changing the plot, from data points to the least squares quadratic polynomial line plot in an instant with just one click.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

#include <GL/glew.h>
#include <GL/glut.h>

#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "../common/shader_utils.h"

#include "res_texture.c"

GLuint program;
GLint attribute_coord2d;
GLint uniform_offset_x;
GLint uniform_offset_y;
GLint uniform_scale_x;
GLint uniform_scale_y;
GLint uniform_sprite;
```

```
GLuint texture_id;
GLint uniform_mytexture;

float offset_x = 0.0;
float offset_y = 0.0;
float scale_x = 1.0;
float scale_y = 1.0;
int mode = 0;

struct point {
	GLfloat x;
	GLfloat y;
};

const int N = 5;
const int C1 = 3;
const int C2 = 5;

GLuint vbo;

using namespace std;
using namespace arma;

float* vecx() {
	static float x[N];
	std::fstream in("vectorX.txt");
	float vectortiles[N];
	for (int i = 0; i < N; ++i)
	{
		in >> vectortiles[i];
		x[i] =vectortiles[i];
	}
	return x;
}

float* vecy() {
	static float y[N];
	std::fstream in("vectorY.txt");
	float vectortiles[N];
	for (int i = 0; i < N; ++i)
	{
		in >> vectortiles[i];
		y[i] =vectortiles[i];
	}
	return y;
}

int init_resources() {
```

```
program = create_program("graph.v.glsl", "graph.f.glsl");
if (program == 0)
return 0;

attribute_coord2d = get_attrib(program, "coord2d");
uniform_offset_x = get_uniform(program, "offset_x");
uniform_offset_y = get_uniform(program, "offset_y");
uniform_scale_x = get_uniform(program, "scale_x");
uniform_scale_y = get_uniform(program, "scale_y");
uniform_sprite = get_uniform(program, "sprite");
uniform_mytexture = get_uniform(program, "mytexture");

if (attribute_coord2d == -1 || uniform_offset_y == -1 ||
    uniform_offset_x == -1 || uniform_scale_y == -1 ||
    uniform_scale_x == -1 || uniform_sprite == -1 ||
    uniform_mytexture == -1)
return 0;

/* Enable blending */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

/* Enable point sprites (not necessary for true OpenGL ES 2.0) */
#ifndef GL_ES_VERSION_2_0
glEnable(GL_POINT_SPRITE);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
#endif

/* Upload the texture for our point sprites */
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, &texture_id);
glBindTexture(GL_TEXTURE_2D, texture_id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, res_texture.width,
    res_texture.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, res_texture.
    pixel_data);

// Create the vertex buffer object
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Take the vector data from textfile as pointer
float* ptrvecx;
ptrvecx = vecx();
float* ptrvecy;
ptrvecy = vecy();
```

```
        point graph[N];
        for (int i = 1; i < N; i++)
        {
                graph[i].x = ptrvecx[i];
                graph[i].y = ptrvecy[i];
        }

        // Tell OpenGL to copy our array to the buffer object
        glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph, GL_STATIC_DRAW);

        return 1;
}

void display() {
        glUseProgram(program);
        glUniform1i(uniform_mytexture, 0);

        glUniform1f(uniform_offset_x, offset_x);
        glUniform1f(uniform_offset_y, offset_y);
        glUniform1f(uniform_scale_x, scale_x);
        glUniform1f(uniform_scale_y, scale_y);

        glClearColor(0.0, 0.0, 0.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);

        /* Draw using the vertices in our vertex buffer object */
        glBindBuffer(GL_ARRAY_BUFFER, vbo);

        glEnableVertexAttribArray(attribute_coord2d);
        glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0,
            0);

        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat M(N,C1,fill::zeros);
        arma::mat M_transpose(C1,C2,fill::zeros);
        arma::mat Y(N,1,fill::zeros);
        arma::mat V(N,1,fill::zeros);

        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C1; ++j)
                {
                        M[i+j*N] = pow(ptrvecx[i],j);
                }
```

```
        }

        M_transpose = M.t();

        for (int i = 0; i < N; ++i)
        {
               Y[i] = ptrvecy[i];
        }
        V = arma::inv(M_transpose * M)*M_transpose * Y ;
        /* Push each element in buffer_vertices to the vertex shader */
        switch (mode) {
               case 0:
               glClearColor(0.0, 0.0, 0.0, 0.0);
               glClear(GL_COLOR_BUFFER_BIT);
               glGenBuffers(1, &vbo);
               glBindBuffer(GL_ARRAY_BUFFER, vbo);
               glEnableVertexAttribArray(attribute_coord2d);
               glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
                   GL_FALSE, 0, 0);

               point graph[N];
               for (int i = 1; i < N; i++)
               {
                      graph[i].x = ptrvecx[i];
                      graph[i].y = ptrvecy[i];
               }

               glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph,
                   GL_STATIC_DRAW);
               glUniform1f(uniform_sprite, res_texture.width);
               glDrawArrays(GL_POINTS, 0, 2000);
               break;
               case 1:
               glUniform1f(uniform_sprite, 1);
               glDrawArrays(GL_POINTS, 0, 2000);
               break;
               case 2:
               glUniform1f(uniform_sprite, res_texture.width);
               glDrawArrays(GL_POINTS, 0, 2000);
               break;
               case 3:
               glClearColor(0.0, 0.0, 0.0, 0.0);
               glClear(GL_COLOR_BUFFER_BIT);

               glGenBuffers(1, &vbo);
               glBindBuffer(GL_ARRAY_BUFFER, vbo);
               glEnableVertexAttribArray(attribute_coord2d);
               glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
```

```
                    GL_FALSE, 0, 0);

                point graphquadraticfit[2000];
                for (int i = 0; i < 2000; i++)
                {
                        float x = (i -1000) / 10.0;
                        graphquadraticfit[i].x = x;
                        graphquadraticfit[i].y = V[0] + V[1]*x + V[2]*x*x;
                }

                glBufferData(GL_ARRAY_BUFFER, sizeof graphquadraticfit,
                    graphquadraticfit, GL_STATIC_DRAW);
                glUniform1f(uniform_sprite, 0);
                glDrawArrays(GL_LINE_STRIP, 0, 2000);
                break;
        }

        glutSwapBuffers();
}

void special(int key, int x, int y) {
        switch (key) {
                case GLUT_KEY_F1:
                mode = 0;
                printf("Now drawing the data points.\n");
                break;
                case GLUT_KEY_F2:
                mode = 1;
                printf("Now drawing using points.\n");
                break;
                case GLUT_KEY_F3:
                mode = 2;
                printf("Now drawing using point sprites.\n");
                break;
                case GLUT_KEY_F4:
                mode = 3;
                printf("Now drawing the quadratic polynomial fit.\n");
                break;
                case GLUT_KEY_LEFT:
                offset_x += 0.1;
                break;
                case GLUT_KEY_RIGHT:
                offset_x -= 0.1;
                break;
                case GLUT_KEY_UP:
                offset_y -= 0.1;
                break;
                case GLUT_KEY_DOWN:
```

```
                offset_y += 0.1;
                break;
                case GLUT_KEY_F5:
                scale_x += 0.1;
                break;
                case GLUT_KEY_F6:
                scale_x -= 0.1;
                break;
                case GLUT_KEY_F7:
                scale_y += 0.1;
                break;
                case GLUT_KEY_F8:
                scale_y -= 0.1;
                break;
                case GLUT_KEY_HOME:
                offset_x = 0.0;
                offset_y = 0.0;
                scale_x = 0.1;
                scale_y = 0.1;
                break;
        }

        glutPostRedisplay();
}

void free_resources() {
        glDeleteProgram(program);
}

int main(int argc, char *argv[]) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB);
        glutInitWindowSize(1024, 600);
        glutCreateWindow("Least Squares Quadratic Polynomial Fit");

        GLenum glew_status = glewInit();

        if (GLEW_OK != glew_status) {
                fprintf(stderr, "Error: %s\n", glewGetErrorString(glew_status
                    ));
                return 1;
        }

        if (!GLEW_VERSION_2_0) {
                fprintf(stderr, "No support for OpenGL 2.0 found\n");
                return 1;
        }
```

```
float* ptrvecx;
ptrvecx = vecx();
float* ptrvecy;
ptrvecy = vecy();

cout << "Vector / Data X: " << setw(30) << "Vector / Data Y: " <<
    endl;
for (int i = 0; i < N; ++i)
{
       cout <<ptrvecx[i]<< "\t \t \t \t" << ptrvecy[i] << endl;
}
cout << endl;

arma::mat M(N,C1,fill::zeros);
arma::mat M_transpose(C1,C2,fill::zeros);
arma::mat Y(N,1,fill::zeros);
arma::mat V(N,1,fill::zeros);

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<C1; ++j)
       {
              M[i+j*N] = pow(ptrvecx[i],j);
       }
}

M_transpose = M.t();

for (int i = 0; i < N; ++i)
{
       Y[i] = ptrvecy[i];
}
V = arma::inv(M_transpose * M)*M_transpose * Y ;

M.print("M:");
cout << endl;
M_transpose.print("M^T:");
cout << endl;
Y.print("y:");
cout << endl;

// determinant and inverse from Armadillo
cout << "det(M^T * M): " << det(M_transpose*M) << endl;
cout << endl;
cout << "inv(M^T * M): " << endl << arma::inv(M_transpose * M) <<
    endl;
cout << "inv(M^T * M) * M^T: " << endl << arma::inv(M_transpose * M)
    *M_transpose << endl;
```

```
        cout << "v* = inv(M^T * M) * M^T * y : " << endl << arma::inv(
            M_transpose * M)*M_transpose * Y << endl;
        cout << "Least Squares Quadratic Polynomial Fit :" << endl;
        cout << ("y = ");
        printf("%.3f + ", V[0]);
        printf("%.3f x + ", V[1]);
        printf("%.3f x^2 ", V[2]);
        cout <<endl;

        GLfloat range[2];

        glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, range);
        if (range[1] < res_texture.width)
        fprintf(stderr, "WARNING: point sprite range (%f, %f) too small\n",
            range[0], range[1]);

        printf("Use left/right to move horizontally.\n");
        printf("Use up/down to move vertically.\n");
        printf("Press home to reset the position and scale.\n");
        printf("Press F1 to draw data points.\n");
        printf("Press F2 to draw points.\n");
        printf("Press F3 to draw point sprites.\n");
        printf("Press F4 to draw the least squares quadratic polynomial fit
            .\n");
        printf("Press F5 to zoom in / F6 to zoom out the horizontal scale.\n
            ");
        printf("Press F7 to zoom in / F8 to zoom out the vertical scale.\n")
            ;

        if (init_resources()) {
               glutDisplayFunc(display);
               glutSpecialFunc(special);
               glutMainLoop();
        }

        free_resources();
        return 0;
    }
```

**C++ Code 250:** *main.cpp "Plot for Least Squares Quadratic Polynomial Fit"*

To compile it with the Makefile, type:
**make**
**./graph**

The codes are long but if you are following from the previous section it won't be too hard to grasp the idea and the logic to create the plot out of the computation done by **Armadillo** library.

**Figure 23.248:** *The plot of the data points, by pressing F1 we can see this plot with C++ and Armadillo. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Least Square Quadratic Polynomial Fit/main.cpp**).*

**Figure 23.249:** *The plot of the quadratic polynomial data fit, by pressing F4 we can see this plot with C++ and Armadillo. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Least Square Quadratic Polynomial Fit/main.cpp**).*

## v. C++ Computation: Least Squares Cubic Polynomial Fit

Consider that we obtain the observation data of $x$ and $y$ from observing the average price in a year for stocks price of ADHI (Adhi Karya), it is a state-owned construction corporation from Indonesia that is listed in IDX, the data is as follow:

| Year | Average Price |
|---|---|
| **2008** | 800 |
| **2009** | 100 |
| **2010** | 400 |
| **2011** | 700 |
| **2012** | 500 |
| **2013** | 3000 |
| **2014** | 1200 |
| **2015** | 2500 |
| **2016** | 2900 |
| **2017** | 2200 |
| **2018** | 1800 |
| **2019** | 1800 |
| **2020** | 1000 |
| **2021** | 1500 |
| **2022** | 700 |
| **2023** | 400 |

**Table 23.5:** *The average price of ADHI from 2008-2023 an Indonesian Stocks from IDX*

we can construct a cubic polynomial (a polynomial of degree at most 3) that can fit the data that we obtained with minimum error, it is called least squares fit with cubic polynomial. The cubic polynomial has the general form of

$$y = a + bx + cx^2 + dx^3 \tag{23.298}$$

Create a cubic polynomial using these data to see the pattern of ADHI stocks price.

**Solution:**

We have sixteen data points:

$$(1, 800), \quad (2, 100), \quad (3, 400), \quad \ldots, \quad (16, 400)$$

With the appropriate adjustments in notation, the matrices $M$ and $\boldsymbol{y}$ are

$$M = \begin{bmatrix} 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \\ 1 & t_3 & t_3^2 & t_3^3 \\ 1 & t_4 & t_4^2 & t_4^3 \\ 1 & t_5 & t_5^2 & t_5^3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ \vdots & \vdots & \vdots & \\ 1 & 16 & 256 & 4096 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{16} \end{bmatrix} = \begin{bmatrix} 800 \\ 100 \\ 400 \\ \vdots \\ 400 \end{bmatrix}$$

and the solution is

$$\boldsymbol{v}* = \begin{bmatrix} a_0^* \\ a_1^* \\ a_2^* \\ a_3^* \end{bmatrix} = (M^T M)^{-1} M^T \boldsymbol{y} \approx \begin{bmatrix} -81.868 \\ 239.141 \\ 19.774 \\ -2.125 \end{bmatrix}$$

Thus, the least square quadratic polynomial fit is

$$y = -81.868 + 239.141x + 19.774x^2 - 2.125x^3$$

We still use **Armadillo** to do the matrix computation till we obtain the least squares solution.

We prepare the data input as two vectors, which are $X$ (the independent variable, the year from 2008-2023, we number it 1 to 16, labeled here as $t$) and $Y$ (the dependent variable, the price of ADHI stocks at time $t$, labeled here as $s$), that we obtain from observing the stocks price. These vectors are saved as texfile named **vectorX.txt** and **vectorY.txt**.

```
#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

using namespace std;
using namespace arma;

const int N = 16;
const int C1 = 4;
const int C2 = 16;

float* vecx() {
       static float x[N];
       std::fstream in("vectorX.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
              in >> vectortiles[i];
              x[i] =vectortiles[i];
       }
       return x;
}

float* vecy() {
       static float y[N];
       std::fstream in("vectorY.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
```

```
                in >> vectortiles[i];
                y[i] =vectortiles[i];
        }
        return y;
}

// Driver code
int main(int argc, char** argv)
{
        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat M(N,C1,fill::zeros); //declare matrix M with size of N X
            C1 all have 0 entries with Armadillo.
        arma::mat M_transpose(C1,C2,fill::zeros); //declare matrix
            M_transpose with size of C1 X C2 all have 0 entries with
            Armadillo.
        arma::mat Y(N,1,fill::zeros);
        arma::mat V(N,1,fill::zeros);
        // Fill matrix M from Armadillo with matrix mat from loading
            textfile
        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C1; ++j)
                {
                        M[i+j*N] = pow(ptrvecx[i],j);
                }
        }

        M_transpose = M.t();

        // Fill vector y from Armadillo from loading textfile vectorY.txt
        for (int i = 0; i < N; ++i)
        {
                Y[i] = ptrvecy[i];
        }
        V = arma::inv(M_transpose * M)*M_transpose * Y ;

        M.print("M:");
        cout << endl;
        M_transpose.print("M^T:");
        cout << endl;
        Y.print("y:");
        cout << endl;

        // determinant and inverse from Armadillo
```

```
            cout << "det(M^T * M): " << det(M_transpose*M) << endl;
            cout << endl;
            cout << "inv(M^T * M): " << endl << arma::inv(M_transpose * M) <<
                endl;
            cout << "inv(M^T * M) * M^T: " << endl << arma::inv(M_transpose * M)
                *M_transpose << endl;
            cout << "v* = inv(M^T * M) * M^T * y : " << endl << arma::inv(
                M_transpose * M)*M_transpose * Y << endl;
            cout << "Least Square Cubic Polynomial Fit :" << endl;
            cout << ("y = ");
            printf("%.3f + ", V[0]);
            printf("%.3f x + ", V[1]);
            printf("%.3f x^2 + ", V[2]);
            printf("%.3f x^3 ", V[3]);
            cout <<endl;
        }
```

**C++ Code 251:** *main.cpp "Least Squares Cubic Polynomial Fit"*

To compile it, type:
**g++ -o result main.cpp -larmadillo**
**./result**

or with the Makefile, type:
**make**
**./main**

```
M:
   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00
   1.0000e+00   2.0000e+00   4.0000e+00   8.0000e+00
   1.0000e+00   3.0000e+00   9.0000e+00   2.7000e+01
   1.0000e+00   4.0000e+00   1.6000e+01   6.4000e+01
   1.0000e+00   5.0000e+00   2.5000e+01   1.2500e+02
   1.0000e+00   6.0000e+00   3.6000e+01   2.1600e+02
   1.0000e+00   7.0000e+00   4.9000e+01   3.4300e+02
   1.0000e+00   8.0000e+00   6.4000e+01   5.1200e+02
   1.0000e+00   9.0000e+00   8.1000e+01   7.2900e+02
   1.0000e+00   1.0000e+01   1.0000e+02   1.0000e+03
   1.0000e+00   1.1000e+01   1.2100e+02   1.3310e+03
   1.0000e+00   1.2000e+01   1.4400e+02   1.7280e+03
   1.0000e+00   1.3000e+01   1.6900e+02   2.1970e+03
   1.0000e+00   1.4000e+01   1.9600e+02   2.7440e+03
   1.0000e+00   1.5000e+01   2.2500e+02   3.3750e+03
   1.0000e+00   1.6000e+01   2.5600e+02   4.0960e+03

M^T:
   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00
 1.0000e+00   1.0000e+00   1.0000e+00
   1.0000e+00   2.0000e+00   3.0000e+00   4.0000e+00   5.0000e+00   6.0000e+00   7.0000e+00   8.0000e+00   9.0000e+00   1.0000e+01   1.1000e+01   1.2000e+01   1.3000e+01
 1.4000e+01   1.5000e+01   1.6000e+01
   1.0000e+00   4.0000e+00   9.0000e+00   1.6000e+01   2.5000e+01   3.6000e+01   4.9000e+01   6.4000e+01   8.1000e+01   1.0000e+02   1.2100e+02   1.4400e+02   1.6900e+02
 1.9600e+02   2.2500e+02   2.5600e+02
   1.0000e+00   8.0000e+00   2.7000e+01   6.4000e+01   1.2500e+02   2.1600e+02   3.4300e+02   5.1200e+02   7.2900e+02   1.0000e+03   1.3310e+03   1.7280e+03   2.1970e+03
 2.7440e+03   3.3750e+03   4.0960e+03

y:
   8.0000e+02
   1.0000e+02
   4.0000e+02
   7.0000e+02
   5.0000e+02
   3.0000e+03
   1.2000e+03
   2.5000e+03
   2.9000e+03
   2.2000e+03
   1.8000e+03
   1.8000e+03
   1.0000e+03
   1.5000e+03
   7.0000e+02
   4.0000e+02
```

**Figure 23.250:** *The computation for least squares fit of a cubic polynomial with C++ and Armadillo. (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Least Square Cubic Polynomial with Armadillo/main.cpp**).*

```
det(M^T * M): 2.8183e+12

inv(M^T * M):
   1.6621e+00  -7.4954e-01   9.0659e-02  -3.2051e-03
  -7.4954e-01   4.0562e-01  -5.3218e-02   1.9703e-03
   9.0659e-02  -5.3218e-02   7.3444e-03  -2.8115e-04
  -3.2051e-03   1.9703e-03  -2.8115e-04   1.1026e-05

inv(M^T * M) * M^T:
   1.0000e+00   5.0000e-01   1.4286e-01  -9.0659e-02  -2.1978e-01  -2.6374e-01  -2.4176e-01  -1.7308e-01  -7.6923e-02   2.7473e-02   1.2088e-01   1.8407e-01   1.9780e-01
 1.4286e-01   7.6217e-16  -2.5000e-01
  -3.9517e-01  -1.3541e-01   4.1562e-02   1.4756e-01   1.9441e-01   1.9393e-01   1.5794e-01   9.8267e-02   2.6733e-02  -4.4844e-02  -1.0464e-01  -1.4084e-01  -1.4161e-01
-9.5134e-02   1.0406e-02   1.8683e-01
   4.4505e-02   1.1352e-02  -1.0486e-02  -2.2695e-02  -2.6964e-02  -2.4978e-02  -1.8424e-02  -8.9902e-03   1.6373e-03   1.1772e-02   1.9726e-02   2.3812e-02   2.2345e-02
 1.3637e-02  -3.9990e-03  -3.2250e-02
  -1.5050e-03  -3.0100e-04   4.7300e-04   8.8315e-04   9.9561e-04   8.7653e-04   5.9207e-04   2.0838e-04  -2.0838e-04  -5.9207e-04  -8.7653e-04  -9.9561e-04  -8.8315e-04
-4.7300e-04   3.0100e-04   1.5050e-03

v* = inv(M^T * M) * M^T * y :
  -8.1868e+01
   2.3914e+02
   1.9774e+01
  -2.1245e+00

Least Square Cubic Polynomial Fit :
y = -81.868 + 239.141 x + 19.774 x^2 + -2.125 x^3
```

**Figure 23.251:** *The computation for least squares fit of a cubic polynomial with C++ and Armadillo. (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Least Square Cubic Polynomial with Armadillo/main.cpp**).*

## vi. C++ Plot and Computation: Least Squares Cubic Polynomial Fit

This is continuing from the previous section, we create the plot after we compute the least squares cubic polynomial that fit the data points.

$$y = a^* + b^*x + c^*x^2 + d^*x^3$$

the sixteen data points are:

$$(1, 800), \quad (2, 100), \quad (3, 400), \quad \ldots, \quad (16, 400)$$

we already know that the least square cubic polynomial fit is

$$y = -81.868 + 239.141x + 19.774x^2 - 2.125x^3$$

The computation stays the same, with **Armadillo**, but remember the placement of the computations that are calling **Armadillo** should be placed in the correct function. The plot will be using **OpenGL**, GLUT and GLEW.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <iostream>
#include <iomanip> // to declare the manipulator of setprecision()
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <vector>
#include <armadillo>

#include <GL/glew.h>
#include <GL/glut.h>

#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "../common/shader_utils.h"

#include "res_texture.c"

GLuint program;
GLint attribute_coord2d;
GLint uniform_offset_x;
GLint uniform_offset_y;
GLint uniform_scale_x;
GLint uniform_scale_y;
GLint uniform_sprite;
GLuint texture_id;
GLint uniform_mytexture;
```

```
float offset_x = 0.3;
float offset_y = 100.0;
float scale_x = 0.05; // first look: stretching out the x—axis when value
    is 0 < scale_x < 1
float scale_y = 0.0002; // first look: stretching out the y—axis when
    value is 0 < scale_y < 1
int mode = 0;

struct point {
       GLfloat x;
       GLfloat y;
};

const int N = 16;
const int C1 = 4;
const int C2 = 16;

GLuint vbo;

using namespace std;
using namespace arma;

float* vecx() {
       static float x[N];
       std::fstream in("vectorX.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
              in >> vectortiles[i];
              x[i] =vectortiles[i];
       }
       return x;
}

float* vecy() {
       static float y[N];
       std::fstream in("vectorY.txt");
       float vectortiles[N];
       for (int i = 0; i < N; ++i)
       {
              in >> vectortiles[i];
              y[i] =vectortiles[i];
       }
       return y;
}

int init_resources() {
```

```
program = create_program("graph.v.glsl", "graph.f.glsl");
if (program == 0)
return 0;

attribute_coord2d = get_attrib(program, "coord2d");
uniform_offset_x = get_uniform(program, "offset_x");
uniform_offset_y = get_uniform(program, "offset_y");
uniform_scale_x = get_uniform(program, "scale_x");
uniform_scale_y = get_uniform(program, "scale_y");
uniform_sprite = get_uniform(program, "sprite");
uniform_mytexture = get_uniform(program, "mytexture");

if (attribute_coord2d == -1 || uniform_offset_y == -1 ||
    uniform_offset_x == -1 || uniform_scale_y == -10 ||
    uniform_scale_x == -1 || uniform_sprite == -1 ||
    uniform_mytexture == -1)
return 0;

/* Enable blending */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

/* Enable point sprites (not necessary for true OpenGL ES 2.0) */
#ifndef GL_ES_VERSION_2_0
glEnable(GL_POINT_SPRITE);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
#endif

/* Upload the texture for our point sprites */
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, &texture_id);
glBindTexture(GL_TEXTURE_2D, texture_id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, res_texture.width,
    res_texture.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, res_texture.
    pixel_data);

// Create the vertex buffer object
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Take the vector data from textfile as pointer
float* ptrvecx;
ptrvecx = vecx();
float* ptrvecy;
ptrvecy = vecy();
```

```
        point graph[N];
        for (int i = 1; i < N; i++)
        {
                graph[i].x = ptrvecx[i];
                graph[i].y = ptrvecy[i];
        }

        // Tell OpenGL to copy our array to the buffer object
        glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph, GL_STATIC_DRAW);

        return 1;
}

void display() {
        glUseProgram(program);
        glUniform1i(uniform_mytexture, 0);

        glUniform1f(uniform_offset_x, offset_x);
        glUniform1f(uniform_offset_y, offset_y);
        glUniform1f(uniform_scale_x, scale_x);
        glUniform1f(uniform_scale_y, scale_y);

        glClearColor(0.0, 0.0, 0.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);

        /* Draw using the vertices in our vertex buffer object */
        glBindBuffer(GL_ARRAY_BUFFER, vbo);

        glEnableVertexAttribArray(attribute_coord2d);
        glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0,
            0);

        float* ptrvecx;
        ptrvecx = vecx();
        float* ptrvecy;
        ptrvecy = vecy();

        arma::mat M(N,C1,fill::zeros);
        arma::mat M_transpose(C1,C2,fill::zeros);
        arma::mat Y(N,1,fill::zeros);
        arma::mat V(N,1,fill::zeros);

        for (int i = 0; i < N; ++i)
        {
                for(int j = 0; j<C1; ++j)
                {
                        M[i+j*N] = pow(ptrvecx[i],j);
                }
```

```
            }

            M_transpose = M.t();

            for (int i = 0; i < N; ++i)
            {
                  Y[i] = ptrvecy[i];
            }
            V = arma::inv(M_transpose * M)*M_transpose * Y ;
            /* Push each element in buffer_vertices to the vertex shader */
            switch (mode) {
                  case 0:
                  glClearColor(0.0, 0.0, 0.0, 0.0);
                  glClear(GL_COLOR_BUFFER_BIT);
                  glGenBuffers(1, &vbo);
                  glBindBuffer(GL_ARRAY_BUFFER, vbo);
                  glEnableVertexAttribArray(attribute_coord2d);
                  glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
                      GL_FALSE, 0, 0);

                  point graph[N];
                  for (int i = 1; i < N; i++)
                  {
                        graph[i].x = ptrvecx[i];
                        graph[i].y = ptrvecy[i];
                  }

                  glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph,
                      GL_STATIC_DRAW);
                  glUniform1f(uniform_sprite, res_texture.width);
                  glDrawArrays(GL_POINTS, 0, 2000);
                  break;
                  case 1:
                  glUniform1f(uniform_sprite, 1);
                  glDrawArrays(GL_POINTS, 0, 2000);
                  break;
                  case 2:
                  glUniform1f(uniform_sprite, res_texture.width);
                  glDrawArrays(GL_POINTS, 0, 2000);
                  break;
                  case 3:
                  glClearColor(0.0, 0.0, 0.0, 0.0);
                  glClear(GL_COLOR_BUFFER_BIT);

                  glGenBuffers(1, &vbo);
                  glBindBuffer(GL_ARRAY_BUFFER, vbo);
                  glEnableVertexAttribArray(attribute_coord2d);
                  glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
```

```
                    GL_FALSE, 0, 0);

                point graphcubicfit[2000];
                for (int i = 0; i < 2000; i++)
                {
                        float x = (i -1000) / 10.0;
                        graphcubicfit[i].x = x;
                        graphcubicfit[i].y = V[0] + V[1]*x + V[2]*x*x + V[3]*x
                            *x*x;
                }

                glBufferData(GL_ARRAY_BUFFER, sizeof graphcubicfit,
                    graphcubicfit, GL_STATIC_DRAW);
                glUniform1f(uniform_sprite, 0);
                glDrawArrays(GL_LINE_STRIP, 0, 2000);
                break;
        }

        glutSwapBuffers();
}

void special(int key, int x, int y) {
        switch (key) {
                case GLUT_KEY_F1:
                mode = 0;
                printf("Now drawing the data points.\n");
                break;
                case GLUT_KEY_F2:
                mode = 1;
                printf("Now drawing using points.\n");
                break;
                case GLUT_KEY_F3:
                mode = 2;
                printf("Now drawing using point sprites.\n");
                break;
                case GLUT_KEY_F4:
                mode = 3;
                printf("Now drawing the cubic polynomial fit.\n");
                break;
                case GLUT_KEY_LEFT:
                offset_x += 2;
                break;
                case GLUT_KEY_RIGHT:
                offset_x -= 2;
                break;
                case GLUT_KEY_UP:
                offset_y -= 30;
                break;
```

```
                case GLUT_KEY_DOWN:
                offset_y += 30;
                break;
                case GLUT_KEY_F5:
                scale_x += 0.05;
                break;
                case GLUT_KEY_F6:
                scale_x -= 0.05;
                break;
                case GLUT_KEY_F7:
                scale_y += 0.0001;
                break;
                case GLUT_KEY_F8:
                scale_y -= 0.0001;
                break;
                case GLUT_KEY_HOME:
                offset_x = 3.0;
                offset_y = 100.0;
                scale_x = 0.05; // stretching the x-axis
                scale_y = 0.0002; // stretching the y-axis
                break;
        }

        glutPostRedisplay();
}

void free_resources() {
        glDeleteProgram(program);
}

int main(int argc, char *argv[]) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB);
        glutInitWindowSize(1024, 600);
        glutCreateWindow("Least Squares Cubic Polynomial Fit");

        GLenum glew_status = glewInit();

        if (GLEW_OK != glew_status) {
                fprintf(stderr, "Error: %s\n", glewGetErrorString(glew_status
                    ));
                return 1;
        }

        if (!GLEW_VERSION_2_0) {
                fprintf(stderr, "No support for OpenGL 2.0 found\n");
                return 1;
        }
```

```
float* ptrvecx;
ptrvecx = vecx();
float* ptrvecy;
ptrvecy = vecy();

cout << "Vector / Data X: " << setw(30) << "Vector / Data Y: " <<
    endl;
for (int i = 0; i < N; ++i)
{
       cout <<ptrvecx[i]<< "\t \t \t \t" << ptrvecy[i] << endl;
}
cout << endl;

arma::mat M(N,C1,fill::zeros);
arma::mat M_transpose(C1,C2,fill::zeros);
arma::mat Y(N,1,fill::zeros);
arma::mat V(N,1,fill::zeros);

for (int i = 0; i < N; ++i)
{
       for(int j = 0; j<C1; ++j)
       {
              M[i+j*N] = pow(ptrvecx[i],j);
       }
}

M_transpose = M.t();

for (int i = 0; i < N; ++i)
{
       Y[i] = ptrvecy[i];
}
V = arma::inv(M_transpose * M)*M_transpose * Y ;

M.print("M:");
cout << endl;
M_transpose.print("M^T:");
cout << endl;
Y.print("y:");
cout << endl;

// determinant and inverse from Armadillo
cout << "det(M^T * M): " << det(M_transpose*M) << endl;
cout << endl;
cout << "inv(M^T * M): " << endl << arma::inv(M_transpose * M) <<
    endl;
cout << "inv(M^T * M) * M^T: " << endl << arma::inv(M_transpose * M)
```

```
                *M_transpose << endl;
            cout << "v* = inv(M^T * M) * M^T * y : " << endl << arma::inv(
                M_transpose * M)*M_transpose * Y << endl;
            cout << "Least Squares Cubic Polynomial Fit :" << endl;
            cout << ("y = ");
            printf("%.3f + ", V[0]);
            printf("%.3f x + ", V[1]);
            printf("%.3f x^2 + ", V[2]);
            printf("%.3f x^3 ", V[3]);
            cout <<endl;

            GLfloat range[2];

            glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, range);
            if (range[1] < res_texture.width)
            fprintf(stderr, "WARNING: point sprite range (%f, %f) too small\n",
                range[0], range[1]);

            printf("Use left/right to move horizontally.\n");
            printf("Use up/down to move vertically.\n");
            printf("Press home to reset the position and scale.\n");
            printf("Press F1 to draw data points.\n");
            printf("Press F2 to draw points.\n");
            printf("Press F3 to draw point sprites.\n");
            printf("Press F4 to draw the least squares cubic polynomial fit.\n")
                ;
            printf("Press F5 to zoom in / F6 to zoom out the horizontal scale.\n
                ");
            printf("Press F7 to zoom in / F8 to zoom out the vertical scale.\n")
                ;

            if (init_resources()) {
                   glutDisplayFunc(display);
                   glutSpecialFunc(special);
                   glutMainLoop();
            }

            free_resources();
            return 0;
      }
```

**C++ Code 252:** *main.cpp "Plot for Least Squares Cubic Polynomial Fit"*

To compile it with the Makefile, type:
**make**
**./graph**

**Figure 23.252:** *The plot for the data points of ADHI stocks price from 2008-2023 with C++ and Armadillo. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Least Square Cubic Polynomial Fit/main.cpp**).*

## XX. Function Approximation with Fourier Series

[DF*] Orthogonal projection can be used to approximate certain types of functions by simpler functions that are easier to work with.

[DF*] The approximation problem:
Given a function $f$ that is continuous on an interval $[a, b]$, find the "best possible approximation" to $f$ using only functions from a specified subspace $W$ of $C[a, b]$. Some examples:

- Find the best possible approximation to $e^x$ over $[0, 1]$ by a polynomial of the form $a_0 + a_1 x + a_2 x^2$.

  $W$ is the subspace of $C[0, 1]$ spanned by $1, x, x^2$.
- Find the best possible approximation to $\sin\ \pi x$ over $[-1, 1]$ by a function of the form $a_0 + a_1 e^x + a_2 e^{2x} + a_3 e^{3x}$.

  $W$ is the subspace of $C[-1, 1]$ spanned by $1, e^x, e^{2x}, e^{3x}$.
- Find the best possible approximation to $x$ over $[0, 2\pi]$ by a function of the form $a_0 + a_1 \sin\ x + a_2 \sin\ 2x + b_1 \cos\ x + b_2 \cos\ 2x$.

  $W$ is the subspace of $C[0, 2\pi]$ spanned by $1, \sin\ x, \sin\ 2x, \cos\ x, \cos\ 2x$.

[DF*] If we were to approximate $f(x)$ by $g(x)$ at a single point $x_0$, then it would be natural to define the error to be

$$\text{error} = |f(x_0) - g(x_0)|$$

**Figure 23.253:** *The plot for the least squares fit of a cubic polynomial of ADHI stocks price from 2008-2023 with C++ and Armadillo.* (***DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Least Square Cubic Polynomial Fit/main.cpp***).

**Figure 23.254:** *This is the ADHI stocks price' graph taken from Google on March 2024, it seems our approximation with least square cubic polynomial is too rough and we should not believe the trend of the stocks price will follow our least squares cubic polynomial fit line.*

sometimes called the deviation between $f$ and $g$ at $x_0$.

We want to measure the error over the entire interval $[a, b]$, but the problem is the approximation may have small deviations in one part of the interval and large deviations in another. One possible way of accounting for this is to integrate the deviation $|f(x) - g(x)|$ over the interval $[a, b]$ and define the error over the interval to be

$$\text{error} = \int_a^b |f(x_0) - g(x_0)| \, dx \tag{23.299}$$

Geometrically, the error is the area between the graph $f(x)$ and $g(x)$ over the interval $[a, b]$; the greater the area, the greater the overall error.

[DF*] Most mathematicians and scientists generally favor the following alternative measure of error, called the mean square error:

$$\text{mean square error} = \int_a^b [f(x_0) - g(x_0)]^2 \, dx \tag{23.300}$$

Mean square error emphasizes the effect of larger errors because of the squaring and has the added advantage that it allows us to bring to bear the theory of inner product spaces.

Suppose that $\boldsymbol{f}$ is a continuous function on $[a, b]$ that we want to approximate by a function $\boldsymbol{g}$ from a subspace $W$ of $C[a, b]$, and suppose that $C[a, b]$ is given the inner product

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_a^b f(x)g(x) \, dx$$

it follows that

$$||\boldsymbol{f} - \boldsymbol{g}||^2 =< \boldsymbol{f} - \boldsymbol{g}, \boldsymbol{f} - \boldsymbol{g} >= \int_a^b [f(x) - g(x)]^2 \, dx = \text{mean square error}$$

so minimizing the mean square error is the same as minimizing $||\boldsymbol{f} - \boldsymbol{g}||^2$.

**Definition 23.70: Least Squares Approximation Problem**

Let $\boldsymbol{f}$ be a function that is continuous on an interval $[a, b]$, let $C[a, b]$ have the inner product

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_a^b f(x)g(x) \, dx$$

and let $W$ be a finite-dimensional subspace of $C[a, b]$. Find a function $\boldsymbol{g}$ in $W$ that minimizes

$$||\boldsymbol{f} - \boldsymbol{g}||^2 = \int_a^b [f(x) - g(x)]^2 \, dx$$

[DF*] $||\boldsymbol{f} - \boldsymbol{g}||^2$ and $||\boldsymbol{f} - \boldsymbol{g}||$ are minimized by the same function $\boldsymbol{g}$, this problem is equivalent to looking for a function $\boldsymbol{g}$ in $W$ that is closest to $\boldsymbol{f}$. Such function is

$$\boldsymbol{g} = \text{proj}_W \boldsymbol{f}$$

we can say that $\boldsymbol{g}$ is the least squares approximation to $\boldsymbol{f}$ from $W$, and $W$ is the subspace of approximating functions.

**Theorem 23.159: Least Squares Approximation**

If $f$ is a continuous function on $[a,b]$, and $W$ is a finite-dimensional subspace of $C[a,b]$, then the function $g$ in $W$ that minimizes the mean square error

$$\int_a^b [f(x)-g(x)]^w \, dx$$

is $g = \text{proj}_W f$, where the orthogonal projection is relative to the inner product

$$< f,g >= \int_a^b f(x)g(x) \, dx$$

The function $g = \text{proj}_W f$ is called the least squares approximation to $f$ from $W$.

[DF*] A function of the form

$$T(x) = c_0 + c_1 \cos x + c_2 \cos 2x + \cdots + c_n \cos nx + d_1 \sin x + d_2 \sin 2x + \cdots + d_n \sin nx \tag{23.301}$$

is called a trigonometric polynomial; if $c_n$ and $d_n$ are not both zero, then $T(x)$ is said to have order $n$. For example:

$$T(x) = 2 + \cos x - 3 \cos 2x + 7 \sin 4x$$

is a trigonometric polynomial of order 4 with

$$\begin{aligned} c_0 &= 2 \\ c_1 &= 1 \\ c_2 &= -3 \\ c_3 &= 0 \\ c_4 &= 0 \\ d_1 &= 0 \\ d_2 &= 0 \\ d_3 &= 0 \\ d_4 &= 7 \end{aligned}$$

the trigonometric polynomials of order $n$ or less are the various possible linear combinations of

$$1, \cos x, \cos 2x, \ldots, \cos nx, \sin x, \sin 2x, \ldots, \sin nx \tag{23.302}$$

it can be shown that these $2n+1$ functions are linearly independent and thus form a basis for a $(2n+1)$-dimensional subspace of $C[a,b]$.

[DF*] Let us now consider the problem of finding the least squares approximation of a continuous function $f(x)$ over the interval $[0, 2\pi]$ by a trigonometric polynomial of order $n$ or less.

The least squares approximation to $f$ from $W$ is the orthogonal projection of $f$ on $W$.

To find this orthogonal projection, we must find an orthonormal basis $g_0, g_1, \ldots, g_{2n}$ for $W$, after which we can compute the orthogonal projection on $W$ from the formula

$$\text{proj}_W f =< f, g_0 > g_0 + < f, g_1 > g_1 + \cdots + < f, g_{2n} > g_{2n} \tag{23.303}$$

An orthonormal basis for $W$ can be obtained by applying the Gram-Schmidt process to the basis vectors

$$\{1,\ \cos\ x, \cos\ 2x, \ldots, \cos\ nx, \sin\ x, \sin\ 2x, \ldots, \sin\ nx\}$$

using the inner product

$$< \boldsymbol{f}, \boldsymbol{g} >= \int_0^{2\pi} f(x)g(x)\ dx$$

This yields the orthonormal basis

$$\begin{aligned}
\boldsymbol{g}_0 &= \frac{1}{\sqrt{2\pi}} \\
\boldsymbol{g}_1 &= \frac{1}{\sqrt{\pi}} \cos\ x \\
&\vdots \\
\boldsymbol{g}_n &= \frac{1}{\sqrt{\pi}} \cos\ nx \\
\boldsymbol{g}_{n+1} &= \frac{1}{\sqrt{\pi}} \sin\ x \\
&\vdots \\
\boldsymbol{g}_{2n} &= \frac{1}{\sqrt{\pi}} \sin\ nx
\end{aligned} \tag{23.304}$$

If we introduce the notation

$$\begin{aligned}
a_0 &= \frac{2}{\sqrt{2\pi}} < \boldsymbol{f}, \boldsymbol{g}_0 > \\
a_1 &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_1 > \\
&\vdots \\
a_n &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_n > \\
b_1 &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_{n+1} > \\
&\vdots \\
b_n &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_{2n} >
\end{aligned} \tag{23.305}$$

then on substituting we obtain

$$\text{proj}_W \boldsymbol{f} = \frac{a_0}{2} + [a_1 \cos\ x + \cdots + a_n \cos\ nx] + [b_1 \sin\ x + \cdots + b_n \sin\ nx] \tag{23.306}$$

or in summation form

$$\text{proj}_W \boldsymbol{f} = \frac{a_0}{2} + \sum_{i=1}^{n} [a_i \cos\ ix + b_i \sin\ ix] \tag{23.307}$$

where

$$
\begin{aligned}
a_0 &= \frac{2}{\sqrt{2\pi}} < \boldsymbol{f}, \boldsymbol{g}_0 >= \frac{2}{\sqrt{2\pi}} \int_0^{2\pi} f(x) \frac{1}{\sqrt{2\pi}}\, dx = \frac{1}{\pi} \int_0^{2\pi} f(x)\, dx \\
a_1 &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_1 >= \frac{1}{\sqrt{\pi}} \int_0^{2\pi} f(x) \frac{1}{\sqrt{\pi}} \cos\ x\, dx = \frac{1}{\pi} \int_0^{2\pi} f(x)\ \cos\ x\, dx \\
&\vdots \\
a_n &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_n >= \frac{1}{\sqrt{\pi}} \int_0^{2\pi} f(x) \frac{1}{\sqrt{\pi}} \cos\ nx\, dx = \frac{1}{\pi} \int_0^{2\pi} f(x)\ \cos\ nx\, dx \\
b_1 &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_{n+1} >= \frac{1}{\sqrt{\pi}} \int_0^{2\pi} f(x) \frac{1}{\sqrt{\pi}} \sin\ x\, dx = \frac{1}{\pi} \int_0^{2\pi} f(x)\ \sin\ x\, dx \\
&\vdots \\
b_n &= \frac{1}{\sqrt{\pi}} < \boldsymbol{f}, \boldsymbol{g}_{2n} >= \frac{1}{\sqrt{\pi}} \int_0^{2\pi} f(x) \frac{1}{\sqrt{\pi}} \sin\ nx\, dx = \frac{1}{\pi} \int_0^{2\pi} f(x)\ \sin\ nx\, dx
\end{aligned}
$$

In short,

$$
\begin{aligned}
a_k &= \frac{1}{\pi} \int_0^{2\pi} f(x)\ \cos\ kx\, dx \\
b_k &= \frac{1}{\pi} \int_0^{2\pi} f(x)\ \sin\ kx\, dx
\end{aligned}
\tag{23.308}
$$

The numbers $a_0, a_1, \ldots, a_n, b_1, \ldots, b_n$ are called the Fourier coefficients of $\boldsymbol{f}$.

[DF*] It is natural to expect that the mean square error will diminish as the number of terms in the least squares approximation

$$
f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{n} (a_k\ \cos\ kx + b_k\ \sin\ kx)
$$

increases. It can be proved that for functions $f$ in $\mathbb{C}[0, 2\pi]$, the mean square error approaches to zero as $n \to +\infty$, this is denoted by writing

$$
f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k\ \cos\ kx + b_k\ \sin\ kx)
$$

The right side of this equation is called the Fourier series for $f$ over the interval $[0, 2\pi]$. Such series are of major importance in engineering, science, and mathematics.

**Computational Guide 23.8: Fourier Series**

You can solve many important problems involving partial differential equations if you can express a given function as an infinite sum of sines and/or cosines [1]. These sum of trigonometric seriers are called Fourier series. There are three Fourier series actually: sines, cosines, and complex exponentials $e^{ikx}$ (which is also a trigonometric series $\cos kx + i \sin kx$).

The general trigonometric series has a form of:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}(a_n \cos\ nx + b_n \sin\ nx) \tag{23.309}$$

the series above is called the Fourier series of $f(x)$, with $a_n$ and $b_n$ are called the Fourier coefficients of the function $f(x)$. The advantage of the equation above is that it can represent very general functions with many dicontinuities, wheras power series can represent only continuous functions that have derivatives of all orders.

In order to form Fourier series, it is not necessary to assume that $f(x)$ is continuous, but only that the integrals

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\ \cos\ nx\ dx$$
$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\ \sin\ nx\ dx$$

exist. Thus, we need to assume that $f(x)$ is integrable on the interval $-\pi \leq x \leq \pi$.

There exist many integrable, even continuous, functions whose Fourier series diverge at one or more points.

It was said from [14] that Fourier studied the infinite combinations of the sines

$$S(x) = b_1 \sin\ x + b_2 \sin\ 2x + b_3 \sin\ 3x + \cdots = \sum_{n=1}^{\infty} b_n \sin\ nx \tag{23.310}$$

Fourier believed the sum $S(x)$ will inherit these three properties:

$$S(x+2\pi) = S(x) \quad \textbf{Periodic}$$
$$S(-x) = -S(x) \quad \textbf{Odd}$$
$$S(0) = S(\pi) = 0 \quad \textbf{Vanishes at } x = 0, \pi$$

**Computational Guide 23.9: Fourier Series**

Three types of Fourier series that are most common are:

1. **Square Wave**

$$f(x) = \sum_{n=1}^{\infty} \frac{\sin\,(2n-1)x}{2n-1} \tag{23.311}$$

   it means that if we add infinite sine waves with formula like above we would have a square wave.

2. **Sawtooth** The Fourier series that will make a sawtooth shape are

$$f(x) = \sum_{n=1}^{\infty} c\left(\frac{\sin\ nx}{n}\right) \tag{23.312}$$

$$f(x) = \sum_{n=1}^{\infty} c\left(\frac{\sin\,(2n-1)x}{2n} - \frac{\sin\,(2n)x}{(2n)}\right) \tag{23.313}$$

$$f(x) = c\sum_{n=1}^{\infty}(-1)^{n+1}\sin\,(nx) \tag{23.314}$$

$$f(x) = c\sum_{n=1}^{\infty}(-1)^{n+1}\sin\left(\frac{nx}{n}\right) \tag{23.315}$$

$$f(x) = \pi - c\sum_{n=1}^{\infty}(-1)^{n+1}\sin\left(\frac{nx}{n}\right) \tag{23.316}$$

   with $c$ as scalar.

3. **Triangle Wave** The Fourier series that will make a triangular wave is

$$f(x) = \frac{\pi}{2} - \frac{4}{\pi}\sum_{n=1}^{\infty}\left(\frac{\cos\,(2n-1)x}{(2n-1)^2}\right) \tag{23.317}$$

   the function above is approximating $f(x) = |x|$ at interval $-\pi \leq x < \pi$.

There are at least 20 Fourier Series that are known, including sinh series and those with both sin and cos, these 3 are the basic and most common for starter, reader who is interested should read books dedicated for advanced Fourier series.

**Figure 23.255:** *The Fourier series that approximate the function $f(x) = |x|$ on $[-\pi, \pi)$, with order from $n = 1$ to $n = 4$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Fourier Series with Known f(x) Triangular Waves with a0 (Full Moon Cold Moon)/main.cpp**).*

## i. C++ Plot and Computation: Least Squares Approximations with Fourier Series

Find the least squares approximation of $f(x) = x$ on $[0, 2\pi]$ by

(a) a trigonometric polynomial of order 2 or less;

(b) a trigonometric polynomial of order $n$ or less.

**Solution:**

(a) First we will compute $a_0$

$$a_0 = \frac{1}{\pi}\int_0^{2\pi} f(x)\ dx = \frac{1}{\pi}\int_0^{2\pi} x\ dx = 2\pi$$

for $k = 1, 2, \ldots,$ integration by part yields

$$a_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\ \cos\ x\ dx = \frac{1}{\pi}\int_0^{2\pi} x\ \cos\ x\ dx = 0$$
$$b_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\ \sin\ x\ dx = \frac{1}{\pi}\int_0^{2\pi} x\ \sin\ x\ dx = -\frac{2}{k}$$

Thus, the least squares approximation to $x$ on $[0, 2\pi]$ by a trigonometric polynomial of order 2 or less is

$$x \approx \frac{a_0}{2} + a_1\ \cos\ x + a_2\ \cos\ 2x + b_1\ \sin\ x + b_2\ \sin\ 2x$$
$$x \approx \pi - 2\ \sin\ x - \sin\ 2x$$

(b) Now we can substitute the Fourier coefficients into the Fourier series

$$\text{proj}_W \boldsymbol{f} = \frac{a_0}{2} + \sum_{k=1}^{n}[a_k \cos\ kx + b_k \sin\ kx]$$

$$\begin{aligned}\text{proj}_W \boldsymbol{f} &= \frac{a_0}{2} + \sum_{k=1}^{n}[a_k \cos\ kx + b_k \sin\ kx] \\ &= \frac{2\pi}{2} + \sum_{k=1}^{n}\left[0 \cos\ kx + \frac{-2}{k}\sin\ kx\right]\end{aligned}$$

so the least squares approximation to $x$ on $[0, 2\pi]$ by a trigonometric polynomial of order $n$ or less is

$$x \approx \frac{a_0}{2} + a_1\ \cos\ x + \cdots + a_n\ \cos\ nx + b_1\ \sin\ x + \cdots + b_n\ \sin\ nx$$
$$x \approx \pi - 2\left(\sin\ x + \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3} + \cdots + \frac{\sin\ nx}{n}\right)$$

$a_k = 0$ so there will be no term with $\cos\ kx$ in the Fourier series.

The hard part for the C++ code is to do integrating by parts computation, we opt to compute the integral by hand for now, but we will come back again if there is a decent C++ library that is able to perform hard symbolic integration like **SymPy**. After we do the integration, we will obtain the Fourier coefficients $a_0, a_k, b_k$ that we can use to form the Fourier series.

We can only do the sum computation with **SymbolicC++**, that is, we need to compute manually by hand the Fourier coefficients $a_0, a_k$, and $b_k$.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include "gnuplot-iostream.h"
#include "symbolicc++.h"

using namespace std;

const double PI = 3.14159265358979;

double division(double x, double y)
{
       return x/y;
}

int main()
{
       Symbolic x("x"), y, n("n"), pi("pi");
       const double period = 1.0; // Period of function
       const int N = 10; // Order for the trigonometric polynomial

       y = pi;

       //freopen ("fourierseries.txt","w",stdout); // You can redirect
           stdout, stderr and stdin using std::freopen
       // Fourier series of the function f(x)=x, 0 <= x <= 2pi
       // x = sum_{1}^{N} (-2/n) * sin(nx)
       for ( int n = 1; n <= N; n++ )
       {
              y += division(-2,n)*sin( n*x );
              cout << n <<" " << y << endl;
       }

       return 0;
}
```

**C++ Code 253:** *main.cpp "Least squares approximations with fourier series"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -lsymbolicc++**
**./main**

or with Makefile, type:
**make**

**./main**

```
1 π-2*sin(x)
2 π-2*sin(x)-sin(2*x)
3 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)
4 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)
5 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)
6 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)-0.333333*sin(6*x)
7 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)-0.333333*sin(6*x)-0.285714*sin(7*x)
8 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)-0.333333*sin(6*x)-0.285714*sin(7*x)-0.25*sin(8*x)
9 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)-0.333333*sin(6*x)-0.285714*sin(7*x)-0.25*sin(8*x)-0.222222*sin(9*x)
10 π-2*sin(x)-sin(2*x)-0.666667*sin(3*x)-0.5*sin(4*x)-0.4*sin(5*x)-0.333333*sin(6*x)-0.285714*sin(7*x)-0.25*sin(8*x)-0.222222*sin(9*x)-0.2*sin(10*x)
```

**Figure 23.256:** *The sum computation to obtain the Fourier series that approximate the function $f(x) = x$ on $[0, 2\pi]$, with order from $n = 1$ to $n = 10$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Fourier Series with Known f(x) Example 1/main.cpp**).*

The second C++ code is to do the plotting with **gnuplot**, we just need to manually input the Fourier series function that we have computed of order $1, 2, 3, 4$.

```
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot—iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [0:12]\nset yrange [0:8]\n";
        gp << "set xlabel 'x—axis'\n set ylabel 'y—axis'\n";
        gp <<"set xtics nomirror (0, '{/:Italic pi/2}' 0.5*pi, '{/:Italic pi
            }' pi, '{/:Italic 3pi/2}' 1.5*pi,\
        '2{/:Italic pi}' 2*pi, '{/:Italic 5pi/2}' 2.5*pi, '3{/:Italic pi}'
            3*pi, '{/:Italic 7pi/2}' 3.5*pi,\
        '4{/:Italic pi}' 4*pi, '5{/:Italic pi}' 5*pi, '6{/:Italic pi}' 6*pi)
            \n";
        // '—' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f1(x) = x\n";
        gp << "f(x) = 3.14159 — 2*sin(x)\n";
        gp << "g(x) = 3.14159 — 2*sin(x) — sin(2*x)\n";
        gp << "h(x) = 3.14159 — 2*sin(x)—sin(2*x) — 0.6666*sin(3*x)\n";
        gp << "i(x) = 3.14159 — 2*sin(x)—sin(2*x) — 0.6666*sin(3*x) —
            0.5*sin(4*x)\n";
        gp << "plot f1(x) title 'y = x' lw 2 lc rgb 'black',\
        f(x) title 'y = pi — 2 sin x' lw 3 lc rgb 'forest—green', \
        g(x) title 'y = pi — (2 sin x + sin 2x)' lw 3 lc rgb 'spring—green
```

```
                ', \
            h(x) title 'y = pi — (2 sin x + sin 2x + (2/3) sin 3x)' lw 3 lc rgb
                 'orchid4',\
            i(x) title 'y = pi — (2 sin x + sin 2x + (2/3) sin 3x + (2/4) sin 4
                x)' lw 3 lc rgb 'light—salmon'\n";

            return 0;
        }
```

**C++ Code 254:** *main.cpp "Plot of least squares approximations with fourier series"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.257:** *The plot of Fourier series that approximate the function $f(x) = x$ on $[0, 2\pi]$, with order from $n = 1$ to $n = 4$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Fourier Series with Known f(x) Sawtooth with a0/main.cpp**).*

## ii. C++ Plot and Computation: Find the Fourier Series of function $f(x)$

Find the Fourier series of $f(x) = x$ on $[-\pi, \pi]$.

**Solution:**

First, we will compute all Fourier coefficients, we will have

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} x\ dx = \frac{1}{\pi}\frac{x^2}{2}|_{-\pi}^{\pi} = 0$$

If $n \geq 1$, then we need to find $a_n$, with integrating by parts we set $u = x, dv = \cos\ nx\ dx$,

$$\begin{aligned} a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} x\ \cos\ nx\ dx \\ &= \frac{1}{\pi}\left[\frac{x\ \sin\ nx}{n} + \frac{\cos\ nx}{n^2}\right]_{-\pi}^{\pi} \\ &= 0 \end{aligned}$$

after that, we need to find $b_n$, with integrating by parts we set $u = x, dv = \sin\ nx\ dx$,

$$\begin{aligned} b_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} x\ \sin\ nx\ dx \\ &= \frac{1}{\pi}\left[-\frac{x\ \cos\ nx}{n} + \frac{\sin\ nx}{n^2}\right]_{-\pi}^{\pi} \\ &= \frac{1}{\pi}\left[-\frac{\pi\ \cos\ n\pi}{n} + \frac{\pi\cos\ (-n\pi)}{n}\right] \\ &= \frac{1}{\pi}\left[-\frac{\pi\ \cos\ n\pi}{n} - \frac{\pi\cos\ (n\pi)}{n}\right] \\ &= \frac{1}{\pi}\left[-\frac{2\pi\ \cos\ n\pi}{n}\right] \\ &= -\frac{2}{n}\cos\ n\pi \\ &= \frac{2}{n}(-1)^{n+1} \end{aligned}$$

since $\cos\ n\pi = (-1)^n$. Now substituting the Fourier coefficients into Fourier series we will obtain

$$\begin{aligned} x &\approx \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n\ \cos\ nx + b_n\ \sin\ nx \\ x &\approx \sum_{n=1}^{\infty} b_n\ \sin\ nx \\ x &\approx \sum_{n=1}^{\infty} \frac{2}{n}(-1)^{n+1}\ \sin\ nx \\ x &\approx 2(-1)^2\ \sin\ x + \left(\frac{2}{2}(-1)^3\ \sin\ 2x\right) + \left(\frac{2}{3}(-1)^4\ \sin\ 3x\right) + \left(\frac{2}{4}(-1)^5\ \sin\ 4x\right) + \dots \\ x &\approx 2\left(\sin\ x - \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3} - \frac{\sin\ 4x}{4} + \cdots - \dots\right) \end{aligned}$$

This means that we hope that the series

$$2\left(\sin\ x - \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3} - \frac{\sin\ 4x}{4} + \cdots - \ldots\right)$$

covnerges to $x$ for $-\pi \leq x \leq \pi$. Remember that $\sin\ x$ has periods $2\pi, 4\pi, 6\pi, \ldots$, and $\sin\ 2x$ has periods $\pi, 2\pi, 3\pi, \ldots$.

The first C++ code is using**SymbolicC++** to do symbolic sum computation, there is no integration done here, we can perform this computation after we compute the integration and determine $a_0, a_n$, and $b_n$, then we approximate the function $f(x) = x, -\pi \leq x, \leq \pi$ with Fourier series with order that can be chosen ourselves.
(on July 26th, 2024 we do not currently have a nice alternative to do symbolic integrating by parts for $\int_{-\pi}^{\pi} x \cos\ nx\ dx$ or $\int_{-\pi}^{\pi} x\ \sin\ nx\ \int_{-\pi}^{\pi}$, we do not want to use **SymPy** since it is purely not C++, so we leave the integration to be done by hand by reader, but if you are curious the Python' package **SymPy** will be able to compute the symbolic integration mentioned here, easily.)

For example the Fourier approximation for the function $f(x) = x, -\pi \leq x, \leq \pi$ with order of $n = 1$ is

$$x \approx 2 \sin\ x$$

the Fourier approximation for the function $f(x) = x, -\pi \leq x, \leq \pi$ with order of $n = 3$ is

$$x \approx 2\left(\sin\ x - \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3}\right)$$

(can also be called the sum of the first three terms of the Fourier series)

the Fourier approximation for the function $f(x) = x, -\pi \leq x, \leq \pi$ with order of $n = 5$ is

$$x \approx 2\left(\sin\ x - \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3} - \frac{\sin\ 4x}{4} + \frac{\sin\ 5x}{5}\right)$$

(can also be called the sum of the first five terms of the Fourier series)

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <functional>
#include <algorithm>
#include <cmath>
#include "gnuplot-iostream.h"
#include "symbolicc++.h"

using namespace std;

const double PI = 3.14159265358979;

double division(double x, double y)
```

```
{
        return x/y;
}

int main()
{
        Symbolic x("x"), y, n("n");
        const double period = 1.0; // Period of function
        const int N = 10; // Order for the trigonometric polynomial

        y = 0;

        //freopen ("fourierseries.txt","w",stdout); // You can redirect
            stdout, stderr and stdin using std::freopen
        // Fourier series of the function f(x), -pi <= x <= pi
        // x = sum_{1}^{N} (2/n) * (-1)^(n+1) * sin(nx)
        for ( int n = 1; n <= N; n++ )
        {
                y += division(2,n)*pow(-1,n+1)*sin( n*x );
                cout << n <<" " << y << endl;
        }

        return 0;
}
```

**C++ Code 255:** *main.cpp "Compute fourier series with known f(x)"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -lsymbolicc++**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- You can uncomment the code with **freopen ("fourierseries.txt","w",stdout)** so the first $N$ terms of Fourier series will be saved as textfile **fourierseries.txt** at current working directory. If you comment this line of code then the Fourier series will be printed on the terminal / xterm.

```
        ...
        ...

        freopen ("fourierseries.txt","w",stdout); // You can redirect
            stdout, stderr and stdin using std::freopen
        // Fourier series of the function f(x), -pi <= x <= pi
        // x = sum_{1}^{N} (2/n) * (-1)^(n+1) * sin(nx)
        for ( int n = 1; n <= N; n++ )
```

```
        {
                y += division(2,n)*pow(-1,n+1)*sin( n*x );
                cout << n <<" " << y << endl;
        }

        ...
```

```
1 2*sin(x)
2 2*sin(x)-sin(2*x)
3 2*sin(x)-sin(2*x)+0.666667*sin(3*x)
4 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)
5 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)
6 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)-0.333333*sin(6*x)
7 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)-0.333333*sin(6*x)+0.285714*sin(7*x)
8 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)-0.333333*sin(6*x)+0.285714*sin(7*x)-0.25*sin(8*x)
9 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)-0.333333*sin(6*x)+0.285714*sin(7*x)-0.25*sin(8*x)+0.222222*sin(9*x)
10 2*sin(x)-sin(2*x)+0.666667*sin(3*x)-0.5*sin(4*x)+0.4*sin(5*x)-0.333333*sin(6*x)+0.285714*sin(7*x)-0.25*sin(8*x)+0.222222*sin(9*x)-0.2*sin(10*x)
```

**Figure 23.258:** *The computation to determine the Fourier series to approximate the function $f(x) = x, -\pi \leq x, \leq \pi$ with order from $n = 1$ to $n = 10$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Fourier Series with Known f(x)/main.cpp**).*

The second C++ code will be working on the plotting part, we do it manually with **gnuplot**, after we get the Fourier series approximation from the code above, then we can input the first four terms of the Fourier series that approximate the function $f(x) = x, -\pi \leq x, \leq \pi$.

```
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-10:10]\nset yrange [-5:5]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = 2*sin(x)\n";
        gp << "g(x) = 2*sin(x)-sin(2*x)\n";
        gp << "h(x) = 2*sin(x)-sin(2*x) + 0.6666*sin(3*x)\n";
        gp << "i(x) = 2*sin(x)-sin(2*x) + 0.6666*sin(3*x) - 0.5*sin(4*x)\n
            ";
        gp << "plot f(x) title 'x = 2 sin x' lw 3 lc rgb 'forest-green', \
        g(x) title 'x = 2 sin x - sin 2x' lw 3 lc rgb 'spring-green', \
        h(x) title 'x = 2 sin x - sin 2x + (2/3) sin 3x' lw 3 lc rgb '
            orchid4',\
```

```
            i(x) title 'x = 2 sin x — sin 2x + (2/3) sin 3x — (2/4) sin 4x' lw
                3 lc rgb 'light—salmon'\n";

            return 0;
    }
```

**C++ Code 256:** *main.cpp "Plot 2D fourier series with known f(x)"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**

Each term of the series has period $2\pi$, it is the smallest period common to all terms, so the sum also has period $2\pi$. This means that the known graph of the sum between $-\pi$ and $\pi$ is simply repeated on each successive interval of length $2\pi$ to the right and left.

The graph of the sum therefore has the sawtooth appearance. It is clear from this that the sum of the series is equal to $x$ only on the interval $-\pi < x < \pi$, and not on the entire real line $-\infty < x < \infty$.

The sum of the serues has a sudden jump from $-\pi$ to $+\pi$ at the points $x = \pm\pi, \pm 3\pi, \pm 5\pi, \ldots$. By putting $x = \pm\pi, \pm 3\pi, \pm 5\pi, \ldots$ into the Fourier series here, every term of the series is zero. Therefore the sum is also zero.

Now, we plot up to the sum of the first four terms for the fourier series, which is

$$y = 2\left(\sin\ x - \frac{\sin\ 2x}{2} + \frac{\sin\ 3x}{3} - \frac{\sin\ 4x}{4}\right)$$

Since this is a partial sum of the Fourier series, and the series converges to $x$, for $-\pi \leq x, \leq \pi$, we expect the partial sum to approximate the function $y = x$ on this interval. As the number of terms increases, the approximating curves approach $y = x$ for each fixed $x$ on the interval $-\pi < x, < \pi$, but not for $\pm\pi$.

**Figure 23.259:** *The plot of Fourier series that approximate the function $f(x) = x, -\pi \leq x, \leq \pi$ with order from $n = 1$ to $n = 4$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Fourier Series with Known f(x) Sawtooth/main.cpp**).*

## iii. C++ Plot: Find the Fourier Series of Two different functions $f(x)$

Find the Fourier series of the function defined by

$$f(x) = 0, \quad -\pi \leq x < 0$$
$$f(x) = \pi, \quad 0 \leq x \leq \pi$$

**Solution:**

We will compute the Fourier coefficients

$$a_0 = \frac{1}{\pi}\left[\int_{-\pi}^{0} 0\, dx + \int_{0}^{\pi} \pi\, dx\right] = \pi$$
$$a_n = \frac{1}{\pi}\left[\int_{0}^{\pi} \pi\ \cos\ nx\, dx\right] = 0$$
$$b_n = \frac{1}{\pi}\left[\int_{0}^{\pi} \pi\ \sin\ nx\, dx\right] = \frac{1}{n}(1 - \cos\ n\pi) = \frac{1}{n}\left[1 - (-1)^n\right]$$

we will assume that $\int_{-\pi}^{0} 0\, dx$ is very small / almost zero and can be negligible. Since the $n$th even number is $2n$ and the $n$th odd number is $2n-1$ the last of these formulas tells us that

$$b_{2n} = 0$$
$$b_{2n-1} = \frac{2}{2n-1}$$

We obtain the Fourier series as

$$\begin{aligned} f(x) &= \frac{1}{2}a_0 + \sum_{n=1}^{\infty}(a_n \cos\ nx + b_n \sin\ nx) \\ &= \frac{\pi}{2} + \sum_{n=1}^{\infty} b_{2n-1} \sin\ (2n-1)x \\ &= \frac{\pi}{2} + \sum_{n=1}^{\infty} \frac{2}{2n-1} \sin\ (2n-1)x \\ f(x) &= \frac{\pi}{2} + 2\left(\sin\ x + \frac{\sin\ 3x}{3} + \frac{\sin\ 5x}{5} + \dots\right) \end{aligned}$$

The successive partial sums are

$$\begin{aligned} y &= \frac{\pi}{2} + 2\sin\ x \\ y &= \frac{\pi}{2} + 2\left(\sin\ x + \frac{\sin\ 3x}{3}\right) \\ y &= \frac{\pi}{2} + 2\left(\sin\ x + \frac{\sin\ 3x}{3} + \frac{\sin\ 5x}{5} + \dots\right) \\ &\vdots \end{aligned}$$

The Fourier series here converges to the function $f(x) = 0$ on the subintervals $-\pi < x < 0$ and to the function $f(x) = \pi$ on the interval $0 < x < \pi$, but not at the points $0, \pi, -\pi$.

The sum of the series is clearly periodic with period $2\pi$, and therefore the graph of this sum has the square wave appearance, with a jump from 0 to $\pi$ at each point $x = 0, \pm\pi, \pm 2\pi, \ldots$.

This sum evidently has the value $\pi/2$ at each of these points of discontinuity.

We compute the integral manually by hand, and do the plotting with C++' **gnuplot**.

```
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set terminal 'wxt' size 1200,800 enhanced font 'Verdana,10'
            persist\n"; // To make the plot interactive
        gp << "set xrange [-1:10]\nset yrange [-2:5]\n";
        gp << "set xlabel 'x-axis'\n set ylabel 'y-axis'\n";
        gp <<"set xtics nomirror (0, '{/:Italic pi/2}' 0.5*pi, '{/:Italic pi
            }' pi, '{/:Italic 3 pi/2}' 1.5*pi,\
        '2{/:Italic pi}' 2*pi, '{/:Italic 5 pi/2}' 2.5*pi, '3{/:Italic pi}'
            3*pi, '{/:Italic 7 pi/2}' 3.5*pi,\
        '4{/:Italic pi}' 4*pi, '5{/:Italic pi}' 5*pi, '6{/:Italic pi}' 6*pi)
            \n";

        // '-' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "f(x) = (0.5*3.14) + 2*sin(x)\n";
        gp << "g(x) = (0.5*3.14) + 2*sin(x) + 0.6666*sin(3*x)\n";
        gp << "h(x) = (0.5*3.14) + 2*sin(x) + 0.6666*sin(3*x) + 0.4*sin(5*x)
            \n";
        gp << "i(x) = (0.5*3.14) + 2*sin(x) + 0.6666*sin(3*x) + 0.4*sin(5*x)
             + 0.2857*sin(7*x)\n";
        gp << "plot f(x) title 'y = pi/2 + 2 sin x' lw 3 lc rgb 'forest-
            green', \
        g(x) title 'y = pi/2 + 2 sin x + (2/3) sin 3x' lw 3 lc rgb 'spring-
            green', \
        h(x) title 'y = pi/2 + 2 sin x + (2/3) sin 3x + (2/5) sin 5x' lw 3
            lc rgb 'orchid4',\
        i(x) title 'y = pi/2 + 2 sin x + (2/3) sin 3x + (2/5) sin 5x + (2/7)
             sin 7x' lw 3 lc rgb 'light-salmon'\n";

        return 0;
}
```

**C++ Code 257:** *main.cpp "Plot fourier series with two different f(x) for square wave"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

**Figure 23.260:** *The plot of Fourier series that converges to the function $f(x) = 0$ on the subintervals $-\pi < x < 0$ and to the function $f(x) = \pi$ on the interval $0 < x < \pi$, but not at the points $0, \pi, -\pi$ with order from $n = 1$ to $n = 4$. (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Fourier Series with Known f(x) Square Wave/main.cpp**).*

## iv. C++ Simulation: Fourier Series

I have found a great reference to create Fourier series reference from internet:
**https://markus-x-buchholz.medium.com/fourier-series-in-c-imgui-1587e866586e**

The original source code can be obtained here:
**https://github.com/markusbuchholz/Fourier-Series-in-Cpp-ImGui/**

Suppose we want to create Fourier series simulation for the first $n$ terms, this section will create the simulation of The Fourier series' square wave that can be composed by adding sine signals with this formula

$$f(x) = \sum_{n=1}^{\infty} \left( \frac{4}{\pi(2n-1)} \right) \sin \left( \frac{\pi(2n-1)x}{P} \right) \tag{23.318}$$

then we will have for the simulation

1. $n$-amount of circles on the left, the first circle is the corresponding first term of the Fourier series $f(x)$, the radius of the circle is associated with the amplitude of the sine signal. The speed of the point that is rotating on the circle equals the sine signal period $P$, one rotation for the circle is the same as one period for the corresponding sine wave of the Fourier series $f(x)$.
2. The Fourier series $f(x)$ for the first $n$ terms on the right side of the circle(s)

More sine signal will add more circles with a smaller radius. Decreasing the circle radius affects the speed rotation of the point around the original circle (the smalller circle will rotate faster than the bigger circle).

The circle that is rotating around circle around another circle is the analogy for the earth that is revolving around the Sun and the moon that is revolving around the Earth. The Moon finishes the revolution after 29 days (go back to its' original position / finish 1 period of trigonometric wave), while the Earth finishes the revolution after 365 days.

Besides square wave, we will also make simulation for Fourier series with appearance of Sawtooth.

The C++ code is using **Imgui** as the widget tool so we can define the number of period / terms for the Fourier series. The directory has structure of

- include
  - stdafx.hpp (headers only)
- src
  - imgui folder
    (this folder can also be obtained after you install Imgui manually, the Imgui version here is **Dear Imgui, v1.80 WIP**, if your installed Imgui in **/usr/include/imgui** has different version, then no need to worry or install new Imgui, since the source code here will be compiled with CMake specifically will look for Imgui from the current working directory **../src/imgui/**)
  - ball.h

- fourier.cpp (can be treated like main.cpp, you edit and modify the source code mainly from here)
- imgui_helper.hpp

```
#include <stdafx.hpp>
#include "imgui_helper.hpp"
#include "ball.h"
#include <tuple>
#include <thread>
#include <chrono>
#include <vector>
#include <math.h>
#include <list>
#include <numeric>
#define pi 3.1415926535897

using namespace std;
//————————————
```

```
                255.0, 1.00f);
            ImVec4 ray_color = ImVec4(1.0f, 1.0f, 1.0f, 1.0f);
            ImVec4 ray_color2 = ImVec4(0.5f, 0.5f, 0.5f, 1.0f);


            // Main loop
            int ii = 0;
            bool flag = true;
            std::list<ImVec2> wave_tail;
            std::list<int> test;


            while (!glfwWindowShouldClose(window) && flag == true)
            {
                   glfwPollEvents();

                   ImGui_ImplOpenGL3_NewFrame();
                   ImGui_ImplGlfw_NewFrame();

                   ImGui::NewFrame();

                   static int num_sig = 3;

                   ImGuiWindowFlags window_flags = 0;
                   ImGui::SetNextWindowPos(ImVec2(0, 0), ImGuiCond_FirstUseEver)
                       ;
                   ImGui::SetNextWindowSize(ImVec2(w, h), ImGuiCond_FirstUseEver
                       );
                   window_flags |= ImGuiWindowFlags_NoMove;
                   window_flags |= ImGuiWindowFlags_NoResize;
                   window_flags |= ImGuiWindowFlags_NoCollapse;
                   window_flags |= ImGuiWindowFlags_NoBackground;
                   window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus;

                   ImGui::Begin("Simulation", nullptr, window_flags);
                   ImDrawList *draw_list = ImGui::GetWindowDrawList();

                   float stroke = 150.0; //the center of the circle is at stroke
                        = 0
                   std::vector<int> N = {1, 3, 5};
                   float r1 = 4.0 / (N[0] * M_PI) * scale;
                   float r2 = 4.0 / (N[1] * M_PI) * scale;
                   float r3 = 4.0 / (N[2] * M_PI) * scale;
                   float size_line = 3;

                   std::vector<ImVec2> circles;
                   std::vector<ImVec2> dots_circles;
                   std::vector<float> vec_r;
```

```
float x = (float)points[0].position.x;
float y = (float)points[0].position.y;

std::vector<float> vec_w;

// this for loop is to create the dots and radius lines that
    is circling the all circles
// For square wave
///*
for (int i = 0; i < num_sig; i++)
{

       float prev_x = x;
       float prev_y = y;
       circles.push_back({prev_x, prev_y});

       int n = 2*i + 1;

       float r = 4.0 / (n * M_PI) * scale;
       vec_r.push_back(r);

       // circle dot
       // f(x) = sum_{n}^{infty} ( 4 / ( pi * (2n-1) ) ) *
           sin ((2n-1)*pi*x/P)
       x += (float)std::cos(ii * dt * n / P) * r;
       y += (float)std::sin(ii * dt * n / P) * r;
       vec_w.push_back((float)std::sin(ii * dt * n / P) * r )
           ;
       dots_circles.push_back({x, y});
}
//*/

// For sawtooth wave
/*
for (int i = 0; i < num_sig; i++)
{

       float prev_x = x;
       float prev_y = y;
       circles.push_back({prev_x, prev_y});

       int n = i + 1;

       float r = 4.0 / (n * M_PI) * scale;
       vec_r.push_back(r);

       // circle dot
```

```
            // f(x) = sum_{n}^{infty} 2 * (-1)^{n+1} * sin (n*pi*
                x/P) * r
            x += (float)std::cos(ii * dt * n / P) * r;
            y += (float)std::sin(ii * dt * n / P) * r;
            vec_w.push_back((float)pi - 2 * (pow(-1,n+1)) *std::
                sin(ii * dt * n / P) * r );
            //vec_w.push_back((float)2*std::sin(ii * dt * n / P) *
                 r );
            dots_circles.push_back({x, y});
        }
        */

        // This for loop is to create all the circles
        for (int jj = 0; jj < circles.size(); jj++)
        {

            draw_list->AddLine(circles[jj], dots_circles[jj],
                ImColor(ray_color), 2.0);
            draw_list->AddCircle(circles[jj], vec_r[jj], ImColor(
                ray_color));
        }

        ImVec2 wave_N = {(float)points[0].position.x + ii + stroke, (
            float)points[0].position.y + std::accumulate(vec_w.begin
            (), vec_w.end(),0)};

        draw_list->AddLine(dots_circles[dots_circles.size() - 1],
            wave_N, ImColor(ray_color), 2.0);

        wave_tail.push_front(wave_N);

        // This for loop is the one running the movement of the dot
            in the circle and the sawtooth wave that is being drawn
        for (auto &it : wave_tail)
        {
            draw_list->AddCircleFilled(it, 1.0, IM_COL32(45, 0,
                255, 255));
        }
        ii++;
        // the if statement to restart the drawing after variable ii
            reach 801
        if (ii > 800)
        {
            wave_tail.erase(wave_tail.begin(), wave_tail.end());
            ii = 0;
        }
        ImGui::PushItemWidth(200);
        ImGui::SliderInt("Number of periods", &num_sig, 1, 20);
```

```
                    ImGui::Text("The Fourier series wave:");
                    ImGui::Text("x: %d ", ii);
                    ImGui::Text("y(x) : %d ", 400 + std::accumulate(vec_w.begin()
                        , vec_w.end(),0) );
                    ImGui::PopItemWidth();
                    ImGui::End();


                    // Rendering
                    ImGui::Render();
                    int display_w, display_h;
                    glfwGetFramebufferSize(window, &display_w, &display_h);
                    glViewport(0, 0, display_w, display_h);
                    glClearColor(clear_color.x, clear_color.y, clear_color.z,
                        clear_color.w);
                    glClear(GL_COLOR_BUFFER_BIT);
                    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

                    glfwSwapBuffers(window);
            }

            termImgui();
            return 0;
    }
```

**C++ Code 258:** *src/fourier.cpp "Fourier series simulation for sawtooth and square wave"*

To compile it, go to the directory **DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Fourier Series Sawtooth and Square Wave Simulation with Imgui/**, then type:
**mkdir build**
**cd build**
**cmake ..**
**make**
**./2D_Fourier_Series_Sawtooth_and_Square_Wave_Simulation_with_Imgui**

Explanation for the codes:

- There are some declared values for variables that you need to know:

$$M_PI = 12$$
$$stroke = 150$$

At the beginning **std::vector<Point> points** defined the center for the first circle, which is $(200, 400)$.

```
            float)points[0].position.x = 200
            (float)points[0].position.y = 400
```

In this **Imgui** screen the $y$ value here is getting larger when you go with down direction on the screen and getting smaller when you go with up direction on the screen.

- If you want to plot Fourier seriers with Square Wave appearance of this formula

$$f(x) = \sum_{n=1}^{\infty} \left( \frac{4}{\pi(2n-1)} \right) \sin \left( \frac{\pi(2n-1)x}{P} \right) \tag{23.319}$$

you may use this inside the **for** loop:

```
...
...

for (int i = 0; i < num_sig; i++)
{

       float prev_x = x;
       float prev_y = y;
       circles.push_back({prev_x, prev_y});

       int n = 2*i + 1;

       float r = 4.0 / (n * M_PI) * scale;
       vec_r.push_back(r);

       // circle dot
       // f(x) = sum_{n}^{infty} ( 4 / ( pi * (2n-1) ) ) * sin
           ((2n-1)*pi*x/P)
       x += (float)std::cos(ii * dt * n / P) * r;
       y += (float)std::sin(ii * dt * n / P) * r;
       vec_w.push_back((float)std::sin(ii * dt * n / P) * r );
       dots_circles.push_back({x, y});
}
...
...
```

If you want to plot the Fourier series with descending Sawtooth appearance of this formula

$$f(x) = 2 \sum_{n=1}^{\infty} (-1)^{n+1} \sin \left( \frac{n\pi x}{P} \right) \tag{23.320}$$

you may use this inside the **for** loop:

```
vec_w.push_back((float)2 * (pow(-1,n+1)) *std::sin(ii * dt * n
     / P) * r );
```

If you want to plot the Fourier series with ascending Sawtooth appearance of this formula

$$f(x) = \pi - 2 \sum_{n=1}^{\infty} (-1)^{n+1} \sin \left( \frac{nx}{n} \right) \tag{23.321}$$

you may use this inside the **for** loop:

```
            vec_w.push_back((float)pi - 2 * (pow(-1,n+1)) *std::sin(ii *
                dt * n / P) * r );
```

- TO draw the Square wave, you may uncomment this for loop

```
            // For square wave
            /*
            for (int i = 0; i < num_sig; i++)
            {

                float prev_x = x;
                float prev_y = y;
                circles.push_back({prev_x, prev_y});

                int n = 2*i + 1;

                float r = 4.0 / (n * M_PI) * scale;
                vec_r.push_back(r);

                // circle dot
                // f(x) = sum_{n}^{infty} ( 4 / ( pi * (2n-1) ) ) * sin
                    ((2n-1)*pi*x/P)
                x += (float)std::cos(ii * dt * n / P) * r;
                y += (float)std::sin(ii * dt * n / P) * r;
                vec_w.push_back((float)std::sin(ii * dt * n / P) * r );
                dots_circles.push_back({x, y});
            }
            */
```

and comment this for loop

```
            // For sawtooth wave
            for (int i = 0; i < num_sig; i++)
            {

                float prev_x = x;
                float prev_y = y;
                circles.push_back({prev_x, prev_y});

                int n = i + 1;

                float r = 4.0 / (n * M_PI) * scale;
                vec_r.push_back(r);

                // circle dot
                // f(x) = sum_{n}^{infty} 2 * (-1)^{n+1} * sin (n*pi*x/
                    P) * r
                x += (float)std::cos(ii * dt * n / P) * r;
                y += (float)std::sin(ii * dt * n / P) * r;
```

```
                    vec_w.push_back((float)pi - 2 * (pow(-1,n+1)) *std::sin
                        (ii * dt * n / P) * r );
                    //vec_w.push_back((float)2*std::sin(ii * dt * n / P) * r
                         );
                    dots_circles.push_back({x, y});

                }
```

**Figure 23.261:** *The simulation of Fourier series with ascending sawtooth appearance that uses formula* $f(x) = \pi - 2\sum_{n=1}^{\infty}(-1)^{n+1}\sin\left(\frac{nx}{n}\right)$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Fourier Series Sawtooth and Square Wave Simulation with Imgui/)****.*

**Figure 23.262:** *The simulation of Fourier series with square wave appearance that uses formula* $f(x) = \sum_{n=1}^{\infty}\left(\frac{4}{\pi(2n-1)}\right)\sin\left(\frac{\pi(2n-1)x}{P}\right)$ ***(DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Fourier Series Sawtooth and Square Wave Simulation with Imgui/).***

## XXI. LU-Decomposition

[DF*] We have learned two methods for solving linear systems:

1. Gaussian elimination (reduction to row echelon form)
2. Gauss-Jordan elimination (reduction to reduced row echelon form)

These methods are fine for small-scale problems, but not suitable for large-scale problems in which computer roundoff error, memory usage, and speed are concerns.

For large-scale problems, $LU$-Decomposition is the basis for many computer algorithms in common use for solving linear system of $n$ equations in $n$ unknowns that is based on factoring its coefficient matrix into a product of lower and upper triangular matrices.

[DF*] LU-Decomposition is learning how to solve a linear system

$$A\boldsymbol{x} = \boldsymbol{b}$$

of $n$ equations in $n$ unknowns by factoring the coefficient matrix $A$ into a product

$$A = LU \tag{23.322}$$

where $L$ is lower triangular and $U$ is upper triangular.

[DF*] The method of $LU$-Decomposition:

1. Rewrite the system $A\boldsymbol{x} = \boldsymbol{b}$ as

$$LU\boldsymbol{x} = \boldsymbol{b} \tag{23.323}$$

2. Define a new $n \times 1$ matrix $\boldsymbol{y}$ by
$$U\boldsymbol{x} = \boldsymbol{y} \tag{23.324}$$
3. Use $U\boldsymbol{x} = \boldsymbol{y}$ as $L\boldsymbol{y} = \boldsymbol{b}$ and solve this system for $\boldsymbol{y}$. We solve the equations from top down with forward substitution.
4. Substitute $\boldsymbol{y}$ in $U\boldsymbol{x} = \boldsymbol{y}$ and solve for $\boldsymbol{x}$. We solve the equations from bottom up with back substitution.

This procedure replaces the single linear system $A\boldsymbol{x} = \boldsymbol{b}$ by a pair of linear systems

$$\begin{aligned} U\boldsymbol{x} &= \boldsymbol{y} \\ L\boldsymbol{y} &= \boldsymbol{b} \end{aligned}$$

This must be solved in succession.

**Definition 23.71: $LU$-Decomposition**

A factorization of a square matrix $A$ as $A = LU$, where $L$ is lower triangular and $U$ is upper triangular is called an $LU$-decomposition (or $LU$-factorization) of $A$.

Not every square matrix has an $LU$-decomposition.

**Theorem 23.160: Gaussian Elimination and $LU$-Decomposition**

If $A$ is a square matrix that can be reduced to a row echelon form $U$ by Gaussian elimination without row interchanges, then $A$ can be factored as $A = LU$, where $L$ is a lower triangular matrix.

[DF*] **Example: Solving $A\boldsymbol{x} = \boldsymbol{b}$ by $LU$-Decomposition**
Solve the linear system
$$\begin{bmatrix} 2 & 6 & 2 \\ -3 & -8 & 0 \\ 4 & 9 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

**Solution:**
We can rewrite this system as

$$\begin{bmatrix} 2 & 0 & 0 \\ -3 & 1 & 0 \\ 4 & -3 & 7 \end{bmatrix} \begin{bmatrix} 1 & 3 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

Let us define $y_1, y_2$, and $y_3$ by the equation $U\boldsymbol{x} = \boldsymbol{y}$

$$\begin{bmatrix} 1 & 3 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

then we can write the equation $L\boldsymbol{y} = \boldsymbol{b}$

$$\begin{bmatrix} 2 & 0 & 0 \\ -3 & 1 & 0 \\ 4 & -3 & 7 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

the equation $L\boldsymbol{y} = \boldsymbol{b}$ can be written equivalently as

$$\begin{array}{llll} 2y_1 & & & = 2 \\ -3y_1 & + y_2 & & = 2 \\ 4y_1 & - 3y_2 & + 7y_3 & = 3 \end{array}$$

now by using forward substitution we will obtain

$$y_1 = 1,\ y_2 = 5,\ y_3 = 2$$

We go back to the equation $U\boldsymbol{x} = \boldsymbol{y}$ and substitute $y$ into the equation, which yields the linear system

$$\begin{bmatrix} 1 & 3 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix}$$

or equivalently,

$$\begin{array}{llll} x_1 & + 3x_2 & + x_3 & = 1 \\ & x_2 & + 3x_3 & = 5 \\ & & x_3 & = 2 \end{array}$$

now by using back substitution we will obtain

$$x_1 = 2,\ x_2 = -1,\ x_3 = 2$$

[DF*] To systemically solve for entries in $L$ and $U$ from matrix $A$, the formula for matrix $U$:

$$\forall j \begin{cases} i = 0 \to U_{ij} = A_{ij} \\ i > 0 \to U_{ij} = A_{ij} - \sum_{k=0}^{i-1} L_{ik} U_{kj} \end{cases} \tag{23.325}$$

the formula for matrix $L$:

$$\forall i \begin{cases} j = 0 \to L_{ij} = \dfrac{A_{ij}}{U_{jj}} \\ j > 0 \to L_{ij} = \dfrac{A_{ij} - \sum_{k=0}^{j-1} L_{ik} U_{kj}}{U_{jj}} \end{cases} \tag{23.326}$$

[DF*] The Gaussian elimination applied to an arbitrary linear system $A\boldsymbol{x} = \boldsymbol{b}$ requires $O\left(\frac{n^3}{3}\right)$ arithmetic operations to determine $\boldsymbol{x}$.

If $A$ has been factored into the triangular form $A = LU$, then we can solve for $\boldsymbol{x}$ more easily by using two-step process. First we let $\boldsymbol{y} = U\boldsymbol{x}$ and solve the system $L\boldsymbol{y} = b$ for $\boldsymbol{y}$. Since $L$ is triangular, determining $\boldsymbol{y}$ from this equation requires only an additional $O(n^2)$ operations to determine the solution $\boldsymbol{x}$.

This fact means that the number of operations needed to solve the system $A\boldsymbol{x} = \boldsymbol{b}$ is reduced from $O\left(\frac{n^3}{3}\right)$ to $O(2n^2)$. In systems greater than 100 by 100, this can reduce the amount of calculation by more than 97%. Determining the specific matrices $L$ and $U$ requires $O\left(\frac{n^3}{3}\right)$ operations. Once the factorization is determined, a system involving the matrix $A$ and any vector $\boldsymbol{b}$ can be solved in this simplified manner.

## i. C++ Computation: LU Decomposition

The C++ code below is used for square matrix $A$ of size 3, to obtain the solution for $A\boldsymbol{x} = \boldsymbol{b}$ after we decompose $A$ into $LU$ we perform Gauss-Jordan elimination two times, first to get the solution for $L\boldsymbol{y} = \boldsymbol{b}$ then to get the solution for $U\boldsymbol{x} = \boldsymbol{y}$.

```
#include <bits/stdc++.h>
#include <fstream>
#include <vector>
#include <cmath>

#define N 3
#define M 4
using namespace std;

// this LU decomposition function works
void LUdecomposition(float a[N][N], float l[N][N], float u[N][N], int n)
{
	int i = 0, j = 0, k = 0;
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < n; j++)
		{
			if (j < i)
			l[j][i] = 0;
			else
			{
				l[j][i] = a[j][i];
				for (k = 0; k < i; k++)
				{
					l[j][i] = l[j][i] - l[j][k] * u[k][i];
				}
			}
		}
		for (j = 0; j < n; j++)
		{
			if (j < i)
			u[i][j] = 0;
			else if (j == i)
			u[i][j] = 1;
			else
			{
				u[i][j] = a[i][j] / l[i][i];
				for (k = 0; k < i; k++)
				{
					u[i][j] = u[i][j] - ((l[i][k] * u[k][j])
						/ l[i][i]);
				}
			}
```

```
                }
        }
}

// function to reduce matrix to reduced row echelon form.
int PerformOperation(float matL[][M], int n)
{
        int i, j, k = 0, c, flag = 0, m = 0;
        float pro = 0;

        // Performing elementary operations
        for (i = 0; i < n; i++)
        {
                if (matL[i][i] == 0)
                {
                        c = 1;
                        while ((i + c) < n && matL[i + c][i] == 0)
                        c++;
                        if ((i + c) == n)
                        {
                                flag = 1;
                                break;
                        }
                        for (j = i, k = 0; k <= n; k++)
                        swap(matL[j][k], matL[j+c][k]);
                }

                for (j = 0; j < n; j++)
                {
                        // Excluding all i == j
                        if (i != j)
                        {
                                // Converting Matrix to reduced row
                                // echelon form(diagonal matrix)
                                float pro = matL[j][i] / matL[i][i];
                                for (k = 0; k <= n; k++)
                                matL[j][k] = matL[j][k] - (matL[i][k]) * pro;
                        }
                }
        }
        return flag;
}

// Function to print the desired result
void PrintResult(float matL[][M], int n, int flag)
{
        cout << "The solution/s for Ly = b : " << endl;
```

```
        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "y(" << i << ") = "<< matL[i][n] / matL[i][i] << ", "
                    ;
        }
}

void PrintResult2(float matU[][M], int n, int flag)
{
        cout << "The solution/s for Ux = y : " << endl;

        if (flag == 2)
        {
                cout << "Infinite Solutions Exists" << endl;
        }
        else if (flag == 3)
        {
                cout << "No Solution Exists" << endl;
        }

        // Printing the solution by dividing constants by
        // their respective diagonal elements
        else
        {
                for (int i = 0; i < n; i++)
                cout << "x(" << i << ") = "<< matU[i][n] / matU[i][i] << ", "
                    ;
        }
}

// To check whether infinite solutions exists or no solution exists
int CheckConsistency(float matL[][M], int n, int flag)
{
        int i, j;
        float sum;
```

```
        // flag == 2 for infinite solution
        // flag == 3 for No solution
        flag = 3;
        for (i = 0; i < n; i++)
        {
                sum = 0;
                for (j = 0; j < n; j++)
                sum = sum + matL[i][j];
                if (sum == matL[i][j])
                flag = 2;
        }
        return flag;
}

// Function to print the matrix
void PrintMatrix(float matL[][M], int n)
{
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j <= n; j++)
                cout << setw(6) << matL[i][j] << "\t";
                cout << endl;
        }
        cout<<endl;
}

void display(int matL[N][N], int row, int col)
{
        cout << setw(14) << "Matrix A" << endl;
        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < col; j++)
                cout<<setw(6) << matL[i][j]<<"\t";
                cout<<endl;
        }
        cout<<endl;
}

// take vectorb.txt as input from textfile
int* vec() {
        static int x[N];
        std::fstream in("vectorb.txt");
        float vectortiles[N];
        for (int i = 0; i < N; ++i)
        {
                in >> vectortiles[i];
                x[i] =vectortiles[i];
        }
```

```
        return x;
}

// Driver code
int main()
{
        int mat[N][N] = { { 2, 6, 2 }, { -3, -8, 0 }, { 4, 9, 2 } };
        display(mat,3,3);

        // declare the matrix as float
        float a[3][3] = { { 2, 6, 2 }, { -3, -8, 0 }, { 4, 9, 2 } };

        float l[3][3], u[3][3];
        LUdecomposition(a, l, u, N);
        // setw is for displaying nicely
        cout << setw(21) << "Lower Triangular"<< setw(32) << "Upper
            Triangular" << endl;


        // Displaying the result :
        for (int i = 0; i < N; i++)
        {
                // Show Lower Triangular Matrix
                for (int j = 0; j < N; j++)
                cout << setw(6) << l[i][j] << "\t";
                cout << "\t";

                // Show Upper Triangular Matrix
                for (int j = 0; j < N; j++)
                cout << setw(6) << u[i][j] << "\t";
                cout << endl;
        }
        cout << endl;


        // Input for the Lower triangular matrix from array of the computed
            result of LUdecomposition()
        int* ptrvec;
        ptrvec = vec();

        float matL[M][M] = {0};
        for(int j=0; j<N; j++)
        {
                for(int k=0; k<N; k++)
                {
                        matL[j][k] = l[j][k];
                }
                matL[j][N] = ptrvec[j];
```

```
        }

        cout << "Augmented Matrix L to solve Ly = b : " << endl;

        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N+1; ++j)
                {
                        cout << setw(6) << matL[i][j] << "\t";
                }
                cout << endl;
        }

        int n1 = 3, flag = 0, flagU = 0;
        //Performing Matrix transformation
        // Solving for Ly = b
        flag = PerformOperation(matL, n1);

        if (flag == 1)
        flag = CheckConsistency(matL, n1, flag);

        //Printing Final Matrix
        cout << endl;
        cout << "Reduced Row Echelon for Augmented Matrix L is : " << endl;
        PrintMatrix(matL, n1);

        // Printing Solutions(if exist)
        PrintResult(matL,n1, flag);
        cout << endl;
        cout << endl;

        // Solving for Ux = y
        // We can declare y1, y2, y3 after we do the PerformOperation(matL,
            n1) to make the reduced row echelon form of Ly=b
        // Store the result in vector y[N] of size N
        float y[N];
        for (int i = 0; i < N; ++i)
        {
                y[i] = matL[i][N] / matL[i][i];
        }

        // Input for the Lower triangular matrix from array of the computed
            result of LUdecomposition()
        float matU[M][M] = {0};
        for(int j=0; j<N; j++)
        {
                for(int k=0; k<N; k++)
                {
```

```
                        matU[j][k] = u[j][k];
                }
                matU[j][N] = y[j];
        }

        cout << "Augmented Matrix U to solve Ux = y : " << endl;

        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N+1; ++j)
                {
                        cout << setw(6) << matU[i][j] << "\t";
                }
                cout << endl;
        }
        flagU = PerformOperation(matU, n1);

        if (flagU == 1)
        flagU = CheckConsistency(matU, n1, flag);

        cout << "Reduced Row Echelon for Augmented Matrix U is : " << endl;
        PrintMatrix(matU, n1);
        cout << endl;
        PrintResult2(matU,n1, flag);

        cout << endl;

        return 0;
}
```

**C++ Code 259:** *main.cpp "LU Decomposition"*

To compile it, type:
**g++ -o result main.cpp**
**./result**

or with the Makefile, type:
**make**
**./main**

Explanation for the codes:

- The first function is the **void LUdecomposition()** that is used to decompose matrix $A$ into $L$ and $U$.

```
void LUdecomposition(float a[N][N], float l[N][N], float u[N][N
    ], int n)
{
        int i = 0, j = 0, k = 0;
        for (i = 0; i < n; i++)
```

```
                {
                        for (j = 0; j < n; j++)
                        {
                                if (j < i)
                                l[j][i] = 0;
                                else
                                {
                                        l[j][i] = a[j][i];
                                        for (k = 0; k < i; k++)
                                        {
                                                l[j][i] = l[j][i] - l[j][k]
                                                    * u[k][i];
                                        }
                                }
                        }
                        for (j = 0; j < n; j++)
                        {
                                if (j < i)
                                u[i][j] = 0;
                                else if (j == i)
                                u[i][j] = 1;
                                else
                                {
                                        u[i][j] = a[i][j] / l[i][i];
                                        for (k = 0; k < i; k++)
                                        {
                                                u[i][j] = u[i][j] - ((l[i][k
                                                    ] * u[k][j]) / l[i][i]);
                                        }
                                }
                        }
                }
        }
```

- We are taking vector $\boldsymbol{b}$ from textfile (**vectorb.txt**) as the input so we are using **#include <fstream>** and **#include <vector>**.
- The reason why we create **void PrintResult** and **void PrintResult2** is to make the output of solution in different letter of the variable, the prior for variable $y$, then the last for variable $x$.
- To create the augmented matrix $L$ with the vector $\boldsymbol{b}$ so we don't need to manually enter the values / the array indices for **matL[M][M]**.

```
        int* ptrvec;
        ptrvec = vec();

        float matL[M][M] = {0};
        for(int j=0; j<N; j++)
        {
                for(int k=0; k<N; k++)
                {
```

```
                        matL[j][k] = l[j][k];
                }
                matL[j][N] = ptrvec[j];
        }
```

- After we finish with **matL[M][M]** above and get the solution from Gauss-Jordan elimination as the vector $\boldsymbol{y}$, we create a vector and declare it as float **y[N]** and then create the augmented matrix $U$. With this we can perform the last Gauss-Jordan elimination to obtain vector $\boldsymbol{x}$ as the final solution that we are looking for with $LU$-Decomposition method.

$$A\boldsymbol{x} = \boldsymbol{b}$$

$$LU\boldsymbol{x} = \boldsymbol{b}$$

$$U\boldsymbol{x} = \boldsymbol{y}$$

$$L\boldsymbol{y} = \boldsymbol{b}$$

```
        float y[N];
        for (int i = 0; i < N; ++i)
        {
                y[i] = matL[i][N] / matL[i][i];
        }

        // Input for the Lower triangular matrix from array of the
            computed result of LUdecomposition()
        float matU[M][M] = {0};
        for(int j=0; j<N; j++)
        {
                for(int k=0; k<N; k++)
                {
                        matU[j][k] = u[j][k];
                }
                matU[j][N] = y[j];
        }
```

```
 Matrix A
 2       6       2
-3      -8       0
 4       9       2

 Lower Triangular                Upper Triangular
 2       0       0               1       3       1
-3       1       0               0       1       3
 4      -3       7               0       0       1

Augmented Matrix L to solve Ly = b  :
 2       0       0       2
-3       1       0       2
 4      -3       7       3

Reduced Row Echelon for Augmented Matrix L is :
 2       0       0       2
 0       1       0       5
 0       0       7      14

The solution/s for Ly = b :
y(0) = 1,  y(1) = 5,  y(2) = 2,

Augmented Matrix U to solve Ux = y  :
 1       3       1       1
 0       1       3       5
 0       0       1       2
Reduced Row Echelon for Augmented Matrix U is :
 1       0       0       2
 0       1       0      -1
 0       0       1       2


The solution/s for Ux = y :
x(0) = 2,  x(1) = -1,  x(2) = 2,
```

**Figure 23.263:** *The computation to get the solution for $A\boldsymbol{x} = \boldsymbol{b}$ with LU-Decomposition (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/LU Decomposition/-main.cpp**).*

## XXII. The Power Method

[DF*] The eigenvalues of a square matrix can, in theory, be found by solving the characteristic equation. However, this procedure has so many computational difficulties that it is almost never used in applications.

[DF*] We will discuss an algorithm that can be used to approximate the eigenvalue with greatest absolute value and a corresponding eigenvector. This particular eigenvalue and its corresponding eigenvectors are important because they arise naturally in many iterative processes. The methods have been recently used to create Internet search engines such as Google.

[DF*] There are many applications in which some vectors $\boldsymbol{x}_0$ in $\mathbb{R}^n$ is multiplied repeatedly by an $n \times n$ matrix $A$ to produce a sequence

$$\boldsymbol{x}_0, \quad A\boldsymbol{x}_0, \quad A^2\boldsymbol{x}_0, \ldots, \quad A^k\boldsymbol{x}_0, \ldots$$

We call a sequence of this form a power sequence generated by $A$. We will be concerned with the convergence of power sequences and how such sequences can be used to approximate eigenvalues and eigenvectors.

**Definition 23.72: Dominant Eigenvalue and Eigenvector**

If the distinct eigenvalues of a matrix $A$ are $\lambda_1, \lambda_2, \ldots, \lambda_k$, and if $|\lambda_1|$ is larger than $|\lambda_2|, \ldots, |\lambda_k|$, then $\lambda_1$ is called a dominant eigenvalue of $A$.

Any eigenvector corresponding to a dominant eigenvalue is called a dominant eigenvector of $A$.

[DF*] If the distinct eigenvalues of a matrix are

$$\lambda_1 = -4, \quad \lambda_2 = -2, \quad \lambda_3 = 1, \quad \lambda_4 = 3$$

then $\lambda_1 = -4$ is dominant since $|\lambda_1| = 4$ is greater than the absolute value of all the other eigenvalues.

If the distinct eigenvalues of a matrix are

$$\lambda_1 = 8, \quad \lambda_2 = -8, \quad \lambda_3 = -2, \quad \lambda_4 = 5$$

then $|\lambda_1| = |\lambda_2| = 8$, so there is no eigenvalue whose absolute value is greater than the absolute value of all the other eigenvalues.

[DF*] The most important theorems about convergence of power sequences apply to $n \times n$ matrices with $n$ linearly independent eigenvectors (symmetric matrices)

**Theorem 23.161: Normalized Power Sequence**

Let $A$ be a symmetric $n \times n$ matrix with a positive dominant eigenvalue $\lambda$. If $\boldsymbol{x}_0$ is a unit vector in $\mathbb{R}^n$ that is not orthogonal to the eigenspace corresponding to $\lambda$, then the normalized power sequence

$$\boldsymbol{x}_0, \quad \boldsymbol{x}_1 = \frac{A\boldsymbol{x}_0}{||A\boldsymbol{x}_0||}, \quad \boldsymbol{x}_2 = \frac{A\boldsymbol{x}_1}{||A\boldsymbol{x}_1||}, \ldots, \quad , \boldsymbol{x}_k = \frac{A\boldsymbol{x}_{k-1}}{||A\boldsymbol{x}_{k-1}||}, \ldots \tag{23.327}$$

converges to a unit dominant eigenvector, and the sequence

$$A\boldsymbol{x}_1 \cdot \boldsymbol{x}_1, \quad A\boldsymbol{x}_2 \cdot \boldsymbol{x}_2, \quad A\boldsymbol{x}_3 \cdot \boldsymbol{x}_3, \ldots, \quad A\boldsymbol{x}_k \cdot \boldsymbol{x}_k, \ldots \tag{23.328}$$

converges to the dominant eigenvalue $\lambda$

[DF*] This form of the power sequence expresses each iterate in terms of the starting vector $\boldsymbol{x}_0$, rather than in terms of its predecessor.

$$\boldsymbol{x}_0, \quad \boldsymbol{x}_1 = \frac{A\boldsymbol{x}_0}{||A\boldsymbol{x}_0||}, \quad \boldsymbol{x}_2 = \frac{A^2\boldsymbol{x}_0}{||A^2\boldsymbol{x}_0||}, \ldots, \boldsymbol{x}_k = \frac{A^k\boldsymbol{x}_0}{||A^k\boldsymbol{x}_0||}, \ldots \tag{23.329}$$

[DF*] Let $A$ be a $2 \times 2$ symmetric matrix with distinct positive eigenvalues, $\lambda_1$ and $\lambda_2$, one of which is dominant. To be specific, assume that $\lambda_1$ is dominant and

$$\lambda_1 > \lambda_2 > 0$$

Since we are assuming that $A$ is symmetric and has distinct eigenvalues, it follows that the eigenspaces corresponding to $\lambda_1$ and $\lambda_2$ are perpendicular lines through the origin. Thus,

the asusmption that $\boldsymbol{x}_0$ is a unit vector that is not orthogonal to the eigenspace corresponding to $\lambda_1$ implies that $\boldsymbol{x}_0$ does not lie in the eigenspace corresponding to $\lambda_2$.

To see the geometric effect of multiplying $\boldsymbol{x}_0$ by $A$, it will be useful to split $\boldsymbol{x}_0$ into the sum

$$\boldsymbol{x}_0 = \boldsymbol{v}_0 + \boldsymbol{w}_0 \tag{23.330}$$

where $\boldsymbol{v}_0$ and $\boldsymbol{w}_0$ are the orthogonal projections of $\boldsymbol{x}_0$ on the eigenspaces of $\lambda_1$ and $\lambda_2$, respectively. This enables us to express $A\boldsymbol{x}_0$ as

**Figure 23.264:** *The illustration where $\boldsymbol{v}_0$ and $\boldsymbol{w}_0$ are the orthogonal projections of $\boldsymbol{x}_0$ on the eigenspaces of $\lambda_1$ and $\lambda_2$.*

$$A\boldsymbol{x}_0 = A\boldsymbol{v}_0 + A\boldsymbol{w}_0 = \lambda_1\boldsymbol{v}_0 + \lambda_2\boldsymbol{w}_0 \tag{23.331}$$

which tells us that multiplying $\boldsymbol{x}_0$ by $A$ "scales" the terms $\boldsymbol{v}_0$ and $\boldsymbol{w}_0$ by $\lambda_1$ and $\lambda_2$, respectively.

However, $\lambda_1$ is larger than $\lambda_2$, so the scaling is greater in the direction of $\boldsymbol{v}_0$ than in the direction of $\boldsymbol{w}_0$. Thus, multiplying $\boldsymbol{x}_0$ by $A$ "pulls" $\boldsymbol{x}_0$ toward the eigenspace of $\lambda_1$, and normalizing produces a vector $\boldsymbol{x}_1 = A\boldsymbol{x}_0/||A\boldsymbol{x}_0||$, which is on the unit circle and is closer to the eigenspace of $\lambda_1$ than $\boldsymbol{x}_0$. Similarly, multiplying $\boldsymbol{x}_1$ by $A$ and normalizing produces a unit vector $\boldsymbol{x}_2$ that is closer to the eigenspace of $\lambda_1$ than $\boldsymbol{x}_1$. Thus, it seems reasonable that by repeatedly multiplying by $A$ and normalizing we will produce a sequence of vectors $\boldsymbol{x}_k$ that lie on the unit circle and converge to a unit vector $\boldsymbol{x}$ in the eigenspace of $\lambda_1$. Moreover, if $\boldsymbol{x}_k$ converges to $\boldsymbol{x}$, then it also seems reasonable that $A\boldsymbol{x}_k \cdot \boldsymbol{x}_k$ will converge to

$$A\boldsymbol{x} \cdot \boldsymbol{x} = \lambda_1\boldsymbol{x} \cdot \boldsymbol{x} = \lambda_1||\boldsymbol{x}||^2 = \lambda_1$$

which is the dominant eigenvalue of $A$.

[DF*] **The Power Method with Euclidean Scaling Algorithm**

1. Choose an arbitrary nonzero vector and normalize it, if need to be, to obtain a unit vector $\boldsymbol{x}_0$.

**Figure 23.265:** *The illustration when we multiplying $\boldsymbol{x}_0$ by $A$, it will "pulls" $\boldsymbol{x}_0$ toward the eigenspace of $\lambda_1$, normalizing $A\boldsymbol{x}_0$ will produce a vector $\boldsymbol{x}_1 = A\boldsymbol{x}_0/||A\boldsymbol{x}_0||$.*

2. Compute $A\boldsymbol{x}_0$ and normalize it to obtain the first approximation $\boldsymbol{x}_1$ to a dominant unit eigenvector.
   Compute $A\boldsymbol{x}_1 \cdot \boldsymbol{x}_1$ to obtain the first approximation to the dominant eigenvalue.
3. Compute $A\boldsymbol{x}_1$ and normalize it to obtain the second approximation $\boldsymbol{x}_2$ to a dominant unit eigenvector.
   Compute $A\boldsymbol{x}_2 \cdot \boldsymbol{x}_2$ to obtain the second approximation to the dominant eigenvalue.
4. Compute $A\boldsymbol{x}_2$ and normalize it to obtain the third approximation $\boldsymbol{x}_3$ to a dominant unit eigenvector.
   Compute $A\boldsymbol{x}_3 \cdot \boldsymbol{x}_3$ to obtain the third approximation to the dominant eigenvalue.

Continuing in this way will usually generate a sequence of better and better approximations to the dominant eigenvalue and a corresponding unit eigenvector.

[DF*] There is a variation of the power method in which the iterates, rather than being normalized at each stage, are scaled to make the maximum entry 1. To describe this method, it will be convenient to denote the maximum absolute value of the entries in a vector $\boldsymbol{x}$ by $\max(\boldsymbol{x})$. Thus, for example, if

$$\boldsymbol{x} = \begin{bmatrix} 5 \\ 3 \\ -10 \\ 2 \end{bmatrix}$$

then $\max(\boldsymbol{x}) = 10$.

**Theorem 23.162: The Power Method with Maximum Entry Scaling**

Let $A$ be a symmetric $n \times n$ matrix with a positive dominant eigenvalue $\lambda$. If $\boldsymbol{x}_0$ is a nonzero vector in $\mathbb{R}^n$ that is not orthogonal to the eigenspace corresponding to $\lambda$, then the sequence

$$\boldsymbol{x}_0, \quad \boldsymbol{x}_1 = \frac{A\boldsymbol{x}_0}{\max(A\boldsymbol{x}_0)}, \quad \boldsymbol{x}_2 = \frac{A\boldsymbol{x}_1}{\max(A\boldsymbol{x}_1)}, \ldots, \quad \boldsymbol{x}_k = \frac{A\boldsymbol{x}_{k-1}}{\max(A\boldsymbol{x}_{k-1})}, \ldots \tag{23.332}$$

converges to an eigenvector corresponding to $\lambda$, and the sequence

$$\frac{A\boldsymbol{x}_1 \cdot \boldsymbol{x}_1}{\boldsymbol{x}_1 \cdot \boldsymbol{x}_1}, \quad \frac{A\boldsymbol{x}_2 \cdot \boldsymbol{x}_2}{\boldsymbol{x}_2 \cdot \boldsymbol{x}_2}, \quad \frac{A\boldsymbol{x}_3 \cdot \boldsymbol{x}_3}{\boldsymbol{x}_3 \cdot \boldsymbol{x}_3}, \ldots, \quad \frac{A\boldsymbol{x}_k \cdot \boldsymbol{x}_k}{\boldsymbol{x}_k \cdot \boldsymbol{x}_k}, \ldots \tag{23.333}$$

converges to $\lambda$.

[DF*] The form of

$$\frac{A\boldsymbol{x} \cdot \boldsymbol{x}}{\boldsymbol{x} \cdot \boldsymbol{x}} \tag{23.334}$$

which is called a Rayleigh quotient of $A$. In the case where $\lambda$ is an eigenvalue of $A$ and $\boldsymbol{x}$ is a corresponding eigenvector, the Rayleigh quotient is

$$\frac{A\boldsymbol{x} \cdot \boldsymbol{x}}{\boldsymbol{x} \cdot \boldsymbol{x}} = \frac{\lambda \boldsymbol{x} \cdot \boldsymbol{x}}{\boldsymbol{x} \cdot \boldsymbol{x}} = \frac{\lambda(\boldsymbol{x} \cdot \boldsymbol{x})}{\boldsymbol{x} \cdot \boldsymbol{x}} = \lambda$$

Thus, if $\boldsymbol{x}_k$ converges to a dominant eigenvector $\boldsymbol{x}$, then it seems reasonable that

$$\frac{A\boldsymbol{x}_k \cdot \boldsymbol{x}_k}{\boldsymbol{x}_k \cdot \boldsymbol{x}_k} \text{ converges to } \frac{A\boldsymbol{x} \cdot \boldsymbol{x}}{\boldsymbol{x} \cdot \boldsymbol{x}} = \lambda$$

which is the dominant eigenvalue.

[DF*] **The Power Method with Maximum Entry Scaling Algorithm**

1. Choose an arbitrary nonzero vector $\boldsymbol{x}_0$.
2. Compute $A\boldsymbol{x}_0$ and multiply it by the factor $1/\max(A\boldsymbol{x}_0)$ to obtain the first approximation $\boldsymbol{x}_1$ to a dominant eigenvector.
   Compute the Rayleigh quotient of $\boldsymbol{x}_1$ to obtain the first approximation to the dominant eigenvalue.
3. Compute $A\boldsymbol{x}_1$ and multiply it by the factor $1/\max(A\boldsymbol{x}_1)$ to obtain the second approximation $\boldsymbol{x}_2$ to a dominant eigenvector.
   Compute the Rayleigh quotient of $\boldsymbol{x}_2$ to obtain the second approximation to the dominant eigenvalue.
4. Compute $A\boldsymbol{x}_2$ and multiply it by the factor $1/\max(A\boldsymbol{x}_2)$ to obtain the third approximation $\boldsymbol{x}_3$ to a dominant eigenvector.
   Compute the Rayleigh quotient of $\boldsymbol{x}_3$ to obtain the third approximation to the dominant eigenvalue.

Continuing in this way will generate a sequence of better and better approximations to the dominant eigenvalue and a corresponding eigenvector.

[DF*] If $A$ is a symmetric matrix whose distinct eigenvalues can be arranged so that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_k|$$

then the "rate" at which the Rayleigh quotients converge to the dominant eigenvalue $\lambda_1$ depends on the ratio $|\lambda_1|/|\lambda_2|$; that is, the convergence is slow when this ratio is near 1 and rapid when it is large-the greater the ratio, the more rapid the convergence.

For example, if $A$ is a $2 \times 2$ symmetric matrix, then the greater the ratio $|\lambda_1|/|\lambda_2|$, the greater the disparity between the scaling effects of $\lambda_1$ and $\lambda_2$, and hence the greater the effect that multiplication by $A$ has on pulling the iterates toward the eigenspace of $\lambda_1$. In case where the ratio is close to 1, the convergence of the power method may be so slow that other methods must be used.

[DF*] **Stopping Procedures**

If $\lambda$ is the exact value of the dominant eigenvalue, and if a power method produces the approximation $\lambda^{(k)}$ at the $k$th iteration, then we call

$$\left|\frac{\lambda - \lambda^{(k)}}{\lambda}\right| \tag{23.335}$$

the relative error in $\lambda^{(k)}$. If this is expressed as a percentage, then it is called the percentage error in $\lambda^{(k)}$.

In applications one usually knows the relative error $E$ that can be tolerated in the dominant eigenvalue, so the goal is to stop computing iterates once the relative error in the approximation to that eigenvalue is less than $E$. However, there is a problem in computing the relative error from (23.335) in that the eigenvalue $\lambda$ is unknown. To circumvent this problem, it is usual to estimate $\lambda$ by $\lambda^{(k)}$ and stop the computations when

$$\left|\frac{\lambda^{(k)} - \lambda^{(k-1)}}{\lambda^{(k)}}\right| < E \tag{23.336}$$

The quantity on the left side of (23.336) is called the estimated relative error in $\lambda^{(k)}$ and its percentage form is called the estimated percentage error in $\lambda^{(k)}$.

[DF*] A rule for deciding when to stop an iterative process is called a stopping procedure.

## i. C++ Computation: The Power Method with Euclidean Scaling

Apply the power method with Euclidean scaling to

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$$

with

$$\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

compare the resulting approximations to the exact values of the dominant eigenvalue and eigenvector.

**Solution:**
The eigenvalues of $A$ are

$$\lambda_1 = 1$$
$$\lambda_2 = 5$$

and the eigenspace corresponding to the dominant eigenvalue $\lambda_2 = 5$ is the line represented by the parametric equations

$$(x_1, x_2) = (t, t) =$$

which we can write in vector form as

$$\boldsymbol{x} = t \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

after we normalize the vector above, we will obtain the dominant eigenvector

$$\boldsymbol{v}_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}$$

Now let us see what happens when we use the power method, starting with the unit vector $\boldsymbol{x}_0$.

$$A\boldsymbol{x}_0 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\boldsymbol{x}_1 = \frac{A\boldsymbol{x}_0}{||A\boldsymbol{x}_0||} = \frac{1}{\sqrt{13}} \begin{bmatrix} 3 \\ 2 \end{bmatrix} \approx \begin{bmatrix} 0.83205 \\ 0.55470 \end{bmatrix}$$

$$A\boldsymbol{x}_1 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0.83205 \\ 0.55470 \end{bmatrix} = \begin{bmatrix} 3.60555 \\ 3.32820 \end{bmatrix}$$

$$\boldsymbol{x}_2 = \frac{A\boldsymbol{x}_1}{||A\boldsymbol{x}_1||} = \frac{1}{4.90682} \begin{bmatrix} 3.60555 \\ 3.32820 \end{bmatrix} \approx \begin{bmatrix} 0.73480 \\ 0.67828 \end{bmatrix}$$

$$A\boldsymbol{x}_2 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0.73480 \\ 0.67828 \end{bmatrix} = \begin{bmatrix} 3.56097 \\ 3.50445 \end{bmatrix}$$

$$\boldsymbol{x}_3 = \frac{A\boldsymbol{x}_2}{||A\boldsymbol{x}_2||} = \frac{1}{4.99616} \begin{bmatrix} 3.56097 \\ 3.50445 \end{bmatrix} \approx \begin{bmatrix} 0.71274 \\ 0.70143 \end{bmatrix}$$

$$Ax_3 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0.71274 \\ 0.70143 \end{bmatrix} = \begin{bmatrix} 3.54108 \\ 3.52976 \end{bmatrix}$$

$$x_4 = \frac{Ax_3}{||Ax_3||} = \frac{1}{4.99985} \begin{bmatrix} 3.54108 \\ 3.52976 \end{bmatrix} \approx \begin{bmatrix} 0.70824 \\ 0.70597 \end{bmatrix}$$

$$Ax_4 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0.70824 \\ 0.70597 \end{bmatrix} = \begin{bmatrix} 3.53666 \\ 3.53440 \end{bmatrix}$$

$$x_5 = \frac{Ax_4}{||Ax_4||} = \frac{1}{4.99999} \begin{bmatrix} 3.53666 \\ 3.53440 \end{bmatrix} \approx \begin{bmatrix} 0.70733 \\ 0.70688 \end{bmatrix}$$

$$\lambda^{(1)} = (Ax_1) \cdot x_1 = (Ax_1)^T x_1 \approx \begin{bmatrix} 3.60555 & 3.32820 \end{bmatrix} \begin{bmatrix} 0.83205 \\ 0.55470 \end{bmatrix} \approx 4.84615$$

$$\lambda^{(2)} = (Ax_2) \cdot x_2 = (Ax_2)^T x_2 \approx \begin{bmatrix} 3.56097 & 3.50445 \end{bmatrix} \begin{bmatrix} 0.73480 \\ 0.67828 \end{bmatrix} \approx 4.99361$$

$$\lambda^{(3)} = (Ax_3) \cdot x_3 = (Ax_3)^T x_3 \approx \begin{bmatrix} 3.54108 & 3.52976 \end{bmatrix} \begin{bmatrix} 0.71274 \\ 0.70143 \end{bmatrix} \approx 4.99974$$

$$\lambda^{(4)} = (Ax_4) \cdot x_4 = (Ax_4)^T x_4 \approx \begin{bmatrix} 3.53666 & 3.53440 \end{bmatrix} \begin{bmatrix} 0.70824 \\ 0.70597 \end{bmatrix} \approx 4.99999$$

$$\lambda^{(5)} = (Ax_5) \cdot x_5 = (Ax_5)^T x_5 \approx \begin{bmatrix} 3.53576 & 3.53531 \end{bmatrix} \begin{bmatrix} 0.70733 \\ 0.70688 \end{bmatrix} \approx 5.00000$$

Thus, $\lambda^{(5)}$ approximates the dominant eigenvalue to five decimal place accuracy and $x_5$ approximates the dominant eigenvector correctly to three decimal place accuracy.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       mat x0;
       x0.load("vectorx0.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
```

```
cout <<"Matrix A:" << "\n" << A <<endl;
cout <<"Vector x_{0}:" << "\n" << x0 <<endl;

eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
    square matrix

cout << "Eigenvalues:" << endl;

int n = 2;
for (int i = 1; i <= n; i++)
{
       cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
           imag(eigval[i-1]) << "i" << endl;
}

cout << endl;

cout << "Eigenvectors:" << endl;
for (int i = 1; i <= n; i++)
{
       cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
           endl;
}
cout << "l_max / Maximum value of Eigenvalue:" << max(real(eigval))
    << endl;

cout << "\nThe Power Method with Euclidean Scaling Computation:" <<
    endl;
float err = 10;
vec x;
x = x0;
float l_approx;
mat Ax(2,2,fill::zeros);
mat Axnew(2,2,fill::zeros);
int N = 10;
int i = 1;

// With while loop
while (err > 10e-7)
{
       Ax = A*x;
       cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

       x = Ax/norm(Ax);
       cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

       Axnew = A*x;
       l_approx = cdot(Axnew,x);
```

```
                cout <<"l^("<< i <<") = " << l_approx <<endl;

                err = max(real(eigval)) - l_approx;
                cout <<"error : l_max - l^(" << i <<") = " << "" << err <<
                    endl;
                i = i+1;
        }
        /*
        // With for loop
        for (int i = 1; i <=N; i++)
        {
                Ax = A*x;
                cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

                x = Ax/norm(Ax);
                cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

                Axnew = A*x;
                l_approx = cdot(Axnew,x);
                cout <<"l^("<< i <<") = " << l_approx <<endl;

                err = max(real(eigval)) - l_approx;

                if (err <= 10e-7)
                {
                        break;
                }
        }
        */
        return 0;
}
```

**C++ Code 260:** *main.cpp "The power method with Euclidean scaling"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- We have created two options between **while** loop and **for** loop for the Numerical Power method computation, both will stop after the stop error criteria is reached.

```
        ...
        ...
```

```
float err = 10;
vec x;
x = x0;
float l_approx;
mat Ax(2,2,fill::zeros);
mat Axnew(2,2,fill::zeros);
int N = 10;
int i = 1;

// With while loop
while (err > 10e-7)
{
       Ax = A*x;
       cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

       x = Ax/norm(Ax);
       cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

       Axnew = A*x;
       l_approx = cdot(Axnew,x);
       cout <<"l^("<< i <<") = " << l_approx <<endl;

       err = max(real(eigval)) - l_approx;
       cout <<"error : l_max - l^(" << i <<") = " << "" << err
           <<endl;
       i = i+1;
}
/*
// With for loop
for (int i = 1; i <=N; i++)
{
       Ax = A*x;
       cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

       x = Ax/norm(Ax);
       cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

       Axnew = A*x;
       l_approx = cdot(Axnew,x);
       cout <<"l^("<< i <<") = " << l_approx <<endl;

       err = max(real(eigval)) - l_approx;

       if (err <= 10e-7)
       {
              break;
       }
}
```

```
            */
```

```
Matrix A:
   3.0000   2.0000
   2.0000   3.0000

Vector x_{0}:
   1.0000
        0

Eigenvalues:
λ_1 = 5 + 0i
λ_2 = 1 + 0i

Eigenvectors:
x[λ_1] :
   0.7071
   0.7071

x[λ_2] :
   -0.7071
   0.7071

λ_max / Maximum value of Eigenvalue:5
```

**Figure 23.266:** *The computation to approximate the dominant eigenvalue and dominant eigenvector of matrix A with the Power Method with Euclidean scaling (part 1) (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Power Method with Euclidean Scaling with Armadillo/main.cpp**).*

```
The Power Method with Euclidean Scaling Computation:

Ax_{0}:
   3.0000
   2.0000

x_{1}:
   0.8321
   0.5547

λ^(1) = 4.84615
error : λ_max - λ^(1) = 0.153846

Ax_{1}:
   3.6056
   3.3282

x_{2}:
   0.7348
   0.6783

λ^(2) = 4.99361
error : λ_max - λ^(2) = 0.00638962

Ax_{2}:
   3.5610
   3.5044

x_{3}:
   0.7127
   0.7014

λ^(3) = 4.99974
error : λ_max - λ^(3) = 0.000256062

Ax_{3}:
   3.5411
   3.5298

x_{4}:
   0.7082
   0.7060

λ^(4) = 4.99999
error : λ_max - λ^(4) = 1.00136e-05

Ax_{4}:
   3.5367
   3.5344

x_{5}:
   0.7073
   0.7069

λ^(5) = 5
error : λ_max - λ^(5) = 4.76837e-07
```

**Figure 23.267:** *The computation to approximate the dominant eigenvalue and dominant eigenvector of matrix A with the Power Method with Euclidean scaling (part 2)(**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Power Method with Euclidean Scaling with Armadillo/main.cpp**).*

## ii. C++ Computation: The Power Method with Maximum Entry Scaling

Apply the power method with Euclidean scaling to

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$$

with

$$\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

compare the resulting approximations to the exact values of the dominant eigenvalue and eigenvector.

**Solution:**

$$A\boldsymbol{x}_0 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\boldsymbol{x}_1 = \frac{A\boldsymbol{x}_0}{\max(A\boldsymbol{x}_0)} = \frac{1}{3} \begin{bmatrix} 3 \\ 2 \end{bmatrix} \approx \begin{bmatrix} 1.00000 \\ 0.66667 \end{bmatrix}$$

$$A\boldsymbol{x}_1 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1.00000 \\ 0.66667 \end{bmatrix} = \begin{bmatrix} 4.33333 \\ 4.00000 \end{bmatrix}$$

$$\boldsymbol{x}_2 = \frac{A\boldsymbol{x}_1}{\max(A\boldsymbol{x}_1)} = \frac{1}{4.33333} \begin{bmatrix} 4.33333 \\ 4.00000 \end{bmatrix} \approx \begin{bmatrix} 1.00000 \\ 0.92308 \end{bmatrix}$$

$$A\boldsymbol{x}_2 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1.00000 \\ 0.92308 \end{bmatrix} = \begin{bmatrix} 4.84615 \\ 4.76923 \end{bmatrix}$$

$$\boldsymbol{x}_3 = \frac{A\boldsymbol{x}_2}{\max(A\boldsymbol{x}_2)} = \frac{1}{4.84615} \begin{bmatrix} 4.84615 \\ 4.76923 \end{bmatrix} \approx \begin{bmatrix} 1.00000 \\ 0.98413 \end{bmatrix}$$

$$A\boldsymbol{x}_3 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1.00000 \\ 0.98413 \end{bmatrix} = \begin{bmatrix} 4.96825 \\ 4.95238 \end{bmatrix}$$

$$\boldsymbol{x}_4 = \frac{A\boldsymbol{x}_3}{\max(A\boldsymbol{x}_3)} = \frac{1}{4.96825} \begin{bmatrix} 4.96825 \\ 4.95238 \end{bmatrix} \approx \begin{bmatrix} 1.00000 \\ 0.99681 \end{bmatrix}$$

$$A\boldsymbol{x}_4 = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1.00000 \\ 0.99681 \end{bmatrix} = \begin{bmatrix} 4.99361 \\ 4.99042 \end{bmatrix}$$

$$\boldsymbol{x}_5 = \frac{A\boldsymbol{x}_4}{\max(A\boldsymbol{x}_4)} = \frac{1}{4.99361} \begin{bmatrix} 4.99361 \\ 4.99042 \end{bmatrix} \approx \begin{bmatrix} 1.00000 \\ 0.99936 \end{bmatrix}$$

$$\lambda^{(1)} = \frac{(A\boldsymbol{x}_1) \cdot \boldsymbol{x}_1}{\boldsymbol{x}_1 \cdot \boldsymbol{x}_1} = \frac{(A\boldsymbol{x}_1)^T \boldsymbol{x}_1}{\boldsymbol{x}_1^T \boldsymbol{x}_1} \approx \frac{7.00000}{1.44444} \approx 4.84615$$

$$\lambda^{(2)} = \frac{(A\boldsymbol{x}_2) \cdot \boldsymbol{x}_2}{\boldsymbol{x}_2 \cdot \boldsymbol{x}_2} = \frac{(A\boldsymbol{x}_2)^T \boldsymbol{x}_2}{\boldsymbol{x}_2^T \boldsymbol{x}_2} \approx \frac{9.24852}{1.85207} \approx 4.99361$$

$$\lambda^{(3)} = \frac{(Ax_3)\cdot x_3}{x_3\cdot x_3} = \frac{(Ax_3)^T x_3}{x_3^T x_3} \approx \frac{9.84203}{1.96851} \approx 4.99974$$

$$\lambda^{(4)} = \frac{(Ax_4)\cdot x_4}{x_4\cdot x_4} = \frac{(Ax_4)^T x_4}{x_4^T x_4} \approx \frac{9.96808}{1.99362} \approx 4.99999$$

$$\lambda^{(5)} = \frac{(Ax_5)\cdot x_5}{x_5\cdot x_5} = \frac{(Ax_5)^T x_5}{x_5^T x_5} \approx \frac{9.99360}{1.99872} \approx 5.00000$$

Thus, $\lambda^{(5)}$ approximates the dominant eigenvalue correctly to five decimal places and $x_5$ closely approximates the dominant eigenvector

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

. that results by taking $t = 1$. The rapid convergence is due to the fact that $|\lambda_1|/|\lambda_2| = 5/1 = 5$, which is considered to be a large ratio.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       mat x0;
       x0.load("vectorx0.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(2,2,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;
       cout <<"Vector x_{0}:" << "\n" << x0 <<endl;

       eig_gen(eigval, eigvec, A); // Eigen decomposition of dense general
           square matrix

       cout << "Eigenvalues:" << endl;

       int n = 2;
       for (int i = 1; i <= n; i++)
       {
              cout << "l_" << i << " = " <<real(eigval[i-1]) << " + " <<
                  imag(eigval[i-1]) << "i" << endl;
```

```
        }

        cout << endl;

        cout << "Eigenvectors:" << endl;
        for (int i = 1; i <= n; i++)
        {
               cout << "x[l_" << i << "] :\n " << real(eigvec.col(i-1)) <<
                   endl;
        }
        cout << "l_max / Maximum value of Eigenvalue:" << max(real(eigval))
            << endl;

        cout << "\nThe Power Method with Maximum Entry Scaling Computation:"
             << endl;
        float err = 10;
        vec x;
        x = x0;
        float l_approx;
        mat Ax(2,2,fill::zeros);
        mat Axnew(2,2,fill::zeros);
        int N = 10;
        int i = 1;

        /*
        // With while loop
        while (err > 10e-7)
        {
               Ax = A*x;
               cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

               x = Ax*(1/max(Ax));
               cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

               Axnew = A*x;
               l_approx = cdot(Axnew,x)/cdot(x,x);
               cout <<"l^("<< i <<") = " << l_approx <<endl;

               err = max(real(eigval)) - l_approx;
               cout <<"error : l_max - l^(" << i <<") = " << "" << err <<
                   endl;
               i = i+1;
        }
        */
        ///*
        // With for loop
        for (int i = 1; i <=N; i++)
        {
```

```
                    Ax = A*x;
                    cout <<"\nAx_{"<< i-1 <<"}:" << "\n" << Ax <<endl;

                    x = Ax*(1/max(Ax));
                    cout <<"x_{"<< i <<"}:" << "\n" << x <<endl;

                    Axnew = A*x;
                    l_approx = cdot(Axnew,x)/cdot(x,x);
                    cout <<"l^("<< i <<") = " << l_approx <<endl;

                    err = max(real(eigval)) - l_approx;
                    cout <<"error : l_max - l^(" << i <<") = " << "" << err <<
                        endl;

                    if (err <= 10e-7)
                    {
                           break;
                    }
            }
            //*/
            return 0;
    }
```

**C++ Code 261:** *main.cpp "The power method with maximum entry scaling"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

```
The Power Method with Maximum Entry Scaling Computation:

Ax_{0}:
   3.0000
   2.0000

x_{1}:
   1.0000
   0.6667

λ^(1) = 4.84615
error : λ_max - λ^(1) = 0.153846

Ax_{1}:
   4.3333
   4.0000

x_{2}:
   1.0000
   0.9231

λ^(2) = 4.99361
error : λ_max - λ^(2) = 0.00638962

Ax_{2}:
   4.8462
   4.7692

x_{3}:
   1.0000
   0.9841

λ^(3) = 4.99974
error : λ_max - λ^(3) = 0.000256062

Ax_{3}:
   4.9683
   4.9524

x_{4}:
   1.0000
   0.9968

λ^(4) = 4.99999
error : λ_max - λ^(4) = 1.00136e-05

Ax_{4}:
   4.9936
   4.9904

x_{5}:
   1.0000
   0.9994

λ^(5) = 5
error : λ_max - λ^(5) = 4.76837e-07
```

**Figure 23.268:** *The computation to approximate the dominant eigenvalue and dominant eigenvector of matrix A with the Power Method with Euclidean Maximum Entry scaling (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Power Method with Maximum Entry Scaling with Armadillo/main.cpp**).*

# XXIII. Internet Search Engines

[DF*] Early search engine on the Internet worked by examining key words and phrases in pages and titles of posted documents. Today's most popular search engines use algorithms based on the power method to analyze hyperlinks (references) between documents.

[DF*] Google, the most widely used engine for searching the Internet, was developed in 1996 by Larry Page and Sergey Brin while both were graduate students at Stanford University. Google used a procedure known as the PageRank algorithm to analyze how documents at relevant sites reference one another. It then assigns to each site a PageRank score, stores those scores as a matrix, and uses the components of the dominant eigenvector of that matrix to establish the relative importance of the sites to the search.

Google starts by using a standard text-based search engine to find an initial set $S_0$ of sites containing relevant pages. Since words can have multiple meanings, the set $S_0$ will typically contain irrelevant sites and miss others of relevance. To compensate for this, the set $S_0$ is expanded to a larger set $S$ by adjoining all sites referenced by the pages in the sites of $S_0$. The underlying assmption is that $S$ will contain the most important sites relevant to the search. This process is then repeated a number of times to refine the search information.

[DF*] To be more specific, suppose that the search set $S$ contain $n$ sites, and define the adjacency matrix for $S$ to be the $n \times n$ matrix $A = [a_{ij}]$ in which

$$a_{ij} = 1$$

if site $i$ references site $j$.

$$a_{ij} = 0$$

if site $i$ does not reference site $j$.

We will assume that no site reference itself, so the diagonal entries of $A$ will all be 0.

[DF*] There are two basic roles that a site can play in the search process-the site may be a hub, meaning that it references many other sites, or it may be an authority, meaning that it is referenced by many other sites. A given site will typically have both hub and authority properties in that it will both reference and be referenced.

[DF*] In general, if $A$ is an adjacency matrix for $n$ sites, then the column sums of $A$ measure the authority aspect of the sites and the row sums of $A$ measure their hub aspect.

Accordingly, if $A$ is an adjacency matrix, then we call the vector $\boldsymbol{h}_0$ of row sums of $A$ the initial hub vector of $A$, and we call the vector $\boldsymbol{a}_0$ of column sums of $A$ the initial authority vector of $A$. Alternatively, we can think of $\boldsymbol{a}_0$ as the vector of row sums of $A^T$, which turns out to be more convenient for computations. The entries in the hub vector are called hub weights and those in the authority vector authority weights.

[DF*] There is an interaction between hubs and authorities that needs to be accounted for in the search process. Accordingly, once the search engine has calculated the initial authority vector $\boldsymbol{a}_0$, it then uses the information in that vector to create new hub and authority vectors $\boldsymbol{h}_1$ and $\boldsymbol{a}_1$ using the formulas

$$\boldsymbol{h}_1 = \frac{A\boldsymbol{a}_0}{||A\boldsymbol{a}_0||} \tag{23.337}$$

and

$$\boldsymbol{a}_1 = \frac{A^T\boldsymbol{h}_1}{||A^T\boldsymbol{h}_1||} \tag{23.338}$$

The numerators in these formulas do the weighting, and the normalization serves to control the size of the entries. To understand how the numerators accomplish the wighting, view the product $A\boldsymbol{a}_0$ as a linear combination of the column vectors of $A$ with coefficients from $\boldsymbol{a}_0$.

Once the updated hub and authority vectors, $\boldsymbol{h}_1$ and $\boldsymbol{a}_1$, are obtained, the search engine repeats the process and computes a succession of hub and authority vectors, thereby generating the interrrelated sequences

$$\boldsymbol{h}_1 = \frac{A\boldsymbol{a}_0}{||A\boldsymbol{a}_0||}, \quad \boldsymbol{h}_2 = \frac{A\boldsymbol{a}_1}{||A\boldsymbol{a}_1||}, \quad \boldsymbol{h}_3 = \frac{A\boldsymbol{a}_2}{||A\boldsymbol{a}_2||}, \ldots, \quad \boldsymbol{h}_k = \frac{A\boldsymbol{a}_{k-1}}{||A\boldsymbol{a}_{k-1}||}, \ldots \tag{23.339}$$

$$\boldsymbol{a}_0, \quad \boldsymbol{a}_1 = \frac{A^T\boldsymbol{h}_1}{||A^T\boldsymbol{h}_1||}, \quad \boldsymbol{a}_2 = \frac{A^T\boldsymbol{h}_2}{||A^T\boldsymbol{h}_2||}, \quad \boldsymbol{a}_3 = \frac{A^T\boldsymbol{h}_3}{||A^T\boldsymbol{h}_3||}, \ldots, \quad \boldsymbol{a}_k = \frac{A^T\boldsymbol{h}_k}{||A^T\boldsymbol{h}_k||}, \ldots \tag{23.340}$$

However, each of these is a power sequence in disguise. For example, if we substitute the expression for $\boldsymbol{h}_k$ into the expression for $\boldsymbol{a}_k$, then we obtain

$$\boldsymbol{a}_k = \frac{A^T\boldsymbol{h}_k}{||A^T\boldsymbol{h}_k||} = \frac{A^T\left(\frac{A\boldsymbol{a}_{k-1}}{||A\boldsymbol{a}_{k-1}||}\right)}{\left|\left|A^T\left(\frac{A\boldsymbol{a}_{k-1}}{||A\boldsymbol{a}_{k-1}||}\right)\right|\right|} = \frac{(A^TA)\boldsymbol{a}_{k-1}}{||(A^TA)\boldsymbol{a}_{k-1}||}$$

which means we can rewrite (23.340) as

$$\boldsymbol{a}_0, \quad \boldsymbol{a}_1 = \frac{(A^TA)\boldsymbol{a}_0}{||(A^TA)\boldsymbol{a}_0||}, \quad \boldsymbol{a}_2 = \frac{(A^TA)\boldsymbol{a}_1}{||(A^TA)\boldsymbol{a}_1||}, \ldots, \quad \boldsymbol{a}_k = \frac{(A^TA)\boldsymbol{a}_{k-1}}{||(A^TA)\boldsymbol{a}_{k-1}||}, \ldots \tag{23.341}$$

Similarly, we can rewrite (23.339) as

$$\boldsymbol{h}_1 = \frac{A\boldsymbol{a}_0}{||A\boldsymbol{a}_0||}, \quad \boldsymbol{h}_2 = \frac{(AA^T)\boldsymbol{h}_1}{||(AA^T)\boldsymbol{h}_1||}, \ldots, \quad \boldsymbol{h}_k = \frac{(AA^T)\boldsymbol{h}_{k-1}}{||(AA^T)\boldsymbol{h}_{k-1}||}, \ldots \tag{23.342}$$

## i. C++ Computation: A Ranking Procedure for Search Engine

Suppose that a search engine produces 10 Internet sites in its search set and that the adjacency matrix for those sites is

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

rank the sites in decreasing order of authority.

**Solution:**

We will take $\boldsymbol{a}_0$ to be the normalized vector of column sums of $A$, and then we will compute the iterates in (23.341) until the authority vectors seem to stabilize.

$$\boldsymbol{a}_0 = \frac{1}{\sqrt{54}}\begin{bmatrix}0\\2\\1\\1\\5\\3\\1\\3\\0\\2\end{bmatrix} \approx \begin{bmatrix}0\\0.27217\\0.13608\\0.13608\\0.68041\\0.40825\\0.13608\\0.40825\\0\\0.27217\end{bmatrix}$$

and that

$$(A^TA)\boldsymbol{a}_0 = \begin{bmatrix}0&0&0&0&0&0&0&0&0&0\\0&2&1&1&2&0&0&2&0&1\\0&1&1&1&1&0&0&1&0&1\\0&1&1&1&1&0&0&1&0&1\\0&2&1&1&5&0&0&2&0&1\\0&0&0&0&0&3&1&0&0&0\\0&0&0&0&0&1&1&0&0&0\\0&2&1&1&2&0&0&3&0&1\\0&0&0&0&0&0&0&0&0&0\\0&1&1&1&1&0&0&1&0&2\end{bmatrix}\begin{bmatrix}0\\0.27217\\0.13608\\0.13608\\0.68041\\0.40825\\0.13608\\0.40825\\0\\0.27217\end{bmatrix} \approx \begin{bmatrix}0\\3.26599\\1.90516\\1.90516\\5.30723\\1.36083\\0.54433\\3.67423\\0\\2.17732\end{bmatrix}$$

Thus,

$$\boldsymbol{a}_1 = \frac{(A^TA)\boldsymbol{a}_0}{||(A^TA)\boldsymbol{a}_0||} \approx \frac{1}{8.15362}\begin{bmatrix}0\\3.26599\\1.90516\\1.90516\\5.30723\\1.36083\\0.54433\\3.67423\\0\\2.17732\end{bmatrix}$$

Continuing in this way yields the following authority iterates:

$$\boldsymbol{a}_0, \quad \boldsymbol{a}_1 = \frac{(A^TA)\boldsymbol{a}_0}{||(A^TA)\boldsymbol{a}_0||}, \quad \boldsymbol{a}_2 = \frac{(A^TA)\boldsymbol{a}_1}{||(A^TA)\boldsymbol{a}_1||}, \ldots, \boldsymbol{a}_9 = \frac{(A^TA)\boldsymbol{a}_8}{||(A^TA)\boldsymbol{a}_8||}, \quad \boldsymbol{a}_{10} = \frac{(A^TA)\boldsymbol{a}_9}{||(A^TA)\boldsymbol{a}_9||}$$

The small changes between $\boldsymbol{a}_9$ and $\boldsymbol{a}_{10}$ suggest that the iterates have stabilized near a dominant eigenvector of $A^TA$. From the entries in $\boldsymbol{a}_{10}$ we conclude that Sites 1,6,7, and 9 are probably irrelevant to the search and that the remaining sites should be searched in order of decreasing importance as

Site 5, Site 8, Site 2, Site 10, Sites 3 and 4 (tie).

The C++ code is using **Armadillo** library only. Basically, the computation is easier than the Power Method. We also include the code to compute the hub vectors up to $\boldsymbol{h}_{10}$.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat I(10,10,fill::eye);
       cout <<"Matrix A:" << "\n" << A <<endl;

       vec colsumA, rowsumA, a0, h0;
       colsumA = sum(A.t(),1);
       rowsumA = sum(A,1);

       a0 = colsumA/norm(colsumA);
       h0 = rowsumA/norm(rowsumA);
       cout << "Column sums of A:\n" << colsumA << endl;
       cout << "Row sums of A:\n" << rowsumA << endl;
       cout << "a_{0}:\n" << a0 << endl;
       cout << "h_{0}:\n" << h0 << endl;

       cout << "\nThe Internet Search Engines Computation:" << endl;
       mat ai_num, hi_num;
       vec ai, hi;
       hi = h0;
       ai = a0;
       int N = 10;

       // With for loop for a_{i}
       for (int i = 1; i <=N; i++)
       {
              ai_num = A.t()*A*ai;
              ai = ai_num/norm(ai_num);

              cout <<"\na_{"<< i <<"}:" << "\n" << ai <<endl;
       }
```

```
            // With for loop for h_{i}
            for (int i = 1; i <=N; i++)
            {
                  hi_num = A*A.t()*hi;
                  hi = hi_num/norm(hi_num);

                  cout <<"\nh_{"<< i-1 <<"}:" << "\n" << hi <<endl;

            }
            return 0;
      }
```

**C++ Code 262:** *main.cpp "A ranking procedure for search engine"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In **Armadillo** we can compute the column sum of a matrix with **sum(A.t(),1)**, since the original function is designed for row sum, we can obtain the column sum by transposing the matrix first.

```
            ...

            colsumA = sum(A.t(),1);
            rowsumA = sum(A,1);

            ...
```

- These are the codes to compute the sequence for the authority vectors, $\boldsymbol{a}_i$, and the sequence for the hub vectors $\boldsymbol{h}_i$ with $i = 0, 1, 2, \ldots, N$. With this formula, the value for $\boldsymbol{a}_i$ can be obtained from $\boldsymbol{a}_{i-1}$ and the value for $\boldsymbol{h}_i$ can be obtained from $\boldsymbol{h}_{i-1}$.

```
            ...

            a0 = colsumA/norm(colsumA);
            h0 = rowsumA/norm(rowsumA);

            ...
            ...

            mat ai_num, hi_num;
            vec ai, hi;
            hi = h0;
            ai = a0;
```

```
int N = 10;

...

for (int i = 1; i <=N; i++)
{
       ai_num = A.t()*A*ai;
       ai = ai_num/norm(ai_num);

       cout <<"\na_{"<< i <<"}:" << "\n" << ai <<endl;
}

// With for loop for h_{i}
for (int i = 1; i <=N; i++)
{
       hi_num = A*A.t()*hi;
       hi = hi_num/norm(hi_num);

       cout <<"\nh_{"<< i-1 <<"}:" << "\n" << hi <<endl;

}

...
```

```
Matrix A:
        0   1.0000        0        0   1.0000        0        0   1.0000        0        0
        0        0        0        0   1.0000        0        0        0        0        0
        0        0        0        0   1.0000        0        0        0        0        0
        0        0        0        0        0   1.0000   1.0000        0        0        0
        0        0        0        0        0        0        0   1.0000        0        0
        0   1.0000   1.0000   1.0000   1.0000        0        0   1.0000        0   1.0000
        0        0        0        0        0        0        0        0        0   1.0000
        0        0        0        0   1.0000        0        0        0        0        0
        0        0        0        0        0   1.0000        0        0        0        0
        0        0        0        0        0   1.0000        0        0        0        0

Column sums of A:
        0
   2.0000
   1.0000
   1.0000
   5.0000
   3.0000
   1.0000
   3.0000
        0
   2.0000

Row sums of A:
   3.0000
   1.0000
   1.0000
   2.0000
   1.0000
   6.0000
   1.0000
   1.0000
   1.0000
   1.0000
```

**Figure 23.269:** *The computation to do a ranking procedure for 10 Internet sites with PageRank' Internet Search Engines algorithm (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Ranking Procedure for Internet Search Engines with Armadillo/-main.cpp**).*

```
a_{0}:
         0
    0.2722
    0.1361
    0.1361
    0.6804
    0.4082
    0.1361
    0.4082
         0
    0.2722

h_{0}:
    0.4009
    0.1336
    0.1336
    0.2673
    0.1336
    0.8018
    0.1336
    0.1336
    0.1336
    0.1336


The Internet Search Engines Computation:

a_{1}:
         0
    0.4006
    0.2337
    0.2337
    0.6509
    0.1669
    0.0668
    0.4506
         0
    0.2670
```

**Figure 23.270:** *The computation to do a ranking procedure for 10 Internet sites with PageRank' Internet Search Engines algorithm (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Ranking Procedure for Internet Search Engines with Armadillo/-main.cpp**).*

```
a_{2}:
         0
    0.4165
    0.2492
    0.2492
    0.6341
    0.0632
    0.0260
    0.4667
         0
    0.2789


a_{3}:
         0
    0.4192
    0.2523
    0.2523
    0.6284
    0.0237
    0.0098
    0.4705
         0
    0.2830


a_{4}:
         0
    0.4197
    0.2531
    0.2531
    0.6266
    0.0089
    0.0037
    0.4714
         0
    0.2842
```

**Figure 23.271:** *The computation to do a ranking procedure for 10 Internet sites with PageRank' Internet Search Engines algorithm (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Ranking Procedure for Internet Search Engines with Armadillo/-main.cpp**).*

```
a_{5}:
         0
    0.4199
    0.2533
    0.2533
    0.6262
    0.0033
    0.0014
    0.4716
         0
    0.2845


a_{6}:
         0
    0.4199
    0.2533
    0.2533
    0.6260
    0.0012
    0.0005
    0.4716
         0
    0.2846


a_{7}:
         0
    0.4199
    0.2534
    0.2534
    0.6260
    0.0005
    0.0002
    0.4716
         0
    0.2846
```

**Figure 23.272:** *The computation to do a ranking procedure for 10 Internet sites with PageRank' Internet Search Engines algorithm (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Ranking Procedure for Internet Search Engines with Armadillo/-main.cpp**).*

```
a_{8}:
            0
   4.1990e-01
   2.5337e-01
   2.5337e-01
   6.2598e-01
   1.7515e-04
   7.2549e-05
   4.7165e-01
            0
   2.8460e-01


a_{9}:
            0
   4.1990e-01
   2.5337e-01
   2.5337e-01
   6.2597e-01
   6.5622e-05
   2.7182e-05
   4.7165e-01
            0
   2.8460e-01


a_{10}:
            0
   4.1990e-01
   2.5337e-01
   2.5337e-01
   6.2597e-01
   2.4586e-05
   1.0184e-05
   4.7165e-01
            0
   2.8460e-01
```

**Figure 23.273:** *The computation to do a ranking procedure for 10 Internet sites with PageRank' Internet Search Engines algorithm (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute the Ranking Procedure for Internet Search Engines with Armadillo/-main.cpp**).*

# XXIV. Singular Value Decomposition

[DF*] This is an extension of the diagonalization theory for $n \times n$ symmetric matrices to general $m \times n$ matrices.

The results have applications to compression, storage, and transmission of digitized information and form the basis for many of the best computational algorithms that are currently available for solving linear systems.

[DF*] Every symmetric matrix $A$ can be expressed as

$$A = PDP^T \tag{23.343}$$

where $P$ is an $n \times n$ orthogonal matrix of eigenvectors of $A$, and $D$ is the diagonal matrix whose diagonal entries are the eigenvalues corresponding to the column vectors of $P$. We will call () an eigenvalue decomposition of $A$.

[DF*] If an $n \times n$ matrix $A$ is not symmetric, then it does not have an eigenvalue decomposition, but it does have a Hessenberg decomposition

$$A = PHP^T$$

in which $P$ is an orthogonal matrix and $H$ is in upper Hessenberg form. Moreover, if $A$ has real eigenvalues, then it has a Schur decomposition

$$A = PSP^T$$

in which $P$ is an orthogonal matrix and $S$ is upper triangular.

[DF*] The eigenvalue, Hessenberg, and Schur decompositions are important in numerical algorithms not only because the matrices $D, H$, and $S$ have simpler forms than $A$, but also because the orthogonal matrices that appear in these factorizations do not magnify roundoff error. To see why this is so, suppose $\hat{\boldsymbol{x}}$ is a column vector whose entries are known exactly and that

$$\boldsymbol{x} = \hat{\boldsymbol{x}} + \boldsymbol{e}$$

is the vector that results when roundoff error is present in the entries of $\hat{\boldsymbol{x}}$. If $P$ is an orthogonal matrix, then the length-preserving property of orthogonal transformations implies that

$$||P\boldsymbol{x} - P\hat{\boldsymbol{x}} = ||\boldsymbol{x} - \hat{\boldsymbol{x}}|| = ||\boldsymbol{e}||$$

which tells us that the error in approximating $P\hat{\boldsymbol{x}}$ by $P\boldsymbol{x}$ has the same magnitude as the error in approximating $\hat{\boldsymbol{x}}$ by $\boldsymbol{x}$.

[DF*] There are two main paths that one might follow in looking for other kinds of decompositions of a general square matrix $A$: One might look for decompositions of the form

$$A = PJP^{-1}$$

in which $P$ is invertible but not necessarily orthogonal, or one might look for decompositions of the form

$$A = U \sum V^T$$

in which $U$ and $V$ are orthogonal but not necessarily the same. The first path leads to decompositions in which $J$ is either diagonal or a certain kind of block diagonal matrix, called a Jordan canonical form in honor of the French mathematician Camille Jordan. Jordan

canonical forms are important theoretically and in certain applications, but they are of lesser importance numerically because of the roundoff difficulties that result from the lack of orthogonality in $P$.

**Theorem 23.163: Matrices $A$ and $A^TA$**

If $A$ is an $m \times n$ matrix, then:

(a) $A$ and $A^TA$ have the same null space.

(b) $A$ and $A^TA$ have the same row space.

(c) $A^T$ and $A^TA$ have the same column space.

(d) $A$ and $A^TA$ have the same rank.

**Definition 23.73: Singular Value of $A$**

If $A$ is an $m \times n$ matrix, and if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $A^TA$, then the numbers

$$\sigma_1 = \sqrt{\lambda_1}, \quad \sigma_2 = \sqrt{\lambda_2}, \ldots \quad \sigma_n = \sqrt{\lambda_n}$$

are called the singular values of $A$.

**Example:**
Find the singular values of the matrix

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

**Solution:**
The first step is to find the eigenvalues of the matrix

$$A^TA = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The characteristic polynomial of $A^TA$ is

$$\lambda^2 - 4\lambda + 3 = (\lambda - 3)(\lambda - 1)$$

so the eigenvalues of $A^TA$ are

$$\lambda_1 = 3$$
$$\lambda_2 = 1$$

and the singular values of $A$ in order of decreasing size are

$$\sigma_1 = \sqrt{\lambda_1} = \sqrt{3}$$
$$\sigma_2 = \sqrt{\lambda_2} = 1$$

[DF*] We will find it useful to extend the notion of a "main diagonal" to matrices that are not square. We define the main diagonal of an $m \times n$ matrix to be the line of entries that starts at the upper left corner and extends diagonally as far as it can go. We will refer to the entries on the main diagonal as the diagonal entries.

[DF*] Factorization of a general $m \times n$ matrix $A$ is called singular value decomposition (abbreviated SVD).

**Theorem 23.164: Singular Value Decomposition**

If $A$ is an $m \times n$ matrix, then $A$ can be expressed in the form

$$A = U \sum V^T$$

where $U$ and $V$ are orthogonal matrices and $\sum$ is an $m \times n$ matrix whose diagonal entries are the singular values of $A$ and whose other entries are zero.

**Theorem 23.165: Singular Value Decomposition (Expanded Form)**

If $A$ is an $m \times n$ matrix of rank $k$, then $A$ can be factored as

$$A = U \sum V^T = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \dots & \boldsymbol{u}_k & | & \boldsymbol{u}_{k+1} & \dots & \boldsymbol{u}_m \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \dots & 0 & | \\ 0 & \sigma_2 & \dots & 0 & | \\ \vdots & \vdots & \ddots & \vdots & | \\ 0 & 0 & \dots & \sigma_k & | \\ & & & & \end{bmatrix}$$

in which $U, \sum$, and $V$ have sizes $m \times m$, $m \times n$, and $n \times n$, respectively, and in which

(a) $V = [\boldsymbol{v}_1 \ \boldsymbol{v}_2 \ \dots \ \boldsymbol{v}_n]$ orthogonally diagonalizes $A^T A$.

(b) The nonzero diagonal entries of $\sum$ are $\sigma_1 = \sqrt{\lambda_1}$, $\sigma_2 = \sqrt{\lambda_2}, \dots, \sigma_k = \sqrt{\lambda_k}$, where $\lambda_1, \lambda_2, \dots, \lambda_k$ are the nonzero eigenvalues of $A^T A$ corresponding to the column vectors of $V$.

(c) The column vector of $V$ are ordered so that $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_k > 0$.

(d) $\boldsymbol{u}_i = \frac{A\boldsymbol{v}_i}{||A\boldsymbol{v}_i||} = \frac{1}{\sigma_i} A\boldsymbol{v}_i \quad (i = 1, 2, \dots, k)$

(e) $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_k\}$ is an orthonormal basis for $\text{col}(A)$.

(f) $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_k, \boldsymbol{u}_{k+1}, \dots, \boldsymbol{u}_m\}$ is an extension of $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_k\}$ to an orthonormal basis for $\mathbb{R}^m$.

## i. C++ Computation: Singular Value Decomposition

Find a singluar value decomposition of the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

**Solution:**
We will have

$$A^T A = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The characteristic polynomial of $A^T A$ is

$$\lambda^2 - 4\lambda + 3 = (\lambda - 3)(\lambda - 1)$$

so the eigenvalues of $A^T A$ are

$$\lambda_1 = 3$$
$$\lambda_2 = 1$$

and the corresponding singular values of $A$ in order of decreasing size are

$$\sigma_1 = \sqrt{\lambda_1} = \sqrt{3}$$
$$\sigma_2 = \sqrt{\lambda_2} = 1$$

the eigenvectors corresponding to $\lambda_1$ and $\lambda_2$ are

$$\boldsymbol{v}_1 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}$$

we know that $V = [\boldsymbol{v}_1 | \boldsymbol{v}_2]$ orthogonally diagonalizes $A^T A$. Thus, the vectors

$$\boldsymbol{u}_1 = \frac{1}{\sigma_1} A\boldsymbol{v}_1 = \frac{\sqrt{3}}{3} \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{6}}{3} \\ \frac{\sqrt{6}}{6} \\ \frac{\sqrt{6}}{6} \end{bmatrix}$$

$$\boldsymbol{u}_2 = \frac{1}{\sigma_1} A\boldsymbol{v}_2 = (1) \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$

are two of the three column vectors of $U$. Note that $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are orthonormal, as expected.

We could extend the set $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ to an orthonormal basis for $\mathbb{R}^3$. However, the computations will be easier if we first remove the messy radicals by multiplying $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ by appropriate scalars. Thus, we will look for a unit vector $\boldsymbol{u}_3$ that is orthogonal to

$$\sqrt{6}\boldsymbol{u}_1 = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}, \quad \sqrt{2}\boldsymbol{u}_2 = \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}$$

to satisfy these two orthogonality conditions, the vector $\boldsymbol{u}_3$ must be a solution of the homogeneous linear system

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

the general solution of this system is

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = t \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$$

Normalizing the vector on the right yields

$$\boldsymbol{u}_3 = \begin{bmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$$

Thus, the singular value decomposition of $A$ is

$$A = U \sum V^T$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{6}}{3} & 0 & -\frac{1}{\sqrt{3}} \\ \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}}{2} & \frac{1}{\sqrt{3}} \\ \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{bmatrix}$$

The C++ code is using **Armadillo** library, this library has a function to compute the SVD of any non-square matrix.

```
#include <iostream>
#include <fstream>
#include <bits/stdc++.h> //for setw(6) at display() function
#include <armadillo>

using namespace std;
using namespace arma;

// Driver code
int main(int argc, char** argv)
{
       mat A;
       A.load("matrixA.txt");
       cx_mat eigvec;
       cx_vec eigval;

       mat AtA = (A.t())*A;
       cout <<"Matrix A:" << "\n" << A <<endl;
       cout <<"A^{T} * A:" << "\n" << AtA <<endl;
       eig_gen(eigval, eigvec, AtA); // Eigen decomposition of dense
           general square matrix
```

```
int n = AtA.n_cols; // number of dimension for A^{T}A
for (int i = 1; i <= n; i++)
{
       cout << "\nl_" << i << " = " <<real(eigval[i-1]) << endl;
       cout << "sigma_" << i << " = " <<sqrt(real(eigval[i-1])) <<
           endl;
}
for (int i = 1; i <= n; i++)
{
       cout << "\nv_" << i << " = \n" << real(eigvec.col(i-1)) <<
           endl;
}
for (int i = 1; i <= n; i++)
{
       cout << "\nu_" << i << " = \n" <<(1/real(eigval[i-1]))*A*
           real(eigvec.col(i-1)) << endl;
}

vec u1 = (1/sqrt(real(eigval[0])))*A*real(eigvec.col(0));
vec u2 = (1/sqrt(real(eigval[1])))*A*real(eigvec.col(1));

int N = u1.n_rows;

cout <<"Singular Value Decomposition of matrix A with Armadillo\n"
    <<endl;
mat U;
vec s;
mat V;

svd(U,s,V,A);

cout <<"U:" << "\n" << U <<endl;
cout <<"s:" << "\n" << s <<endl;
cout <<"V:" << "\n" << V <<endl;
cout <<"V^{T}:" << "\n" << V.t() <<endl;

mat Z(N,n,fill::zeros);
Z(0) = s(0);
Z(0+N+1) = s(1);
cout <<"Z:" << "\n" << Z <<endl;

mat A_SVD = U*Z*V.t();
A_SVD(0) = lround(A_SVD(0));
A_SVD(1) = lround(A_SVD(1));
A_SVD(2) = lround(A_SVD(2));
A_SVD(0+N) = lround(A_SVD(0+N));
A_SVD(1+N) = lround(A_SVD(1+N));
```

```
        A_SVD(2+N) = lround(A_SVD(2+N));

        cout <<"U * Z * V^{T}:" << "\n" << A_SVD <<endl;

        return 0;
    }
```

**C++ Code 263:** *main.cpp "Singular value decomposition"*

To compile it, type:
**g++ -o main main.cpp -larmadillo**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In **Armadillo** if we want to compute the SVD of matrix $A$, we can just use the funciton **svd(U,s,V,A)**, the matrix $\sum$ then can be created from vector $s$, the vector that represent the singular values of $A$.

```
        ...

        mat U;
        vec s;
        mat V;

        svd(U,s,V,A);

        ...
```

```
Matrix A:
   1.0000   1.0000
        0   1.0000
   1.0000        0

A^{T} * A:
   2.0000   1.0000
   1.0000   2.0000


λ_1 = 3
sigma_1 = 1.73205

λ_2 = 1
sigma_2 = 1

v_1 =
   0.7071
   0.7071


v_2 =
  -0.7071
   0.7071


u_1 =
   0.4714
   0.2357
   0.2357


u_2 =
        0
   0.7071
  -0.7071
```

**Figure 23.274:** *The computation to compute singular value decomposition of matrix A (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Singular Value Decomposition with Armadillo/main.cpp**).*

```
Singular Value Decomposition of matrix A with Armadillo

U:
  -8.1650e-01  -1.6653e-16  -5.7735e-01
  -4.0825e-01  -7.0711e-01   5.7735e-01
  -4.0825e-01   7.0711e-01   5.7735e-01

s:
   1.7321
   1.0000

V:
  -0.7071   0.7071
  -0.7071  -0.7071

V^{T}:
  -0.7071  -0.7071
   0.7071  -0.7071

Z:
   1.7321        0
        0   1.0000
        0        0

U * Z * V^{T}:
   1.0000   1.0000
        0   1.0000
   1.0000        0
```

**Figure 23.275:** *The computation to compute singular value decomposition of matrix A (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/Compute Singular Value Decomposition with Armadillo/main.cpp**).*

# Chapter 24

# DFSimulatorC++ XVIII: Linear and Nonlinear Programming

*"My only place is by your side. If you want to carry out your wish, I will follow you. That is my Nature." -* ***Sarah (Suikoden III)***

Linear programming is an optimization problem, a problem that is well rooted as a principle underlying the analysis of many complex decision or allocation problems. For example, if we have limited resources and we want to achieve a certain goal, we need to learn to optimize our resources with either linear programming or nonlinear programming depends on our problems at hand. Like how we decide to buy groceries depending on our needs / diet (the goal), and the money we have (the constraints).

Optimization, also known as mathematical programming, collection of mathematical principles and methods used for solving quantitative problems in many disciplines, including physics, biology, engineering, economics, and business. The subject grew from a realization that quantitative problems in manifestly different disciplines have important mathematical elements in common. Because of this commonality, many problems can be formulated and solved by using the unified set of ideas and methods that make up the field of optimization.

Optimization problems typically have three fundamental elements. The first is a single numerical quantity, or objective function, that is to be maximized or minimized. The objective may be the expected return on a stock portfolio, a company's production costs or profits, the time of arrival of a vehicle at a specified destination, or the vote share of a political candidate. The second element is a collection of variables, which are quantities whose values can be manipulated in order to optimize the objective. Examples include the quantities of stock to be bought or sold, the amounts of various resources to be allocated to different production activities, the route to be followed by a vehicle through a traffic network, or the policies to be advocated by a candidate. The third element of an optimization problem is a set of constraints, which are restrictions on the values that the variables can take. For instance, a manufacturing process cannot require more resources than are available, nor can it employ less than zero resources. Within this broad framework, optimization problems can have different mathematical properties. Problems in which the variables are continuous quantities (as in the resource allocation example) require a different approach from problems in which the variables are discrete or combinatorial quantities (as in the selection of a vehicle route from among a predefined set of possibilities).

# I. Linear Programming

[DF*]

## i. C++ Computation:

**C++ Code 264:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# II. Nonlinear Programming

[DF*]

## i. C++ Computation:

**C++ Code 265:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 25

# DFSimulatorC++ XIX: Numerical Methods

*"In this parallel world, battles are fought between dharma and chaos. The balance of power brings justice. Human have needs in their world, but the world does not need them." - **Luc (Suikoden III)***

When we have movement and we record the velocity, we realize that Gnuplot alone able to plot the graph with numerical data of the velocity, position, angle or even the kinetic energy. Plotting of the numeric data toward the time value from 0 to $n$, easy, you just need to enter 1 line in gnuplot after you print the output from C++, but now the question comes after we made the Force and Motion chapter, concerning about acceleration, Box2D has no **getAcceleration()** function yet, we use certain technique in Gnuplot to use past data to find the difference between current velocity and past velocity to get the acceleration. Thus, it could be easier if we know the data of a moving object along with its time, and it is having a pattern in its movement, we can cut that sample of time of all the time space population, thus make an approximation to get the function that can represent the numerical data in symbolic function.

Basically, numerical analysis is the study of algorithms that use numerical approximation (as opposed to symbolic manipulations) for the problems of mathematical analysis (as distinguished from discrete mathematics). It is the study of numerical methods that attempt at finding approximate solutions of problems rather than the exact ones. Numerical analysis finds application in all fields of engineering and the physical sciences, and in the 21st century also the life and social sciences, medicine, business and even the arts. Current growth in computing power has enabled the use of more complex numerical analysis, providing detailed and realistic mathematical models in science and engineering. Examples of numerical analysis include: ordinary differential equations as found in celestial mechanics (predicting the motions of planets, stars and galaxies), numerical linear algebra in data analysis, and stochastic differential equations and Markov chains for simulating living cells in medicine and biology.

Before modern computers, numerical methods often relied on hand interpolation formulas, using data from large printed tables. Since the mid 20th century, computers calculate the required functions instead, but many of the same formulas continue to be used in software algorithms.

Another one is when I read a book about Numerical Methods for Chemical Engineering, the balance of chemical reaction needs to be maintained, how to calculate how many substance needed can use the algorithm to find the solution of system of linear equations or system of differential

equations, thus numerical methods is a must have tools in that field of engineering. You must probably have known this one very famous package from Python language: **SymPy**. When using SymPy it is very easy to calculate the derivative or integral for all kinds of equations, univariate or even multivariate. In C++, we have "SymbolicC++", "GiNaC", "Armadillo", and "Boost" that have been explained in Chapter 6 to do Linear Algebra , Integral and Differentiation computation, we can mix these libraries to perform all the computations that SymPy can do.

## I. Mathematical Preliminaries

To refresh memory, and since Sentinel saw me from far away and want to teach me how to learn science the right way, telepathy and teleconnection are really something. People today is like stone age flint, they watched Youtube, TV, think it is amazing, Nature and Goddesses watched me from afar, from Valhalla, from Valhalla Projection, from Puncak Bintang, from L'Aquila, from Mount Fuji in Japan, etc. They don't need antenna and TV, they already advanced and powerful. Sometimes, no need to speak or write she(Freya) already reads my mind and told me through telepathy, "scroll up again the Riemann hypothesis you are searching for is on the previous page." I am pass the stage of surprised and saying "You are real!" it is thousands of times she helps me from GFreya OS, Arduino Memo, Lasthrim Projection, and now this, next? Self-made Sports Car for her birthday one day. Some intermezzo before we are going to the mathematical part.

To comprehend how we will be able to build C++ codes for Numerical Methods chapter, we will need to know some of this definitions and theorems:

**Definition 25.1: Limit**

A function $f$ on a set $X$ of real numbers has the limit $L$ at $x_0$, thus

$$\lim_{x \to x_0} f(x) = L \tag{25.1}$$

if given any number $\epsilon > 0$, there exists a real number $\delta > 0$ such that

$$|f(x) - L| < \epsilon$$

whenever $x \in X$ and

$$0 < |x - x_0| < \delta$$

**Definition 25.2: Continuity**

Let $f$ be a function defined on a set $X$ of real numbers and $x_0 \in X$. Then $f$ is continuous at $x_0$ if

$$\lim_{x \to x_0} f(x) = f(x_0) \tag{25.2}$$

The function $f$ is continuous on the set $X$ if it is continuous at each number in $X$.

[DF*] $C(X)$ denotes the set of all functions that are continuous on $X$. When $X$ is an interval of the real line, the parentheses in this notation are omitted. The set of all functions continuous on the closed interval $[a, b]$ is denoted by $C[a, b]$.

**Definition 25.3: Limit of a Sequence**

Let $\{x_n\}_{n=1}^{\infty}$ be an infinite sequence of real or complex numbers. The sequence $\{x_n\}_{n=1}^{\infty}$ has the limit $x$ (converges to $x$) if, for any $\epsilon > 0$, there exists a positive integer $N(\epsilon)$ such that

$$|x_n - x| < \epsilon$$

whenever $n > N(\epsilon)$. The notation

$$\lim_{n\to\infty} x_n = x \tag{25.3}$$

or

$$x_n \to x \quad \text{as} \quad n \to \infty \tag{25.4}$$

means that the sequence $\{x_n\}_{n=1}^{\infty}$ converges to $x$.

**Theorem 25.1: Equivalent Statements**

If $f$ is a function defined on a set $X$ of real numbers and $x_0 \in X$, then the following statement are equivalent:

(a) $f$ is continuous at $x_0$;

(b) If $\{x_n\}_{n=1}^{\infty}$ is any sequence in $X$ converging to $x_0$, then

$$\lim_{n\to\infty} f(x_n) = f(x_0)$$

[DF*] The functions we consider when discussing numerical methods are assumed to be continuous since this is a minimal requirement for predictable behavior. Functions that are not continuous can skip over points of interest, which can cause difficulties when attempting to approximate a solution to a problem. More sophisticated assumptions about a function generally lead to better approximation results.

**Definition 25.4: Differentiable**

Let $f$ be a function defined in an open interval containing $x_0$. The function $f$ is differentiable at $x_0$ if

$$f'(x_0) = \lim_{x\to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{25.5}$$

exists. The number $f'(x_0)$ is called the derivative of $f$ at $x_0$. A function that has a derivative at each number in a set $X$ is differentiable on $X$.
The derivative of $f$ at $x_0$ is the slope of the tangent line to the graph of $f$ at $(x_0, f(x_0))$.

**Theorem 25.2: Differentiable implies Continuity**

If the function $f$ is differentiable at $x_0$, then $f$ is continuous at $x_0$.

The set of all functions that have $n$ continuous derivatives on $X$ is denoted by $C^n(X)$, and the set of functions that have derivatives of all orders on $X$ is denoted by $C^\infty(X)$.

Functions in $C^\infty(X)$:

1. Polynomial
2. Rational
3. Trigonometric
4. Exponential
5. Logarithmic

where $X$ consists of all numbers for which the functions are defined.

Notes: If you are wondering, the theorem above is mathematics purely, you are wrong, all is connected, you just need to have solid reasoning / analysis and good at connecting the dots. Differentiability will reminds you of Motion in Physics and then to know that acceleration is the derivative of velocity with respect to time for Newtonian mechanics. We don't need to find a good financial reimbursement, as time goes on, it is revealed by itself how comprehend and mastering science will show all the gold and glitters potential when you master the knowledge, but remember, the knowledge itself that gained by rigorous study is a diamond, more powerful and valuable than gold. Another reminder, that nuclear bomb, atomic bomb all use physics, how can it not rule the world? There are still much more to explore in physics universe. Another example of idea on my mind: 10 most wanted FBI fugitives can be found immediately with more advanced science of computer graphics, physics, biology (gene) and mathematics.

**Theorem 25.3: Rolle's Theorem**

Suppose $f \in C[a,b]$ and $f$ is differentiable on $(a,b)$. If $f(a) = f(b)$, then a number $c$ in $(a,b)$ exists with

$$f'(c) = 0$$

**Theorem 25.4: Mean Value Theorem**

If $f \in C[a,b]$ and $f$ is differentiable on $(a,b)$, then a number $c$ in $(a,b)$ exists with

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

**Theorem 25.5: Extreme Value Theorem**

If $f \in C[a,b]$, then $c_1, c_2 \in [a,b]$ exist with

$$f(c_1) \leq f(x) \leq f(c_2)$$

for all $x \in [a,b]$. In addition, if $f$ is differentiable on $(a,b)$, then the numbers $c_1$ and $c_2$ occur neither at the endpoints of $[a,b]$ or where $f'$ is zero.

**Definition 25.5: Riemann Integral**

The Riemann integral of the function $f$ on the interval $[a,b]$ is the following limit, provided it exists:

$$\int_a^b f(x)\,dx = \lim_{\max \triangle x_{i\to 0}} \sum_{i=1}^{n} f(z_i)\triangle x_i \tag{25.6}$$

where the numbers $x_0, x_1, \ldots, x_n$ satisfy $a = x_0 \leq x_1 \leq \cdots \leq x_n = b$, and where $\triangle x_i = x_i - x_{i-1}$, for each $i = 1, 2, \ldots, n$, and $z_i$ is arbitrarily chosen in the interval $[x_{i-1}, x_i]$.

Every continuous function $f$ on $[a,b]$ is Riemann integrable on $[a,b]$. This permits us to choose, for computational convenience, the points $x_i$ to be equally spaced in $[a,b]$ and for each $i = 1, 2, \ldots, n$, to choose $z_i = x_i$. In this case,

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$$

with $x_i = a + \frac{i(b-a)}{n}$, $a = x_0$ and $b = x_n$. Basically, Riemann Integral is partitioning a function into $n$ rectangles to approximate the integration value with summation.

**Figure 25.1:** *Riemann integral, to be precise it is the right Riemann sum approximation for increasing function (**DFSimulatorC/Source Codes/Julia/ch11-riemannintegral.jl**).*

**Theorem 25.6: Weighted Mean Value Theorem for Integrals**

Suppose $f \in C[a,b]$, the Riemann integral of $g$ exists on $[a,b]$, and $g(x)$ does not change sign on $[a,b]$. Then there exists a number $c$ in $(a,b)$ with

$$\int_a^b f(x)g(x)\,dx = f(c)\int_a^b g(x)\,dx \tag{25.7}$$

when $g(x) \equiv 1$ then it is the usual Mean Value Theorem for Integrals. It gives the average value of the function $f$ over the interval $[a,b]$ as

$$f(c) = \frac{1}{b-a}\int_a^b f(x)\,dx \tag{25.8}$$

**Theorem 25.7: Generalized Rolle's Theorem**

Suppose $f \in C[a,b]$ is $n$ times differentiable on $(a,b)$. If $f(x)$ is zero at the $n+1$ distinct numbers $x_0, \ldots, x_n$ in $[a,b]$, then a number $c$ in $(a,b)$ exists with

$$f^{(n)}(c) = 0$$

**Theorem 25.8: Intermediate Value Theorem**

If $f \in C[a,b]$ and $K$ is any number between $f(a)$ and $f(b)$, then there exists a number $c$ in $(a,b)$ for which

$$f(c) = K$$

**Theorem 25.9: Taylor's Theorem**

Suppose $f \in C^n[a,b]$, that $f^{(n+1)}$ exists on $[a,b]$, and $x_0 \in [a,b]$. For every $x \in [a,b]$, there exists a number $\xi(x)$ between $x_0$ and $x$ with

$$f(x) = P_n(x) + R_n(x)$$

where

$$\begin{aligned} P_n(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \end{aligned} \tag{25.9}$$

and

$$R_n = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1} \tag{25.10}$$

Here $P_n(x)$ is called the $n$th Taylor polynomial for $f$ about $x_0$, and $R_n(x)$ is called the remainder term (or truncation error) associated with $P_n(x)$.

The infinite series obtained by taking the limit of $P_n(x)$ as $n \to \infty$ is called the Taylor series for $f$ about $x_0$. In the case where $x_0 = 0$, the Taylor polynomial is often called a Maclaurin polynomial, and the Taylor series is called a Maclaurin series.

The term truncation error refers to the error involved in using a truncated, or finite, summation to approximate the sum of an infinite series.

**Definition 25.6: Absolute and Relative Error**

If $p*$ is an approximation to $p$, the absolute error is

$$|p - p*|$$

and the relative error is

$$\frac{|p - p*|}{|p|}$$

provided that $p \neq 0$.

**Definition 25.7: Significant Digits**

The number $p*$ is said to approximate $p$ to $t$ significant digits if $t$ is the largest nonnegative integer for which

$$\frac{|p - p*|}{|p|} < 5 \times 10^{-t}$$

[DF*] An algorithm is a procedure that describes, in an unambiguous manner, a finite sequence of steps to be performed in a specified order. The object of the algorithm is to implement a procedure to solve a problem or approximate a solution to the problem.

We use pseudocode to describe the algorithms. This pseudocode specifies the form of the input to be supplied and the form of the desired output. Not all numerical procedures give satisfactory output for arbitrarily chosen input. As a consequence, a stopping technique independent of the numerical technique is incorporated into each algorithm to avoid infinite loops.

**Definition 25.8: Growth of Error**

Suppose that $E_0 > 0$ denotes an initial error and $E_n$ represents the magnitude of an error after $n$ subsequent operations. If $E_n \approx CnE_0$, where $C$ is a constant independent of $n$, then the growth of error is called to be linear.

If $E_n \approx C^n E_0$, for some $C > 1$, then the growth of error is called exponential.

## II. Solutions of Equations in One Variable

[DF*] In this section, we consider one of the most basic problems of numerical approximation, the root-finding problem. This process involves finding a root, or solution of an equation of the form

$$f(x) = 0$$

for a given function $f$.

[DF*] We realize this chapter is related with Linear Algebra when we start to learn about eigenvalue, when we want to compute eigenvalue, we will have to find the root of the characteristic equation $p(\lambda)$, a univariate function. The method that we are going to discuss in this section are going to make finding eigenvalue easier.

**Definition 25.9: Bisection Method**

The Bisection Method is based on the Intermediate Value Theorem. Suppose $f$ is a continuous function defined on the interval $[a, b]$ with $f(a)$ and $f(b)$ of opposite sign. By the Intermediate Value Theorem, there exists a number $p$ in $(a, b)$ with

$$f(p) = 0$$

We assume for simplicity that the root in the interval $[a, b]$ is unique. This method calls for a repeated halving of subintervals of $[a, b]$

[DF*] The bisection method is slow to converge, $N$ may become quite large before $|p - p_N|$ is sufficiently small, and a good intermediate approximation can be inadvertently discarded.

[DF*] The bisectionmethod has the important property that it always converges to a solution.

**Figure 25.2:** *The illustration of Bisection method to find the solution to a continuous function $f(x) = 0$, we need to input the endpoints a and b with the condition that $f(a) \times f(b) < 0$.*

**Computational Guide 25.1: Algorithm for Bisection Method**

To find a solution to $f(x) = 0$ given the continuous function $f$ on the interval $[a, b]$, where $f(a)$ and $f(b)$ have opposite signs:

**Input:**

- Endpoints $a, b$;
- Tolerance $\epsilon$;
- Maximum number of iterations $N_0$;

**Output:**

- Approximate solution $p$ or message of failure

The procedure:

1. Set $i = 1$;
   $fa = f(a)$;

2. While $i \leq N_0$ do
   {
   $p = a + \frac{b-a}{2}$;
   $fp = f(p)$;

   If $\frac{b-a}{2} < \epsilon$ then
   {
   OUTPUT(p);
   STOP; (Procedure completed successfully)

   $i = i + 1$;

   If $fa \times fp > 0$ then
   {
   $a = p$;
   $fa = fp$;
   }
   else
   {
   $b = p$;
   }

   }
   }

3. OUTPUT ('Method failed after $N_0$ iterations, the procedure was unsuccessful.');
   STOP;

**Definition 25.10: Fixed-Point Iteration**

A number $p$ is a fixed point for a given function $g$ if

$$g(p) = p$$

[DF*] Root-finding problems and fixed-point problems are equivalent classes in the following sense:

Given a root-finding problem $f(p) = 0$, we can define functions $g$ with a fixed point at $p$ in a number of ways, for example,

$$g(x) = x - f(x)$$

or

$$g(x) = x + 3f(x)$$

Conversely, if the function $g$ has a fixed point at $p$, then the function defined by $f(x) = x - g(x)$ has a zero at $p$.

[DF*] The fixed-point form is easier to analyze than the root-finding form. Certain fixed-point choices lead to very powerful root-finding techniques.

[DF*] True question to master fixed-point iteration: How can we find a fixed-point problem that produces a sequence that reliably and rapidly converges to a solution to a given root-finding problem?

**Theorem 25.10: Existence and Uniqueness of a Fixed Point**

These are the sufficient conditions for the existence and uniqueness of a fixed point

(a) If $g \in \mathbb{C}[a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$, then $g$ has a fixed point in $[a, b]$.

(b) If, in addition, $g'(x)$ exists on $(a, b)$ and a positive constant $k < 1$ exists with

$$|g'(x)| \leq k, \quad \text{for all} \in (a, b)$$

then the fixed point in $[a, b]$ is unique.

**Theorem 25.11: Fixed Point Theorem**

Let $g \in \mathbb{C}[a, b]$ be such that $g(x) \in [a, b]$, for all $x$ in $[a, b]$. Suppose, in addition, that $g'$ exists on $(a, b)$ and that a constant $0 < k < 1$ exists with

$$|g'(x)| \leq k, \quad \text{for all } x \in (a, b)$$

Then, for any number $p_0$ in $[a, b]$, the sequence defined by

$$p_n = g(p_{n-1}), \quad n \geq 1$$

converges to the unique fixed point $p$ in $[a, b]$.

**Figure 25.3:** *The illustration of Fixed-Point iteration, we need to give the initial approximation $p_0$ that will be located in the function $g(x)$ as $(p_0, g(p_0)) = (p_0, p_1)$.*

**Computational Guide 25.2: Algorithm for Fixed-Point Iteration**

To find a solution to $p = g(p)$ given an initial approximation $p_0$:

**Input:**

- Initial approximation $p_0$;
- Tolerance $\epsilon$;
- Maximum number of iterations $N_0$;

**Output:**

- Approximate solution $p$ or message of failure

The procedure:

1. Set $i = 1$;
2. While $i \leq N_0$ do
   {
   $p = g(p_0)$;

   If $|p - p_0| < \epsilon$ then
   {
   OUTPUT(p);
   STOP; (Procedure completed successfully)

   $i = i + 1$;

   $p_0 = p$;
   }

   }
3. OUTPUT ('Method failed after $N_0$ iterations, the procedure was unsuccessful.');
   STOP;

**Definition 25.11: Newton-Raphson Method**

The Newton-Raphson Method is one of the most powerful and well known numerical methods for solving a root-finding problem.
Suppose that $f \in \mathbb{C}^2[a,b]$. Let $\bar{x} \in [a,b]$ be an approximate to $p$ such that $f'(\bar{x}) \neq 0$ and $|p - \bar{x}|$ is "small." Consider the first Taylor polynomial for $f(x)$ expanded about $(\bar{x})$,

$$f(x) = f(\bar{x}) + (x - \bar{x})f'(\bar{x}) + \frac{(x-\bar{x})^2}{2} f''(\xi(x))$$

where $\xi(x)$ lies between $x$ and $\bar{x}$. Since $f(p) = 0$, this equation with $x = p$ gives

$$0 = f(\bar{x}) + (p - \bar{x})f'(\bar{x}) + \frac{(p-\bar{x})^2}{2} f''(\xi(p))$$

Newton's method is derived by assuming that since $|p - \bar{x}|$ is small, the term involving $(p - \bar{x})^2$ is much smaller, so

$$0 \approx f(\bar{x}) + (p - \bar{x})f'(\bar{x})$$

Solving for $p$ gives

$$p \approx \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}$$

Thus, the Newton-Raphson method, which starts with an initial approximation $p_0$ and generate the sequence $\{p_n\}_{n=0}^{\infty}$, is

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \quad \text{for } n \geq 1 \tag{25.11}$$

Starting with the initial approximation $p_0$, the approximation $p_1$ is the $x$-intercept of the tangent line to the graph of $f$ at $(p_0, f(p_0))$. The approximation $p_2$ is the $x$-intercept of the tangent line to the graph of $f$ at $(p_1, f(p_1))$ and so on.

[DF*] Newton-Raphson method is a functional iteration technique of the form

$$p_n = g(p_{n-1})$$

for which

$$g(p_{n-1}) = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \quad \text{for } n \geq 1 \tag{25.12}$$

this is the functional iteration technique that was used to give the rapid convergence.

[DF*] It is clear that Newton-Raphson method cannot be continued if $f'(p_{n-1}) = 0$ for some $n$. This method is the most effective when $f'$ is bounded away from zero near $p$.

**Theorem 25.12: Converging Sequence in Newton-Raphson Method**

Let $f \in \mathbb{C}^2[a,b]$. If $p \in [a,b]$ is such that $f(p) = 0$ and $f'(p) \neq 0$, then there exists a $\delta > 0$ such that Newton's method generates a sequence $\{p_n\}_{n=1}^{\infty}$ converging to $p$ for any initial approximation $p_0 \in [p - \delta . p + \delta]$.

**Figure 25.4:** *The illustration of Newton-Raphson method, at the beginning we use the first initial approximation $p_0$ then the approximation to $p_1$ is the x-intercept of the tangent line to the graph $f(x)$ at $(p_0, f(p_0))$. In the end, the root / the solution for $f(x)$ will be obtained using successive tangents if the conditions met.*

**Computational Guide 25.3: Algorithm for Newton-Raphson Method**

To find a solution to $f(x) = 0$ given the continuous function $f$ with initial approximation $p_0$:

**Input:**

- Initial approximation $p_0$;
- Tolerance $\epsilon$;
- Maximum number of iterations $N_0$;

**Output:**

- Approximate solution $p$ or message of failure

The procedure:

1. Set $i = 1$;
2. While $i \leq N_0$ do
   {
   $p = p_0 - \frac{f(p_0)}{f'(p_0)}$

   If $|p - p_0| < \epsilon$ then
   {
   OUTPUT(p);
   STOP; (Procedure completed successfully)

   }

   $i = i + 1$;
   $p_0 = p$;

   }

3. OUTPUT ('Method failed after $N_0$ iterations, the procedure was unsuccessful.');
   STOP;

**Definition 25.12: Secant Method**

By definition,

$$f'(p_{n-1}) = \lim_{x \to p_{n-1}} \frac{f(x) - f(p_{n-1})}{x - p_{n-1}}$$

Letting $x = p_{n-2}$, we have

$$f'(p_{n-1}) \approx \frac{f(p_{n-2}) - f(p_{n-1})}{p_{n-2} - p_{n-1}} = \frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}}$$

Using this approximation for $f'(p_{n-1})$ in Newton's formula gives:

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})} \tag{25.13}$$

This technique is called the Secant method.

**Figure 25.5:** *The illustration of Secant method, at the beginning we use the first initial approximations $p_0$ and $p_1$, the approximation $p_2$ is the x-intercept of the line joining $(p_0, f(p_0))$ and $(p_1, f(p_1))$. The approximation $p_3$ is the x-intercept of the line joining $(p_1, f(p_1))$ and $(p_2, f(p_2))$, and so on.*

**Computational Guide 25.4: Algorithm for Secant Method**

To find a solution to $f(x) = 0$ given the initial approximation $p_0$ and $p_1$:

**Input:**

- Initial approximation $p_0, p_1$;
- Tolerance $\epsilon$;
- Maximum number of iterations $N_0$;

**Output:**

- Approximate solution $p$ or message of failure

The procedure:

1. Set $i = 2$;
   $q_0 = f(p_0)$
   $q_1 = f(p_1)$
2. While $i \leq N_0$ do
   {
   $p = p_1 - q_1 \frac{p_1 - p_0}{q_1 - q_0}$

   If $|p - p_1| < \epsilon$ then
   {
   OUTPUT(p);
   STOP; (Procedure completed successfully)

   }

   $i = i + 1$;
   $p_0 = p_1$;
   $q_0 = q_1$;
   $p_1 = p$;
   $q_1 = f(p)$;

   }

3. OUTPUT ('Method failed after $N_0$ iterations, the procedure was unsuccessful.');
   STOP;

[DF*] Newton-Raphson mehod or the Secant method is often used to refine an answer obtained by another technique, such as the Bisection method, since these methods require a good first approximation but generally rapid convergence.

## i. C++ Computation: Bisection Method

This example is taken from Example 1 chapter 2.1 of [2] book.

The equation $f(x) = x^3 + 4x^2 - 10 = 0$ has a root in $[1, 2]$ since $f(1) = -5$ and $f(2) = 14$, use bisection method to find the root.

**Solution:**

$$f(a) = f(1) = -5$$
$$f(b) = f(2) = 14$$

the conditions are met, $f(a) \times f(b) < 0$, then

$$p_1 = a + \frac{b-a}{2} = 1 + \frac{2-1}{2} = 1.5$$
$$f(p_1) = 2.375$$

now we check that $f(p_1) \times f(a) < 0$ then

$$b = p_1 = 1.5$$

At the next iteration, $n = 2$, we will have

$$f(a) = f(1) = -5$$
$$f(b) = f(1.5) = 2.375$$

the conditions are met, $f(a) \times f(b) < 0$, then

$$p_2 = a + \frac{b-a}{2} = 1 + \frac{1.5-1}{2} = 1.25$$
$$f(p_2) = -1.796875$$

now we check that $f(p_2) \times f(b) < 0$ then

$$a = p_2 = 1.25$$

At the next iteration, $n = 3$, we will have

$$f(a) = f(1.25) = -1.796875$$
$$f(b) = f(1.5) = 2.375$$

the conditions are met, $f(a) \times f(b) < 0$, then

$$p_3 = a + \frac{b-a}{2} = 1.25 + \frac{1.5-1.25}{2} = 1.375$$
$$f(p_3) = 0.16210938$$

now we check that $f(p_3) \times f(a) < 0$ then

$$b = p_3 = 1.375$$

We will do iteration until the maximum number of iterations reached or the stop criteria is fulfilled / the solution is converging to the root of the function. After 13 iterations, we obtain

$$p_{13} = 1.36511$$
$$f(p_{13}) = -0.0019438$$

We can also check the stop criteria

$$\frac{b_{13} - a_{13}}{2} \leq \epsilon$$
$$\frac{1.36523 - 1.36499}{2} \leq 10^{-4}$$
$$0.00012 \leq 10^{-4}$$

it is a very rough estimate, we can say that $0.00012 \leq 10^{-4}$. The approximation is correct at least to four significant digits.

The correct value of $p$ is $p = 1.365230013$, when we compute till 17 iterations we get the correct $p$, the iteration number 13 fits the bill because the condition has met the stop criteria $\frac{b_n - a_n}{2} \leq \epsilon$.

We are going to use **GiNaC** library to input the univariate function and we can then substitute the value for the independent variable $x$ inside the for loop.

```
// g++ main.cpp -o result -lginac -lcln
#include <iostream>
#include <ginac/ginac.h>

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 5; // define maximum decimal digits
       symbol x("x");
       ex f, fa, fp;
       float a = 1;
       float b = 2;
       int N = 17;

       float p = a + (b-a)/2 ;

       f = pow(x,3) + 4*pow(x,2) - 10;
       fa = subs(f,x==a);
       fp = subs(f,x==p);

       cout << "f(x) = " << f <<endl;
       cout << endl;
       cout << "a = " << a <<endl;
       cout << "b = " << b <<endl;
```

```
        cout << "p = " << p <<endl;

        cout << "f(p) = " << subs(f,x==p) <<endl;
        cout << "f(b) = " << subs(f,x==b) <<endl;
        cout << "f(a) = " << fa << endl;
        cout << "f(a)*f(p) =" << fa*fp << endl;
        cout << endl;

        cout << setw(6) << "iteration" << "\t\t" << "a" << "\t\t\t" << "b"
            << "\t\t\t" << "p" << "\t\t\t" << "f(p)" << "\n";
        for (int i = 1; i <=N; i++)
        {
            p = a + (b-a)/2 ;
            fa = subs(f,x==a);
            fp = subs(f,x==p);
            if (fa * fp > 0)
            {
                cout << setw(6) << i << "\t\t\t" << a << "\t\t\t" << b
                     << "\t\t\t" << p << "\t\t\t" << subs(f,x==p) << "
                    \n";
                a = p;
                if ((b-a)/2 < pow(10,-4))
                {
                    cout << setw(6) << "Procedure completed
                        successfully" << "\n";
                }
            }
            else
            {
                cout << setw(6) << i << "\t\t\t" << a << "\t\t\t"<< b
                    << "\t\t\t" << p << "\t\t\t" << subs(f,x==p) << "\
                    n";
                b = p;
                if ((b-a)/2 < pow(10,-4))
                {
                    cout << setw(6) << "Procedure completed
                        successfully" << "\n";
                }
            }


            cout << endl;
        }

        return 0;
    }
```

**C++ Code 266:** *main.cpp "Bisection method"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- Inside the main **for loop** we compute the value of $p, f(a), f(p)$ first then we create the **if.. else** conditional statement to compute the value of $a, b, p$ for the next iteration.

  We don't really create a stop command or stop criteria here, we just use the number of iterations ($N$) we want to compute then it will prompt us a message "Procedure completed successfully" when $p$ is converges to the root (when $\frac{b-a}{2} < 10^{-4}$), thus at that iteration we already find the root, the current value for $p$ / $p_n$ with $n = 1, 2, \ldots, N$.

```
for (int i = 1; i <=N; i++)
{
        p = a + (b—a)/2 ;
        fa = subs(f,x==a);
        fp = subs(f,x==p);
        if (fa * fp > 0)
        {
                cout << setw(6) << i << "\t\t\t" << a << "\t\t\t"
                     << b << "\t\t\t" << p << "\t\t\t" << subs(f,
                    x==p) << "\n";
                a = p;
                if ((b—a)/2 < pow(10,—4))
                {
                        cout << setw(6) << "Procedure completed
                            successfully" << "\n";
                }
        }
        else
        {
                cout << setw(6) << i << "\t\t\t" << a << "\t\t\t"
                    << b << "\t\t\t" << p << "\t\t\t" << subs(f,x
                    ==p) << "\n";
                b = p;
                if ((b—a)/2 < pow(10,—4))
                {
                        cout << setw(6) << "Procedure completed
                            successfully" << "\n";
                }
        }

        cout << endl;
```

```
                }
```

```
f(x) = -10+4*x^2+x^3

a = 1
b = 2
p = 1.5
f(p) = 2.375
f(b) = 14.0
f(a) = -5
f(a)*f(p) =-11.875

iteration               a                       b                       p                       f(p)
     1                  1                       2                       1.5                     2.375

     2                  1                       1.5                     1.25                    -1.796875

     3                  1.25                    1.5                     1.375                   0.16210938

     4                  1.25                    1.375                   1.3125                  -0.8483887

     5                  1.3125                  1.375                   1.34375                 -0.35098267

     6                  1.34375                 1.375                   1.35938                 -0.096408844

     7                  1.35938                 1.375                   1.36719                 0.032355785

     8                  1.35938                 1.36719                 1.36328                 -0.03215003

     9                  1.36328                 1.36719                 1.36523                 7.200241E-5

    10                  1.36328                 1.36523                 1.36426                 -0.016046762

    11                  1.36426                 1.36523                 1.36475                 -0.007989168

    12                  1.36475                 1.36523                 1.36499                 -0.0039594173

    13                  1.36499                 1.36523                 1.36511                 -0.0019438267
Procedure completed successfully

    14                  1.36511                 1.36523                 1.36517                 -9.357929E-4
Procedure completed successfully

    15                  1.36517                 1.36523                 1.3652                  -4.3201447E-4
Procedure completed successfully

    16                  1.3652                  1.36523                 1.36522                 -1.7952919E-4
Procedure completed successfully

    17                  1.36522                 1.36523                 1.36523                 -5.4121017E-5
Procedure completed successfully
```

**Figure 25.6:** *The computation to find the root of $f(x) = x^3 + 4x^2 - 10 = 0$ with Bisection method (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Bisection Method with GiNaC/main.cpp**).*

## ii. C++ Computation: Fixed-Point Iteration

The equation $x^3 + 4x^2 - 10 = 0$ has a unique root in $[1, 2]$. There are many ways to change the equation to the fixed point form of

$$x = g(x)$$

using simple algebraic manipulation. For example, to obtain the function $g_3(x)$, we can manipulate the equation $x^3 + 4x^2 - 10 = 0$ as follows:

$$\begin{aligned} 4x^2 &= 10 - x^3 \\ x^2 &= \frac{1}{4}(10 - x^3) \\ x &= \pm\frac{1}{2}(10 - x^3)^{1/2} \end{aligned}$$

we choose the positive root for $g_3(x)$. Here is the list of some possible $g(x)$:

(a) $x = g_1(x) = x - x^3 - 4x^2 + 10$

(b) $x = g_2(x) = \left(\frac{10}{x} - 4x\right)^{1/2}$

(c) $x = g_3(x) = \frac{1}{2}(10 - x^3)^{1/2}$

(d) $x = g_4(x) = \left(\frac{10}{4+x}\right)^{1/2}$

(e) $x = g_5(x) = x - \frac{x^3+4x^2-10}{3x^2+8x}$

The C++ code is still using GiNaC, the results obtained for $g(x)$ of (c), (d), and (e) are impressive, knowing that the Bisection method requires 27 iterations for this accuracy. Choice (a) was divergent and that choice (b) become undefined because it involved the square root of a negative number, that explains why it has multiplication with $I = \sqrt{-1}$.

```
#include <iostream>
#include <ginac/ginac.h>

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#define pi 3.1415926535897

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 10; // define maximum decimal digits
       symbol x("x");
       ex f, g1, g2, g3, g4, g5, pn1, pn2, pn3, pn4, pn5;
       ex p0 = 1.5;
       ex p01 = 1.5;
       ex p02 = 1.5;
       ex p03 = 1.5;
```

```
ex p04 = 1.5;
ex p05 = 1.5;
int N = 20;

f = pow(x,3) + 4*pow(x,2) - 10;
g1 = x - pow(x,3) - 4*pow(x,2) - 10;
g2 = pow(10/x - 4*x,0.5);
g3 = 0.5*pow(10 - pow(x,3),0.5);
g4 = pow(10/(4+x),0.5);
g5 = x - (pow(x,3) + 4*(pow(x,2)) - 10)/(3*pow(x,2) + 8*x);

cout << "f(x) = " << f <<endl;
cout << "g1(x) = " << g1 <<endl;
cout << "g2(x) = " << g2 <<endl;
cout << "g3(x) = " << g3 <<endl;
cout << "g4(x) = " << g4 <<endl;
cout << "g5(x) = " << g5 <<endl;
cout << endl;

cout << setw(6) << "n" << "\t\t" << "p_{n} for g1" << "\t\t\t" << "
    p_{n} for g2" << "\n";
cout << setw(6) << "0" << "\t\t" << p0 << "\t\t\t\t" << p0 << "\n";

for (int i = 1; i <=4; i++)
{
       pn1 = subs(g1,x==p01);
       pn2 = subs(g2,x==p02);

       cout << setw(6) << i << "\t\t" << pn1 << "\t\t\t" << pn2 << "
           \n";

       if (abs(p0 - pn3) < pow(10,-5))
       {
              cout << "The procedure was successful." << endl;
              break;
       }
       p01 = pn1;
       p02 = pn2;
}
cout << endl;
cout << endl;

cout << setw(6) << "n" << "\t\t" << "p_{n} for g3" << "\t\t\t" << "
    p_{n} for g4" << "\t\t\t" << "p_{n} for g5" << "\n";
cout << setw(6) << "0" << "\t\t" << p0 << "\t\t\t\t" << p0 << "\t\t\
    t\t" << p0 << "\n";
for (int i = 1; i <=N; i++)
{
```

```
                    pn3 = subs(g3,x==p03);
                    pn4 = subs(g4,x==p04);
                    pn5 = subs(g5,x==p05);

                    cout << setw(6) << i << "\t\t" << pn3 << "\t\t" << pn4 << "\t
                        \t" << pn5 << "\n";

                    if (abs(p0 - pn3) < pow(10,-5))
                    {
                           cout << "The procedure was successful." << endl;
                           break;
                    }
                    p03 = pn3;
                    p04 = pn4;
                    p05 = pn5;
            }
            return 0;
      }
```

**C++ Code 267:** *main.cpp "Fixed point iteration"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
f(x) = -10+x^3+4*x^2
g1(x) = -10-x^3-4*x^2+x
g2(x) = sqrt(10*x^(-1)-4*x)
g3(x) = (0.5)*sqrt(10-x^3)
g4(x) = (3.1622776601683795)*(4+x)^(-0.5)
g5(x) = -(3*x^2+8*x)^(-1)*(-10+x^3+4*x^2)+x

     n          p_{n} for g1                    p_{n} for g2
     0          1.5                             1.5
     1          -20.875                 0.8164965809277257
     2          7322.669921875                  2.9969088057872217
     3          -3.928669856926092E11                   1.800772207646954E-16+2.9412350614769713*I
     4          6.063684607656086E34                    2.7536223884357987-2.753622388435798*I


     n          p_{n} for g3                    p_{n} for g4                    p_{n} for g5
     0          1.5                             1.5                             1.5
     1          1.286953767623375               1.3483997249264843              1.3733333333333333
     2          1.4025408035395783              1.367376371991283               1.3652620148746266
     3          1.3454583740232942              1.364957015402487               1.3652300139161466
     4          1.375170252816038               1.3652647481134423              1.3652300134140969
     5          1.3600941927617332              1.3652255941605251              1.3652300134140969
     6          1.3678469675921328              1.365230575673434               1.3652300134140969
     7          1.3638870038840212              1.3652299418781833              1.3652300134140969
     8          1.36591673339004                1.3652300225155685              1.3652300134140969
     9          1.3648782171936773              1.365230012256122               1.3652300134140969
    10          1.3654100611699567              1.3652300135614255              1.3652300134140969
    11          1.365137820669213               1.3652300133953525              1.3652300134140969
    12          1.3652772085244786              1.3652300134164816              1.3652300134140969
    13          1.3652058502970472              1.3652300134137936              1.3652300134140969
    14          1.3652423837188388              1.3652300134141355              1.3652300134140969
    15          1.365223680225282               1.365230013414092               1.3652300134140969
    16          1.3652332557424998              1.3652300134140976              1.3652300134140969
    17          1.3652283534626268              1.3652300134140969              1.3652300134140969
    18          1.365230863243637               1.3652300134140969              1.3652300134140969
    19          1.3652295783339585              1.3652300134140969              1.3652300134140969
    20          1.3652302361581814              1.3652300134140969              1.3652300134140969
```

**Figure 25.7:** *The computation to find the root of $f(x) = x^3 + 4x^2 - 10 = 0$ with 5 possible $g(x)$ functions by using Fixed-Point method (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Fixed-Point Method with GiNaC/main.cpp**).*

## iii. C++ Computation: Newton-Raphson Method

Suppose we would like to find the root of $g(x) = \cos\ x - x$ with Newton-Raphson method.

First, we will have to give the initial approximation $p_0$ then compute

$$p_n = p_{n-1} - \frac{g(p_{n-1})}{f'(p_{n-1})}$$

with $n \geq 1$.

We will do the computation, let it iterates till the solution converges, and we will obtain the root. Watch out, as there might be more than 1 root, so if we change our initial approximation $p_0$ we might end up with another root we are looking for.

With $p_0 = \frac{\pi}{4}$ we will obtain the solution / the root for $g(x) = \cos\ x - x$ at

$$p_n = 0.73908514$$

at $n = 3$ or after the third iteration. Since we are having a function of trigonometric and input of initial approximation in $\pi$ (radian) term, thus we can convert the result to be in degree term by multiplying it with the conversion number from radian to degree.

If you look closely at the function it will become like this:

$$\cos\ x = x$$

Thus the root is actually the intersection point of the line $y = x$ and $y = \cos\ x$, which is true at 0.73908514 radian or 42.34646 degree.

The C++ code is still using **GiNaC** library, it is capable of computing the symbolic derivative of an univariate function and we can substitute the independent variable at the **for** loop.

```
#include <iostream>
#include <ginac/ginac.h>

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#define pi 3.1415926535897

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 5; // define maximum decimal digits
       symbol x("x");
       ex f, fd, fp, fpd;
       ex pn;
       ex p0 = (pi/4);
       int N = 5;
```

```
        f = cos(x) - x; // the input x has to be in Radian
        fd = diff(f,x);
        fp = subs(f,x==p0);
        fpd = subs(fd,x==p0);

        cout << "f(x) = " << f <<endl;
        cout << endl;
        cout << "f'(x) = " << fd <<endl;
        cout << endl;
        cout << "p_{0} = " << p0 <<endl;
        cout << endl;

        cout << setw(6) << "n" << "\t\t" << "p_{n}" << "\t\t\t" << "p_{n} in
             Degree" << "\n";
        cout << setw(6) << "0" << "\t\t" << p0 << "\t\t" << p0*RADTODEG << "
            \n";
        for (int i = 1; i <=N; i++)
        {
            fp = subs(f,x==p0);
            fpd = subs(fd,x==p0);
            pn = p0 - (fp/fpd);

            cout << setw(6) << i << "\t\t" << pn << "\t\t" << pn*RADTODEG
                 << "\n";

            if (abs(p0 - pn) < pow(10,-5))
            {
                cout << "The procedure was successful." << endl;
                break;
            }
            p0 = pn;
        }
        return 0;
    }
```

**C++ Code 268:** *main.cpp "Newton-Raphson method"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

Explanation for the codes:

- In Numerical Method, the most important thing is the iteration part, so just like in Bisection Method, we are going to talk about what happen inside the **for** loop here.

We follow the algorithm of Newton-Raphson and we will compute at the first iteration when $i = 1$ to compute

$$p_n = p_{n-1} - \frac{g(p_{n-1})}{f'(p_{n-1})}$$

We give input of the initial approximation, $p_0$, thus the order of the computation will be like this:

$$\begin{aligned} p_1 &= p_0 - \frac{g(p_0)}{f'(p_0)} \\ p_2 &= p_1 - \frac{g(p_1)}{f'(p_1)} \\ &\vdots \\ p_N &= p_{N-1} - \frac{g(p_{N-1})}{f'(p_{N-1})} \end{aligned}$$

from the code we know that we give the maximum number of iteration $N = 5$, $p_0 = \frac{\pi}{4}$, the tolerance $\epsilon = 10^{-5}$ and the function $g(x) = \cos\ x - x$.

The iteration / the **for** loop will be terminated when we already find the root of the function $g(x)$, the condition for the convergence is

$$|p_n - p_{n-1}| < \epsilon$$

$$|p_n - p_{n-1}| < 10^{-5}$$

and it will prompt a message "The procedure was successul." The

```
for (int i = 1; i <=N; i++)
{
        fp = subs(f,x==p0);
        fpd = subs(fd,x==p0);
        pn = p0 - (fp/fpd);

        cout << setw(6) << i << "\t\t" << pn << "\t\t" << pn*
            RADTODEG << "\n";

        if (abs(p0 - pn) < pow(10,-5))
        {
                cout << "The procedure was successful." << endl;

                break;
        }
        p0 = pn;
}
```

```
f(x) = cos(x)-x

f'(x) = -1-sin(x)

p_{0} = 0.7853982

     n          p_{n}                    p_{n} in Degree
     0          0.7853982                45.0
     1          0.73953617               42.372303
     2          0.73908514               42.34646
     3          0.73908514               42.34646
The procedure was successful.
```

**Figure 25.8:** *The computation to find the root of $f(x) = \cos\ x - x$ with Newton-Raphson method (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Newton-Raphson Method with GiNaC/main.cpp**).*

## iv. C++ Computation: Secant Method

Use the Secant method to find a solution to $x = \cos\ x$.

**Solution:**

If Newton-Raphson method needs 1 initial approximation, in Secant method, we need to use 2 initial approximations $p_0$ and $p_1$. Thus we will have

$$f(x) = \cos\ x - x$$

and we will choose the initial approximations

$$p_0 = 0.5$$
$$p_1 = \frac{\pi}{4}$$

we just need to follow the Secant method formula:

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})}$$

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})}$$
$$p_n = p_{n-1} - \frac{(\cos(p_{n-1}) - p_{n-1})(p_{n-1} - p_{n-2})}{(\cos(p_{n-1}) - p_{n-1}) - (\cos(p_{n-2}) - p_{n-2})}$$

Do the iteration until the stop criteria requirement is met or the maximum number of iteration has reached.

We see that $p_5$' accuracy is up to tenth decimal place. The convergence of the Secant method is much faster than functional iteration but slightly slower than Newton-Raphson method, which obtained this degree of accuracy with $p_3$ / after 3 iterations.

For the C++ code, we still using **GiNaC** library to compute the Secant method.

```
#include <iostream>
#include <ginac/ginac.h>
```

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#define pi 3.1415926535897

using namespace std;
using namespace GiNaC;

int main()
{
       Digits = 10; // define maximum decimal digits
       symbol x("x");
       ex f, fp0, fp1, fpn;
       ex pn;
       ex p0 = 0.5;
       ex p1 = (pi/4);
       int N = 10;

       f = cos(x) - x; // the input x has to be in Radian
       fp0 = subs(f,x==p0);
       fp1 = subs(f,x==p1);

       cout << "f(x) = " << f <<endl;
       cout << endl;
       cout << "p_{0} = " << p0 <<endl;
       cout << endl;
       cout << "p_{1} = " << p1 <<endl;
       cout << endl;

       cout << setw(6) << "n" << "\t\t" << "p_{n}" << "\t\t\t\t" << "p_{n}
           in Degree" << "\n";
       cout << setw(6) << "0" << "\t\t" << p0 << "\t\t\t\t" << p0*RADTODEG
           << "\n";
       cout << setw(6) << "1" << "\t\t" << p1 << "\t\t" << p1*RADTODEG << "
           \n";
       for (int i = 2; i <=N; i++)
       {
              fp0 = subs(f,x==p0);
              fp1 = subs(f,x==p1);
              pn = p1 - ((fp1*(p1-p0)) / (fp1 - fp0));

              cout << setw(6) << i << "\t\t" << pn << "\t\t" << pn*RADTODEG
                   << "\n";

              if (abs(p1 - pn) < pow(10,-5))
              {
                     cout << "The procedure was successful." << endl;
                     break;
```

```
                }
                p0 = p1;
                fp0 = fp1;
                p1 = pn;
                fpn = subs(f,x==pn);
                fp1 = fpn;

            }
            return 0;
        }
```

**C++ Code 269:** *main.cpp "Secant method"*

To compile it, type:
**g++ -o main main.cpp -lginac -lcln**
**./main**

or with Makefile, type:
**make**
**./main**

```
f(x) = cos(x)-x

p_{0} = 0.5

p_{1} = 0.785398163397425

     n          p_{n}                          p_{n} in Degree
     0          0.5                            28.647890090942383
     1          0.785398163397425              45.00000052527487
     2          0.7363841388365835             42.1917037482074
     3          0.7390581392138897             42.34491268603182
     4          0.7390851493372764             42.34646025212407
     5          0.7390851332150645             42.34645932838936
The procedure was successful.
```

**Figure 25.9:** *The computation to find the root of $f(x) = \cos\ x - x$ with Secant method (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Secant Method with GiNaC/-main.cpp**).*

## III. Interpolation and Polynomial Approximation

One of the most useful and well-known classes of functions mapping the set of real numbers into itself is the class of algebraic polynomials, the set of functions of the form

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

where $n$ is nonnegative integer and $a_0, a_1, \ldots, a_n$ are real constants. One reason for their importance is that they uniformly approximate continuous functions. Given any function, defined and continuous on a closed and bounded interval, there exists a polynomial that is as "close" to the given function as desired.

**Theorem 25.13: Weierstrass Approximation Theorem**

Suppose that $f$ is defined and continuous on $[a, b]$. For each $\epsilon > 0$, there exists a polynomial $P(x)$, with the property that

$$f(x) - P(x) < \epsilon, \quad \forall\, x \in [a, b]$$

[DF*] We will learn how to produce an interpolating polynomials from input of: $x_0, x_1, \ldots, x_n$ as the $x$ axis, and $f_0, f_1, \ldots, f_n$ as the $y$ axis.

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $x_i$ | 1 | 2 | 3 | 4 | 5 |
| $f_i$ | 0.5 | 3.3 | 4.2 | 5.1 | 6.0 |

**Table 25.1:** *The variable $x_i$ denotes time, while $f_i = f(x_i)$ is the function at time $i$ that represents velocity of an object. With $t_0$ as the initial time and $f_0$ is the initial velocity at time $x_0$.*

[DF*] The Taylor polynomials are not appropriate for interpolation [2]. The problem of determining a polynomial of degree one that passes through the distinct points $(x_0, y_0)$ and $(x_1, y_1)$ is the same as approximating a function $f$ for which $f(x_0) = y_0$ and $f(x_1) = y_1$ by means of a first-degree polynomial interpolating the values of $f$ at the given points.

We first define the function

$$L_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

and then define

$$P(x) = L_0(x)f(x_0) + L_1(x)f(x_1)$$

Since

$$L_0(x_0) = 1,\ L_0(x_1) = 0,\ L_1(x_0) = 0,\ L_1(x_1) = 1$$

we have

$$P(x_0) = 1 \cdot f(x_0) + 0 \cdot f(x_1) = f(x_0) = y_0$$

and

$$P(x_1) = 0 \cdot f(x_0) + 1 \cdot f(x_1) = f(x_1)y_1$$

so $P$ is the unique linear function passing through $(x_0, y_0)$ and $(x_1, y_1)$.

To generalize the concept of linear interpolation, consider construction of a polynomial of degree at most $n$ that passes through $n+1$ points

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

to approximate that $n+1$ points, we will use the well-established Lagrange Interpolating Polynomial.

**Theorem 25.14: $n$th Lagrange Interpolating Polynomial**

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct numbers and $f$ is a funciton whose values are given at these numbers, then a unique polynomial $P(x)$ of degreen at most $n$ exists with

$$f(x_k) = P(x_k)$$

for each $k = 0, 1, \ldots, n$ This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x) \tag{25.14}$$

where, for each $k = 0, 1, \ldots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x-x_0)(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})(x_k-x_n)} \\ &= \prod_{i=0,\ i\neq k}^{n} \frac{(x-x_i)}{(x_k-x_i)} \end{aligned} \tag{25.15}$$

## IV. Numerical Differentiation

The derivative of the function $f$ at $x_0$ is defined as

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h} \tag{25.16}$$

for small values of $h$, we can generate an approximation to $f'(x)$. But, it is not always successful due to roundoff error.

To approximate $f'(x_0)$, suppose first that $x_0 \in (a, b)$, where $f \in C^2[a, b]$, and that $x_1 = x_0 + h$ for some $h \neq 0$ that is sufficiently small to ensure that $x_1 \in [a, b]$. Now we will need to remember the theorem for Lagrange Polynomial

**Theorem 25.15: $n$th Lagrange Interpolating Polynomial**

If $x_0, x_1, \ldots, x_n$ are $n+1$ distinct numbers and $f$ is a funciton whose values are given at these numbers, then a unique polynomial $P(x)$ of degreen at most $n$ exists with

$$f(x_k) = P(x_k)$$

for each $k = 0, 1, \ldots, n$ This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x) \tag{25.17}$$

where, for each $k = 0, 1, \ldots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x-x_0)(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})(x_k-x_n)} \\ &= \prod_{i=0,\ i\neq k}^{n} \frac{(x-x_i)}{(x_k-x_i)} \end{aligned} \tag{25.18}$$

**Definition 25.13: Forward and Backward-Difference Formula**

Forward-difference Formula

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \tag{25.19}$$

Backward-difference Formula

$$f'(x_0) = \frac{f(x_0-h) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \tag{25.20}$$

## i. C++ Computation: Numerical Derivative of a Univariate Function

Suppose we have a function

$$f(x) = x^3$$

and we want to compute the derivative till the fifth order at $x = 30$, we can use Autodiff, an automatic differentiation C++ library that has moved under **Boost** library (math/differentiation/autodiff).

Autodiff facilitates the automatic differentiation (forward mode) of mathematical functions of single and multiple variables.

The implementation of autodiff is based upon the Taylor series expansion of an analytic function $f$ at the point $x_0$:

$$\begin{aligned} f(x_0+\epsilon) &= f(x_0) + f'(x_0)\epsilon + \frac{f''(x_0)}{2!}\epsilon^2 + \frac{f'''(x_0)}{3!}\epsilon^3 + \dots \\ &= \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\epsilon^n + O(\epsilon^{N+1}) \end{aligned}$$

The essential idea of autodiff is the substitution of numbers with polynomials in the evaluation of $f(x_0)$. By substituting the number $x_0$ with the first-order polynomial $x_0 + \epsilon$ and using the same algorithm to compute $f(x_0 + \epsilon)$, the resulting polynomial in $\epsilon$ contains the function's derivatives $f'(x_0), f''(x_0), f'''(x_0), \dots$ within the coefficients. Each coefficient is equal to the derivative of its respective order, divided by the factorial of the order.

In greater detail, assume one is interested in calculating the first $N$ derivatives of $f$ at $x_0$. Without loss of precision to the calculation of the derivatives, all terms $O(\epsilon^{N+1})$ that include powers of $\epsilon$ greater than $N$ can be discarded. This is due to the fact that each term in a polynomial depends only upon equal and lower-order terms under arithmetic operations. Under this truncation rules, $f$ provides a polynomial-to-polynomial transformation:

$$f: \quad x_0 + \epsilon \quad \rightarrowtail \sum_{n=0}^{N} y_n \epsilon^n = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\epsilon^n$$

C++'s ability to overload operators and functions allows for the creation of a class **fvar** that represents polynomials in $\epsilon$. Thus the same algorithm $f$ that calculates the numeric value of $y_0 = f(x_0)$, when written to accept and return variables of a generic (template) type, is also used to calculate the polynomial

$$\sum_{n=0}^{N} y_n \epsilon^n = f(x_0 + \epsilon)$$

The derivative $f^{(n)}(x_0)$ are then found from the product of the respective factorial $n!$ and coefficient $y_n$:

$$\frac{d^n f}{dx^n}(x_0) = n! y_n$$

To calculate the derivative of

$$f(x) = x^3$$

at $x = 30$, we will use the C++ code below with Autodiff from Boost.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>
#include <math.h>

using namespace std;

template <typename T>
T get_x_coordinate(T const & variable)
{
       return variable;
}

int main()
{
       float var_x= 30;
       double const variable{var_x};
       auto const x{::boost::math::differentiation::make_fvar<double, 5>(
           variable)}; // to find derivative till order 5
       auto const dx{get_x_coordinate(x*x*x)};
       cout << "x = " << var_x <<endl;
       cout << "x in autodiff = " << x << endl;
       cout << "dx in autodiff = " << dx << endl;
       cout << "f(x) = x*x*x = " << dx.derivative(0) << endl;
       cout << "d f(x) = 3*x*x = " << dx.derivative(1) << endl;
       cout << "d(d f(x)) = 6*x = " << dx.derivative(2) << endl;
       cout << "d^2(d f(x)) = " << dx.derivative(3) << endl;
       cout << "d^3(d f(x)) = " << dx.derivative(4) << endl; cout << "d^4(d
            f(x)) = " << dx.derivative(5) << endl; return 0;
}
```

**C++ Code 270:** *main.cpp "Compute Numerical Derivative with Autodiff"*

To compile it, type:
**g++ -o result main.cpp -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./main**

Some explanations for the codes:

- The above calculate

$$\begin{array}{lll} f(30) & = x^3|_{x=30} & = 27000 \\ f'(30) & = 3x^2|_{x=30} & = 2700 \\ f''(30) & = 6x|_{x=30} & = 180 \\ f'''(30) & = 6|_{x=30} & = 6 \\ f^{(4)}(30) & = 0|_{x=30} & = 0 \\ f^{(5)}(30) & = 0|_{x=30} & = 0 \end{array}$$

- We define the $x = 30$ as a float variable, we define the function $f(x)$ inside the **auto const dx** under the **get_x_coordinate(x*x*x)**, with $x * x * x$ represents the function $f(x) = x^3 = x * x * x$

```
float var_x= 30;
double const variable{var_x};
auto const x{::boost::math::differentiation::make_fvar<double,
    5>(variable)}; // to find derivative till order 5
auto const dx{get_x_coordinate(x*x*x)};
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C+++ Gnuplot
mbolicC++/ch25-Numerical Differentiation and Integration/Autodiff Compute Der
tive ]# ./main
x = 30
x in autodiff = depth(1)(30,1,0,0,0,0)
dx in autodiff = depth(1)(27000,2700,90,1,0,0)
f(x) = x*x*x = 27000
d f(x) = 3*x*x = 2700
d(d f(x))  = 6*x = 180
d^2(d f(x)) = 6
d^3(d f(x)) = 0
d^4(d f(x)) = 0
```

**Figure 25.10:** *The computation till the fifth derivative of $f(x) = x^3$ at $x = 30$ with Boost' Autodiff and C++ (**DF-SimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/Autodiff Compute Derivative/main.cpp**).*

## ii. C++ Computation: Numerical Derivative of a Multivariable Function

### iii. C++ Plot and Computation: 2D Plot for Derivative of a Univariate Function

When we are using Python or JULIA it is very easy to compute and then plot the symbolic derivative of a univariate function, for example, consider

$$f(x) = x^3 + x^2$$

the derivative will be

$$\frac{df(x)}{dx} = 3x^2 + 2x$$

with SymPy and Plots packages from JULIA or matplotlib and SymPy from Python we can plot the derivative and the underlying function in less than 10 lines, but with C++ we will take more than 100 lines (it contains 361 lines of codes to be precise), the good news is the compiling and running when we finish the long lines of work worth all our time.

**Figure 25.11:** *The plot of the original function $f(x) = x^3 + x^2$ and the derivative $f'(x)$ with JULIA.*

The code originally come from:
**http://en.wikibooks.org/wiki/OpenGL_Programming**
**https://en.wikibooks.org/wiki/OpenGL_Programming/Scientific_OpenGL_Tutorial_03**

we modify it and add **SymbolicC++** and **Boost**.

In this section, we use **SymbolicC++** to compute the symbolic derivative, afterwards for the plotting part we are going to use **Boost** library to compute the numerical derivative of a function with **autodiff** library API inside the **Boost**.

**Figure 25.12:** *The common directory to make the graphing works for this section (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/common**).*

You need to also copy the **common** folder that have **shader_utils.cpp, shader_utils.h, shader_utils.o** files in parallel with the folder containing this source code.

The code will be very long since we are using full OpenGL for the graphics API, with GLEW along with GLUT instead of Gnuplot. It is an example of raw C++ coding by mixing some C++ libraries to compute the derivative then hand it over to OpenGL and friends to do the plotting.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <GL/glew.h>
#include <GL/glut.h>

#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

```
#include <glm/gtc/type_ptr.hpp>

#include "symbolicc++.h" // for symbolic computation
#include <boost/math/differentiation/autodiff.hpp>

#include "../common/shader_utils.h"

GLuint program;
GLint attribute_coord2d;
GLint uniform_color;
GLint uniform_transform;

float offset_x = 0;
float scale_x = 1;
int mode = 0;

struct point {
       GLfloat x;
       GLfloat y;
};
GLuint vbo_derivative;
GLuint vbo[3];

const int border = 10;
const int ticksize = 10;

template <typename T>
T get_x_coordinate(T const & phi)
{
       return phi;
}

int init_resources() {
       program = create_program("graph.v.glsl", "graph.f.glsl");
       if (program == 0)
       return 0;

       attribute_coord2d = get_attrib(program, "coord2d");
       uniform_transform = get_uniform(program, "transform");
       uniform_color = get_uniform(program, "color");

       if (attribute_coord2d == -1 || uniform_transform == -1 ||
           uniform_color == -1)
       return 0;

       // Create the vertex buffer object
       glGenBuffers(3, vbo);
       glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
```

```
        // Create our own temporary buffer
        point graph[2000];

        // Fill it in just like an array
        for (int i = 0; i < 2000; i++)
        {
            float x = (i -1000) / 10.0;

            graph[i].x = x;
            graph[i].y = (x*x*x) + (x*x);
        }

        // Tell OpenGL to copy our array to the buffer object
        glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph,
            GL_STATIC_DRAW);

        // Create a VBO for the border
        static const point border[4] = { {-1, -1}, {1, -1}, {1, 1},
             {-1, 1} };
        glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
        glBufferData(GL_ARRAY_BUFFER, sizeof border, border,
            GL_STATIC_DRAW);

        return 1;
}

// Create a projection matrix that has the same effect as glViewport
    ().
// Optionally return scaling factors to easily convert normalized
    device coordinates to pixels.
//
glm::mat4 viewport_transform(float x, float y, float width, float
    height, float *pixel_x = 0, float *pixel_y = 0) {
        // Map OpenGL coordinates (-1,-1) to window coordinates (x,y
            ),
        // (1,1) to (x + width, y + height).

        // First, we need to know the real window size:
        float window_width = glutGet(GLUT_WINDOW_WIDTH);
        float window_height = glutGet(GLUT_WINDOW_HEIGHT);

        // Calculate how to translate the x and y coordinates:
        float offset_x = (2.0 * x + (width - window_width)) /
            window_width;
        float offset_y = (2.0 * y + (height - window_height)) /
            window_height;
```

```
        // Calculate how to rescale the x and y coordinates:
        float scale_x = width / window_width;
        float scale_y = height / window_height;

        // Calculate size of pixels in OpenGL coordinates
        if (pixel_x)
        *pixel_x = 2.0 / width;
        if (pixel_y)
        *pixel_y = 2.0 / height;

        return glm::scale(glm::translate(glm::mat4(1), glm::vec3(
            offset_x, offset_y, 0)), glm::vec3(scale_x, scale_y, 1));
}

void display() {
        int window_width = glutGet(GLUT_WINDOW_WIDTH);
        int window_height = glutGet(GLUT_WINDOW_HEIGHT);

        glUseProgram(program);

        glClearColor(1, 1, 1, 1);
        glClear(GL_COLOR_BUFFER_BIT);

        /* Draw the graph */

        // Set our viewport, this will clip geometry
        glViewport(border + ticksize, border + ticksize, window_width
             - border * 2 - ticksize, window_height - border * 2 -
             ticksize);

        // Set the scissor rectangle,this will clip fragments
        glScissor(border + ticksize, border + ticksize, window_width
            - border * 2 - ticksize, window_height - border * 2 -
            ticksize);

        glEnable(GL_SCISSOR_TEST);

        // Set our coordinate transformation matrix
        glm::mat4 transform = glm::translate(glm::scale(glm::mat4(1.0
            f), glm::vec3(scale_x, 1, 1)), glm::vec3(offset_x, 0, 0))
            ;
        glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm::
            value_ptr(transform));

        // Set the color to green
        GLfloat green[4] = { 0, 1, 0, 1 };
        glUniform4fv(uniform_color, 1, green);
```

```
// Draw using the vertices in our vertex buffer object
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

glEnableVertexAttribArray(attribute_coord2d);
glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
    GL_FALSE, 0, 0);
glDrawArrays(GL_LINE_STRIP, 0, 2000);

// Stop clipping
glViewport(0, 0, window_width, window_height);
glDisable(GL_SCISSOR_TEST);

/* Draw the borders */

float pixel_x, pixel_y;

// Calculate a transformation matrix that gives us the same
    normalized device coordinates as above
transform = viewport_transform(border + ticksize, border +
    ticksize, window_width - border * 2 - ticksize,
    window_height - border * 2 - ticksize, &pixel_x, &
    pixel_y);

// Tell our vertex shader about it
glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm::
    value_ptr(transform));

// Set the color to black
GLfloat black[4] = { 0, 0, 0, 1 };
glUniform4fv(uniform_color, 1, black);

// Draw a border around our graph
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
    GL_FALSE, 0, 0);
glDrawArrays(GL_LINE_LOOP, 0, 4);

/* Draw the y tick marks */

point ticks[42];

for (int i = 0; i <= 20; i++) {
       float y = -1 + i * 0.1;
       float tickscale = (i % 10) ? 0.5 : 1;

       ticks[i * 2].x = -1;
       ticks[i * 2].y = y;
       ticks[i * 2 + 1].x = -1 - ticksize * tickscale *
```

```
            pixel_x;
        ticks[i * 2 + 1].y = y;
}

glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
glBufferData(GL_ARRAY_BUFFER, sizeof ticks, ticks,
    GL_DYNAMIC_DRAW);
glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
    GL_FALSE, 0, 0);
glDrawArrays(GL_LINES, 0, 42);

/* Draw the x tick marks */

float tickspacing = 0.1 * powf(10, -floor(log10(scale_x)));
    // desired space between ticks, in graph coordinates
float left = -1.0 / scale_x - offset_x; // left edge, in
    graph coordinates
float right = 1.0 / scale_x - offset_x; // right edge, in
    graph coordinates
int left_i = ceil(left / tickspacing); // index of left tick,
     counted from the origin
int right_i = floor(right / tickspacing); // index of right
    tick, counted from the origin
float rem = left_i * tickspacing - left; // space between
    left edge of graph and the first tick

float firsttick = -1.0 + rem * scale_x; // first tick in
    device coordinates

int nticks = right_i - left_i + 1; // number of ticks to
    show

if (nticks > 21)
nticks = 21; // should not happen

for (int i = 0; i < nticks; i++) {
        float x = firsttick + i * tickspacing * scale_x;
        float tickscale = ((i + left_i) % 10) ? 0.5 : 1;

        ticks[i * 2].x = x;
        ticks[i * 2].y = -1;
        ticks[i * 2 + 1].x = x;
        ticks[i * 2 + 1].y = -1 - ticksize * tickscale *
            pixel_y;
}

glBufferData(GL_ARRAY_BUFFER, sizeof ticks, ticks,
    GL_DYNAMIC_DRAW);
```

```
glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
    GL_FALSE, 0, 0);
glDrawArrays(GL_LINES, 0, nticks * 2);

// And we are done.

glDisableVertexAttribArray(attribute_coord2d);
switch (mode)
{
       case 0:
       glDrawArrays(GL_LINE_STRIP, 0, 2000);
       break;
       case 1:
       glClearColor(1, 1, 1, 1);
       glClear(GL_COLOR_BUFFER_BIT);

       glViewport(border + ticksize, border + ticksize,
           window_width - border * 2 - ticksize,
           window_height - border * 2 - ticksize);

       glScissor(border + ticksize, border + ticksize,
           window_width - border * 2 - ticksize,
           window_height - border * 2 - ticksize);

       glEnable(GL_SCISSOR_TEST);

       // Set our coordinate transformation matrix
       glm::mat4 transform = glm::translate(glm::scale(glm::
           mat4(1.0f), glm::vec3(scale_x, 1, 1)), glm::vec3(
           offset_x, 0, 0));
       glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm
           ::value_ptr(transform));

       // Set the color to green
       GLfloat green[4] = { 0, 1, 0, 1 };
       glUniform4fv(uniform_color, 1, green);

       glGenBuffers(1, &vbo_derivative);
       glBindBuffer(GL_ARRAY_BUFFER, vbo_derivative);

       // Create our own temporary buffer
       point graph_derivative[2000];

       // Fill it in just like an array
       for (int i = 0; i < 2000; i++)
       {
              float var_x = (i - 1000) / 10.0;
              double const phi{var_x};
```

```
                        auto const x{::boost::math::differentiation::
                            make_fvar<double, 5>(phi)}; // to find
                            derivative till order 5
                        auto const dx{get_x_coordinate(x*x*x + x*x)};
                            // we define the function f(x) here :
                            get_x_coordinate(f(x))

                        graph_derivative[i].x = var_x;
                        graph_derivative[i].y = dx.derivative(1);
                }
                // Tell OpenGL to copy our array to the buffer object
                glBufferData(GL_ARRAY_BUFFER, sizeof graph_derivative,
                     graph_derivative, GL_STATIC_DRAW);

                // Draw using the vertices in our vertex buffer object
                glBindBuffer(GL_ARRAY_BUFFER, vbo_derivative);

                glEnableVertexAttribArray(attribute_coord2d);
                glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
                    GL_FALSE, 0, 0);

                glDrawArrays(GL_LINE_STRIP, 0, 2000); //this draw the
                    function

                // Stop clipping
                glViewport(0, 0, window_width, window_height);
                glDisable(GL_SCISSOR_TEST);
                break;
        }
        glutSwapBuffers();
}

void special(int key, int x, int y) {
        switch (key) {
                case GLUT_KEY_F1:
                mode = 0;
                printf("Drawing the original function.\n");
                break;
                case GLUT_KEY_F2:
                mode = 1;
                printf("Drawing the derivative.\n");
                break;
                case GLUT_KEY_LEFT:
                offset_x -= 0.03;
                break;
                case GLUT_KEY_RIGHT:
                offset_x += 0.03;
                break;
```

```
                case GLUT_KEY_UP:
                scale_x *= 1.5;
                break;
                case GLUT_KEY_DOWN:
                scale_x /= 1.5;
                break;
                case GLUT_KEY_HOME:
                offset_x = 0.0;
                scale_x = 1.0;
                break;
        }

        glutPostRedisplay();
}

void free_resources() {
        glDeleteProgram(program);
}

int main(int argc, char *argv[]) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB);
        glutInitWindowSize(1280, 800);
        glutCreateWindow("Graphs of Function and Its Derivative");

        GLenum glew_status = glewInit();

        // Showing the symbolic derivative
        Symbolic x("x");
        Symbolic y, dy;
        y = (x*x*x)+ (x*x);
        cout << "Plot y = f(x) and its derivative " << y << endl;
        cout << "y = " << y << endl;
        dy = df(y, x);
        cout << "derivative of y = " << dy << endl;
        cout << endl;

        if (GLEW_OK != glew_status) {
                fprintf(stderr, "Error: %s\n", glewGetErrorString(
                    glew_status));
                return 1;
        }

        if (!GLEW_VERSION_2_0) {
                fprintf(stderr, "No support for OpenGL 2.0 found\n");
                return 1;
        }
```

```
                printf("Use left/right to move horizontally.\n");
                printf("Use up/down to change the horizontal scale.\n");
                printf("Press home to reset the position and scale.\n");
                printf("Press F1 to draw the original function.\n");
                printf("Press F2 to draw the derivative.\n");

                if (init_resources()) {
                        glutDisplayFunc(display);
                        glutSpecialFunc(special);
                        glutMainLoop();
                }

                free_resources();
                return 0;
        }
```

**C++ Code 271:** *main.cpp "2D Plot of Derivative of Univariate Function"*

```
LDLIBS=-lm -lglut -lGLEW -lGL -lsymbolicc++ -lboost_iostreams
all: graph
clean:
rm -f *.o graph
graph: ../common/shader_utils.o
.PHONY: all clean
```

**C++ Code 272:** *Makefile "2D Plot of Derivative of Univariate Function"*

To compile it, type:
**g++ -o result main.cpp -lm -lglut -lGLEW -lGL -lsymbolicc++ -lboost_iostreams**
**./result**

or with the Makefile, type:
**make**
**./graph**

To check the compiling time you can type:
**time make**

Explanation for the codes:

- When you press F1 it will draw the original univariate function which is

$$f(x) = x^3 + x^2$$

when you press F2 it will draw the derivative of $f(x)$

$$f'(x) = \frac{df(x)}{dx} = 3x^2 + 2x$$

Inside the function **void display()** we define from

```
            switch (mode)
            {
                    case 0:
                    glDrawArrays(GL_LINE_STRIP, 0, 2000);
                    break;
                    case 1:
                    glClearColor(1, 1, 1, 1);
                    glClear(GL_COLOR_BUFFER_BIT);

                    glViewport(border + ticksize, border + ticksize,
                        window_width - border * 2 - ticksize, window_height
                         - border * 2 - ticksize);

                    glScissor(border + ticksize, border + ticksize,
                        window_width - border * 2 - ticksize, window_height
                         - border * 2 - ticksize);

                    glEnable(GL_SCISSOR_TEST);

                    // Set our coordinate transformation matrix
                    glm::mat4 transform = glm::translate(glm::scale(glm::
                        mat4(1.0f), glm::vec3(scale_x, 1, 1)), glm::vec3(
                        offset_x, 0, 0));
                    glUniformMatrix4fv(uniform_transform, 1, GL_FALSE, glm::
                        value_ptr(transform));

                    // Set the color to green
                    GLfloat green[4] = { 0, 1, 0, 1 };
                    glUniform4fv(uniform_color, 1, green);

                    glGenBuffers(1, &vbo_derivative);
                    glBindBuffer(GL_ARRAY_BUFFER, vbo_derivative);

                    // Create our own temporary buffer
                    point graph_derivative[2000];

                    // Fill it in just like an array
                    for (int i = 0; i < 2000; i++)
                    {
                            float var_x = (i - 1000) / 10.0;
                            double const phi{var_x};
                            auto const x{::boost::math::differentiation::
                                make_fvar<double, 5>(phi)}; // to find
                                derivative till order 5
                            auto const dx{get_x_coordinate(x*x*x + x*x)}; //
                                we define the function f(x) here :
                                get_x_coordinate(f(x))
```

```
                graph_derivative[i].x = var_x;
                graph_derivative[i].y = dx.derivative(1);
            }
            // Tell OpenGL to copy our array to the buffer object
            glBufferData(GL_ARRAY_BUFFER, sizeof graph_derivative,
                graph_derivative, GL_STATIC_DRAW);

            // Draw using the vertices in our vertex buffer object
            glBindBuffer(GL_ARRAY_BUFFER, vbo_derivative);

            glEnableVertexAttribArray(attribute_coord2d);
            glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT,
                GL_FALSE, 0, 0);

            glDrawArrays(GL_LINE_STRIP, 0, 2000); //this draw the
                function

            // Stop clipping
            glViewport(0, 0, window_width, window_height);
            glDisable(GL_SCISSOR_TEST);
            break;
        }
```

the codes above tell what would happens when we press F1 (case 0) or when we press F2 (case 1), this definition of events is the consequences of the **switch(key)** inside the function **void special(int key, int x, int y)**

- To draw the default / first graph it is defined inside the **int init_resources()**, we call the vertex and fragment shader files then build the Vertex Buffer Object, the first VBO will be used to create the graph of the original function $f(x) = x^3 + x^2$.

  Vertex buffer objects (VBOs) are just buffer objects that hold vertex data. It is very similar to using vertex arrays. First, OpenGL will allocate and deallocate the storate space for us. Second, we must explicitly tell OpenGL when we want access to the VBO. The idea is that when we don't want to access the VBO ourself, the GPU can have exclusive access to the contents of the VBO, and can even store the contents in its own memory, so it doesn't need to fetch the data from the slow main memory every time it needs the vertices.

  An alternative way to access the data that is in a VBO instead of telling OpenGL to copy data from our own memory to the graphics card, we can ask OpenGL to map the VBO into main memory. Depending on the graphics card, this might avoid the need to perform a copy, so it could be faster. On the other hand, the mapping itself could be expensive, or it might not really map anything but perform a copy anyway.

```
        glGenBuffers(3, vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

        // Create our own temporary buffer
        point graph[2000];
```

```
// Fill it in just like an array
for (int i = 0; i < 2000; i++)
{
	float x = (i - 1000) / 10.0;

	graph[i].x = x;
	graph[i].y = (x*x*x) + (x*x);
}

// Tell OpenGL to copy our array to the buffer object
glBufferData(GL_ARRAY_BUFFER, sizeof graph, graph,
    GL_STATIC_DRAW);
```

- At first the vertex arrays and shaders look like a pain to work compared to the old immediate mode and fixed rendering pipeline. However, in the end, especially if you are using buffer objects, your code will be much cleaner and the graphics will be faster.
- The vertex shader **graph.v.glsl** is a GLSL program that we use to pass each vertex individually and it will compute their on-screen (2D) coordinates.

```
attribute vec2 coord2d;
uniform mat4 transform;

void main(void) {
	gl_Position = transform * vec4(coord2d.xy, 0, 1);
}
```

The fragment shader **graph.v.glsl** is a GLSL program that we use to make OpenGL pass each pixel that is contained in our object, and it will compute its color.

```
uniform vec4 color;

void main(void) {
	gl_FragColor = color;
}
```

**Figure 25.13:** *The plot of the original function $f(x) = x^3 + x^2$ with OpenGL, GLEW, GLUT in C++ (**DFSimulator-C/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/2D Plot Derivative of Univariate Function/main.cpp**).*

**Figure 25.14:** *When you press F2 it will show the plot of* $f'(x) = \frac{df(x)}{dx} = 3x^2 + 2x$ *with C++ (**DFSimulatorC/-Source Codes/C++/C+++ Gnuplot SymbolicC++/ch25-Numerical Methods/2D Plot Derivative of Univariate Function/main.cpp**).*

# V. Numerical Integration

[DF*]

## i. C++ Computation:

**C++ Code 273:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# VI. Initial-Value Problems for Ordinary Differential Equations

[DF*]

## i. C++ Computation:

**C++ Code 274:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# VII. Approximation Theory

[DF*]

## i. C++ Computation:

**C++ Code 275:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# VIII. Numerical Solutions of Nonlinear Systems of Equations

[DF*]

### i. C++ Computation:

**C++ Code 276:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

## IX. Boundary-Value Problems for Ordinary Differential Equations

[DF*]

### i. C++ Computation:

**C++ Code 277:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

## X. Numerical Solutions to Partial Differential Equations

[DF*]

### i. C++ Computation:

**C++ Code 278:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 26

# DFSimulatorC++ XX: Complex Numbers

*"Tobikata Wo Wasureta Tori No You Ni (A Little bird who forgot how to fly)." - **Star Ocean 3 OST - MISIA***

## I. Preliminaries

[DF*] Complex analysis, traditionally known as the theory of functions of a complex variable, is the branch of mathematical analysis that investigates functions of complex numbers. It is helpful in many branches of mathematics, including algebraic geometry, number theory, analytic combinatorics, applied mathematics; as well as in physics, including the branches of hydrodynamics, thermodynamics, and particularly quantum mechanics. By extension, use of complex analysis also has applications in engineering fields such as nuclear, aerospace, mechanical and electrical engineering.

As a differentiable function of a complex variable is equal to its Taylor series (that is, it is analytic), complex analysis is particularly concerned with analytic functions of a complex variable (that is, holomorphic functions).

[DF*] Complex number arises when no one on this planet up until 2024 A.D. able to compute this:

$$\sqrt{-1}$$

That number is called imaginary number, the notation is $i = \sqrt{-1s}$.

[DF*] Let $x, y \in \mathbb{R}$, the complex number $z \in \mathbb{C}$ can be defined as

$$z = x + iy \tag{26.1}$$

Some important properties of complex numbers:

- $\mathrm{Re}(z) = x$ and $\mathrm{Im}(z) = y$ are called the real part of $z$ and the imaginary part of $z$, respectively.
- $|z| = \sqrt{x^2 + y^2}$ is called the modulus (or absolute value) of $z$.
- $\overline{z} = x - yi$ is called the complex conjugate of $z$.
- $z\overline{z} = x^2 + y^2 = |z|^2$
- The angle $\theta$ is called an argument of $z$.

**Figure 26.1:** *The complex number can be represented as $z = x + iy = re^{i\theta}$, with $r = |z|$ and the angle $\theta$ is the argument of $z$.*

- $\text{Re}(z) = |z| \cos\ \theta$
- $\text{Im}(z) = |z| \sin\ \theta$
- $z = |z|(\cos\ \theta + i\ \sin\ \theta)$ is called the polar form of $z$.

[DF*] We try to apply Newton-Raphson method to compute the root of $x^2 + 1 = 0$ and it cannot even compute $i$.

## i. C++ Plot: 2D Plot of Complex Function

Suppose we have a complex function

$$e^{ix}$$

and we want to plot it.

Note that

$$e^{ix} = \cos\ x + i\ \sin\ x$$

With **gnuplot** we can plot the function as separate functions, the first function is the real part, the second function is the imaginary part.

Thus, the real part of $e^{ix}$ is $\cos\ x$ and the imaginary part of $e^{ix}$ is $\sin\ x$.

```
#include <vector>
#include <cmath>
#include <utility>
#include <boost/tuple/tuple.hpp>

#include "gnuplot-iostream.h"

int main() {
        Gnuplot gp;

        // Don't forget to put "\n" at the end of each line!
        gp << "set xrange [-10:5]\nset yrange [-3:3]\n";
```

```
f(x) = 1+x^2

f'(x) = 2*x

p_{0} = 0.7853982

     n          p_{n}                    p_{n} in Degree
     0          0.7853982                45.0
     1          -0.24392074              -13.975629
     2          1.9278858                110.45972
     3          0.7045915                40.37012
     4          -0.35733533              -20.473806
     5          1.2205782                69.93398
     6          0.20064712               11.496233
     7          -2.3916135               -137.02936
     8          -0.98674273              -56.536194
     9          0.013346314              0.7646875
    10          -37.456852               -2146.1196
    11          -18.715078               -1072.295
    12          -9.330823                -534.61676
    13          -4.6118255               -264.23813
    14          -2.1974957               -125.907234
    15          -0.8712162               -49.91701
    16          0.13830233               7.9241395
    17          -3.5461168               -203.17754
    18          -1.6320591               -93.5101
    19          -0.509668                -29.201826
    20          0.72619677               41.60801
    21          -0.32542026              -18.645208
    22          1.3737646                78.710915
    23          0.3229189                18.50189
    24          -1.3869171               -79.4645
    25          -0.33294678              -19.076445
    26          1.3352681                76.505226
    27          0.2931775                16.797832
    28          -1.5588628               -89.31626
    29          -0.4586848               -26.280704
    30          0.8607309                49.316246
    31          -0.15053618              -8.625088
    32          3.2461925                185.99313
    33          1.4690697                84.17149
    34          0.3941834                22.585045
    35          -1.0713533               -61.384026
    36          -0.06897712              -3.9520977
    37          7.2142925                413.3485
    38          3.5378394                202.70326
    39          1.6275904                93.25407
    40          0.50659263               29.02562
    41          -0.73369                 -42.037342
```

**Figure 26.2:** *The Newton-Raphson method to compute the root of* $x^2 + 1 = 0$ *cannot converge.*

```
        // '—' means read from stdin. The send1d() function sends data to
            gnuplot's stdin.
        gp << "i = {0,1}\n";
        gp << "plot real(exp(i*x)), imag(exp(i*x))\n";

        // To do parametric plot uncomment below and comment the plot line
            above
        //gp << "set parametric\n";
        //gp << "plot real(exp(i*t)), imag(exp(i*t))\n";

        return 0;
    }
```

**C++ Code 279:** *main.cpp "2D plot of complex function"*

To compile it, type:
**g++ -o main main.cpp -lboost_iostreams**
**./main**

or with Makefile, type:
**make**
**./main**
s

Explanation for the codes:

- In **gnuplot** complex constants are expressed as $\{< \text{real} >, < \text{imag} >\}$, where $< \text{real} >$ and $< \text{imag} >$ must be numerical constants. For instance:

$$a + bi \to \{a, b\}$$
$$i \to \{0, 1\}$$
$$5 - 2i \to \{5, -2\}$$

  **Gnuplot** can plot only real quantities. Thus we can plot the real part of $e^{ix}$ which is cos $x$ and the imaginary part of $e^{ix}$ which is sin $x$, but first we need to define the variable $i$ as $\{0, 1\}$.

```
        gp << "i = {0,1}\n";
        gp << "plot real(exp(i*x)), imag(exp(i*x))\n";
```

- If you want to do parametric plot with

$$(\text{real}(e^{ix}), \text{imag}(e^{ix}))$$

  you can uncomment the two lines at the bottom and comment the plot line above it

```
        //gp << "plot real(exp(i*x)), imag(exp(i*x))\n";

        // To do parametric plot uncomment below and comment the plot
            line above
        gp << "set parametric\n";
        gp << "plot real(exp(i*t)), imag(exp(i*t))\n";
```

**Figure 26.3:** *The 2D plot of the real part of $e^{ix}$ and the imaginary part of $e^{xi}$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Complex Function/main.cpp**).*

**Figure 26.4:** *The 2D parametric plot of $(real(e^{ix}), imag(e^{ix}))$ (**DFSimulatorC/Source Codes/C++/C+++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/2D Plot Complex Function/main.cpp**).*

# Chapter 27

# DFSimulatorC++ XXI: Introduction to Stochastic Processes

*"One side will make you taller, the other side will make you shorter" (Alice in Wonderland 1951) - **The Caterpillar***

In probability theory and related fields, a stochastic or random process is a mathematical object usually defined as a sequence of random variables, where the index of the sequence has the interpretation of time. Stochastic processes are widely used as mathematical models of systems and phenomena that appear to vary in a random manner. Examples include the growth of a bacterial population, an electrical current fluctuating due to thermal noise, or the movement of a gas molecule. Stochastic processes have applications in many disciplines such as biology, chemistry, ecology, neuroscience, physics, image processing, signal processing, control theory, information theory, computer science, and telecommunications. Furthermore, seemingly random changes in financial markets have motivated the extensive use of stochastic processes in finance.

Applications and the study of phenomena have in turn inspired the proposal of new stochastic processes. Examples of such stochastic processes include the Wiener process or Brownian motion process, used by Louis Bachelier to study price changes on the Paris Bourse, and the Poisson process, used by A. K. Erlang to study the number of phone calls occurring in a certain period of time. These two stochastic processes are considered the most important and central in the theory of stochastic processes, and were discovered repeatedly and independently, both before and after Bachelier and Erlang, in different settings and countries.

[DF*]

## i. C++ Computation:

**C++ Code 280:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 28

# DFSimulatorC++ XXII: Introduction to the Theory of Interest and Financial Mathematics

*"In my world everything would be what it wouldn't be, and what would be it wouldn't." (Alice in Wonderland 1951) - **Alice***

Mathematical finance, also known as quantitative finance and financial mathematics, is a field of applied mathematics, concerned with mathematical modeling of financial markets.

In general, there exist two separate branches of finance that require advanced quantitative techniques: derivatives pricing on the one hand, and risk and portfolio management on the other. Mathematical finance overlaps heavily with the fields of computational finance and financial engineering. The latter focuses on applications and modeling, often by help of stochastic asset models, while the former focuses, in addition to analysis, on building tools of implementation for the models. Also related is quantitative investing, which relies on statistical and numerical models (and lately machine learning) as opposed to traditional fundamental analysis when managing portfolios.

French mathematician Louis Bachelier's doctoral thesis, defended in 1900, is considered the first scholarly work on mathematical finance. But mathematical finance emerged as a discipline in the 1970s, following the work of Fischer Black, Myron Scholes and Robert Merton on option pricing theory. Mathematical investing originated from the research of mathematician Edward Thorp who used statistical methods to first invent card counting in blackjack and then applied its principles to modern systematic investing

[DF*]

## i. C++ Computation:

**C++ Code 281:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 29

# DFSimulatorC++ XXIII: PDE - Wave Equation

*"LAM VAM RAM YAM HAM OM AH." - **7 Chakras Chanting***

[DF*]

i. C++ Computation:

**C++ Code 282:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 30

# DFSimulatorC++ XXIV: PDE - Heat Equation

*"Through the fire, to the limit, to the wall, for a chance to be with you, I gladly risk it all. Through the fire, to whatever come one day, for a chance of loving you, I'll take it all away. Right down through the wire, even through the fire." (Through the Fire song) - **Chaka Khan***

If you ever played Suikoden III then you must have known the story when Flame Champion sealed his True Fire Rune to be mortal again, it causes a great fire burning all Harmonian and Grasslands armies till perish. Why release his tremendous power? For Love, like Captain America, it seems Love can bring higher power than True Rune and immortality. This heat equation is one of the most famous partial differential equation that I will try to simulate with C++, the applications are tremendous, from Black-Scholes equation derived from this, create a long lasting ice cream, to be able to create better computer components will need the comprehension of this equation as well.

[DF*]

## i. C++ Computation:

**C++ Code 283:** *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- 
- 

# Chapter 31

# DFSimulatorC++ XXX: Plotting Physical Systems with Gnuplot

*"Puis, sous les yeux des disciples, Jesus s'eleve, une nuee l'emportee au ciel et il s'assoit a la droite de Dieu, le Pere, Tout-Puissant." - **Deuxieme Mysteres Glorieux (L'Ascension de Jesus au ciel)***

THis chapter will explains how to do various plotting with gnuplot. When we have made simulations over various physics problems or have system of differential equations, we would like to see the relation or connection between certain function toward another function, or certain variable toward another variable of interest to see the pattern, just like how Hooke's law is established after measuring the relation between the displacement of the mass attached to the spring toward the spring' force, thus able to determine the spring constant. This chapter can come in handy, when you have the data that only needs direct plotting with a bit of numerical manipulation, no need to process it symbolically anymore.

All the files and data used in this chapter can be found in this book repository:

**DFSimulatorC/Source Codes/C++/Gnuplot/**

[DF*] **Polar Plot**

Open gnuplot and type:

```
set polar
plot t*sin(t)
plot [-2*pi:2*pi] [-5:5] [-5:5] t*sin(t)
```

**Figure 31.1:** *Polar plot with gnuplot.*

```
set polar
set trange [0:12*pi]
plot 2*t
```

**Figure 31.2:** *Spiral plot with gnuplot.*

[DF*] **Histogram Plot**

Create a file or prepare a file with one column / one vector array of data, name it

'**pendulumdata**', this pendulum data is retrieved from Box2D simulation on Simple pendulum, it is taking the data of the angle of the pendulum, and we want to create the histogram of it.

```
0.00
0.00
0.04
0.21
0.47
0.85
1.24
1.66
2.12
2.61
...
```

**C++ Code 284:** *pendulumdata*

Create another file called **histogramplot** (this is the file containing command for plotting in gnuplot, so you don't need to type one by one anymore, just edit in text editor and call it) in the working directory.

```
binwidth=5
set boxwidth binwidth
bin(x,width)=width*floor(x/width)
plot 'pendulumdata' using (bin($1,binwidth)):(1.0) smooth freq
    with boxes lc rgb"blue" title 'Pendulum angle'
```

**C++ Code 285:** *histogramplot*

Then open gnuplot and type:
**load 'histogramplot'**

**Figure 31.3:** *Histogram plot for the pendulum angle data.*

If you want to add some styling you can try this:

```
binwidth=5
set boxwidth binwidth
bin(x,width)=width*floor(x/width)
n=100 #number of intervals
max=3. #max value
min=-3. #min value
width=(max-min)/n #interval width

#function used to map a value to the intervals
hist(x,width)=width*floor(x/width)+width/2.0
set boxwidth width*5.9
set style fill solid 0.5 # fill style
plot 'pendulumdata' using (bin($1,binwidth)):(1.0) smooth freq
    with boxes title 'Pendulum angle'
```

**C++ Code 286:** *histogramplot*

[DF*] **Probability Density Function Plot**
Open gnuplot and type:
**load 'probabilitydensityfunction-normal'**
the code can be seen below or you can also get it from the repository.

```
#!/usr/local/bin/gnuplot -persist
# set terminal pngcairo transparent enhanced font "arial,10" fontscale
    1.0 size 600, 400
# set output 'prob.39.png'
set format x "%.1f"
set format y "%.1f"
set key fixed left top vertical Right noreverse enhanced
    autotitle box lt black linewidth 1.000 dashtype solid
unset parametric
set style data lines
set xzeroaxis
set yzeroaxis
set zzeroaxis
set title "normal (also called Gauss or bell-curved) PDF"
set trange [ -0.200000 : 3.20913 ] noreverse nowriteback
set xlabel "x"
set xrange [ 0.00000 : 2.50000 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "probability density"
set yrange [ 0.00000 : 1.10000 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zrange [ * : * ] noreverse writeback
set cbrange [ * : * ] noreverse writeback
set rrange [ * : * ] noreverse writeback
set colorbox vertical origin screen 0.9, 0.2 size screen 0.05,
    0.6 front noinvert bdefault
```

```
isint(x)=(int(x)==x)
Binv(p,q)=exp(lgamma(p+q)-lgamma(p)-lgamma(q))
arcsin(x,r)=r<=0?1/0:abs(x)>r?0.0:invpi/sqrt(r*r-x*x)
beta(x,p,q)=p<=0||q<=0?1/0:x<0||x>1?0.0:Binv(p,q)*x**(p-1.0)
    *(1.0-x)**(q-1.0)
binom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0||x>n?0.0:exp(lgamma(n+1)-lgamma(n-x+1)-lgamma(x+1) +x*
    log(p)+(n-x)*log(1.0-p))
cauchy(x,a,b)=b<=0?1/0:b/(pi*(b*b+(x-a)**2))
chisq(x,k)=k<=0||!isint(k)?1/0: x<=0?0.0:exp((0.5*k-1.0)*log(x
    )-0.5*x-lgamma(0.5*k)-k*0.5*log2)
erlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0: x<0?0.0:x
    ==0?(n==1?real(lambda):0.0):exp(n*log(lambda)+(n-1.0)*log(
    x)-lgamma(n)-lambda*x)
extreme(x,mu,alpha)=alpha<=0?1/0:alpha*(exp(-alpha*(x-mu)-exp
    (-alpha*(x-mu))))
f(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0: Binv(0.5*
    d1,0.5*d2)*(real(d1)/d2)**(0.5*d1)*x**(0.5*d1-1.0)/(1.0+(
    real(d1)/d2)*x)**(0.5*(d1+d2))
gmm(x,rho,lambda)=rho<=0||lambda<=0?1/0: x<0?0.0:x==0?(rho
    >1?0.0:rho==1?real(lambda):1/0): exp(rho*log(lambda)+(rho
    -1.0)*log(x)-lgamma(rho)-lambda*x)
geometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==1?(x
    ==0?1.0:0.0):exp(log(p)+x*log(1.0-p))
halfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:sqrt2invpi/sigma*exp
    (-0.5*(x/sigma)**2)
hypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x>d||x>C||x<0||x<d-(N-C)
    ?0.0:exp(lgamma(C+1)-lgamma(C-x+1)-lgamma(x+1)+ lgamma(N-
    C+1)-lgamma(d-x+1)-lgamma(N-C-d+x+1)+lgamma(N-d+1)+
    lgamma(d+1)-lgamma(N+1))
laplace(x,mu,b)=b<=0?1/0:0.5/b*exp(-abs(x-mu)/b)
logistic(x,a,lambda)=lambda<=0?1/0:lambda*exp(-lambda*(x-a))
    /(1.0+exp(-lambda*(x-a)))**2
lognormal(x,mu,sigma)=sigma<=0?1/0: x<0?0.0:invsqrt2pi/sigma/x*
    exp(-0.5*((log(x)-mu)/sigma)**2)
maxwell(x,a)=a<=0?1/0:x<0?0.0:fourinvsqrtpi*a**3*x*x*exp(-a*a*
    x*x)
negbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
    <0?0.0:p==1?(x==0?1.0:0.0):exp(lgamma(r+x)-lgamma(r)-
    lgamma(x+1)+ r*log(p)+x*log(1.0-p))
nexp(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*exp(-lambda*x)
normal(x,mu,sigma)=sigma<=0?1/0:invsqrt2pi/sigma*exp(-0.5*((x-
    mu)/sigma)**2)
pareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0:real(b)/x*(real(a)
    /x)**b
poisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:exp(x*log(mu)-
    lgamma(x+1)-mu)
```

```
rayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*2.0*x*exp(-
    lambda*x*x)
sine(x,f,a)=a<=0?1/0: x<0||x>=a?0.0:f==0?1.0/a:2.0/a*sin(f*pi*x
    /a)**2/(1-sin(twopi*f))
t(x,nu)=nu<0||!isint(nu)?1/0: Binv(0.5*nu,0.5)/sqrt(nu)*(1+real
    (x*x)/nu)**(-0.5*(nu+1.0))
triangular(x,m,g)=g<=0?1/0:x<m-g||x>=m+g?0.0:1.0/g-abs(x-m)/
    real(g*g)
uniform(x,a,b)=x<(a<b?a:b)||x>=(a>b?a:b)?0.0:1.0/abs(b-a)
weibull(x,a,lambda)=a<=0||lambda<=0?1/0: x<0?0.0:x==0?(a>1?0.0:
    a==1?real(lambda):1/0): exp(log(a)+a*log(lambda)+(a-1)*log
    (x)-(lambda*x)**a)
carcsin(x,r)=r<=0?1/0:x<-r?0.0:x>r?1.0:0.5+invpi*asin(x/r)
cbeta(x,p,q)=ibeta(p,q,x)
cbinom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0?0.0:x>=n?1.0:ibeta(n-x,x+1.0,1.0-p)
ccauchy(x,a,b)=b<=0?1/0:0.5+invpi*atan((x-a)/b)
cchisq(x,k)=k<=0||!isint(k)?1/0:x<0?0.0:igamma(0.5*k,0.5*x)
cerlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0:x<0?0.0:
    igamma(n,lambda*x)
cextreme(x,mu,alpha)=alpha<=0?1/0:exp(-exp(-alpha*(x-mu)))
cf(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0:1.0-ibeta
    (0.5*d2,0.5*d1,d2/(d2+d1*x))
cgmm(x,rho,lambda)=rho<=0||lambda<=0?1/0:x<0?0.0:igamma(rho,x*
    lambda)
cgeometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==0?0.0:p
    ==1?1.0:1.0-exp((x+1)*log(1.0-p))
chalfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:erf(x/sigma/sqrt2)
chypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x<0||x<d-(N-C)?0.0:x>d||x>C
    ?1.0:hypgeo(x,N,C,d)+chypgeo(x-1,N,C,d)
claplace(x,mu,b)=b<=0?1/0:(x<mu)?0.5*exp((x-mu)/b):1.0-0.5*exp
    (-(x-mu)/b)
clogistic(x,a,lambda)=lambda<=0?1/0:1.0/(1+exp(-lambda*(x-a)))
clognormal(x,mu,sigma)=sigma<=0?1/0:x<=0?0.0:cnormal(log(x),mu,
    sigma)
cnormal(x,mu,sigma)=sigma<=0?1/0:0.5+0.5*erf((x-mu)/sigma/
    sqrt2)
cmaxwell(x,a)=a<=0?1/0:x<0?0.0:igamma(1.5,a*a*x*x)
cnegbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
    <0?0.0:ibeta(r,x+1,p)
cnexp(x,lambda)=lambda<=0?1/0:x<0?0.0:1-exp(-lambda*x)
cpareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0.0:1.0-(real(a)/x
    )**b
cpoisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:1-igamma(x+1.0,
    mu)
crayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:1.0-exp(-lambda*x*x)
csine(x,f,a)=a<=0?1/0: x<0?0.0:x>a?1.0:f==0?real(x)/a:(real(x)/
```

```
    a-sin(f*twopi*x/a)/(f*twopi))/(1.0-sin(twopi*f)/(twopi*f)
    )
ct(x,nu)=nu<0||!isint(nu)?1/0:0.5+0.5*sgn(x)*(1-ibeta(0.5*nu
    ,0.5,nu/(nu+x*x)))
ctriangular(x,m,g)=g<=0?1/0: x<m-g?0.0:x>=m+g?1.0:0.5+real(x-m
    )/g-(x-m)*abs(x-m)/(2.0*g*g)
cuniform(x,a,b)=x<(a<b?a:b)?0.0:x>=(a>b?a:b)?1.0:real(x-a)/(b-
    a)
cweibull(x,a,lambda)=a<=0||lambda<=0?1/0:x<0?0.0:1.0-exp(-(
    lambda*x)**a)
gsampfunc(t,n) = t<0?0.5*1/(-t+1.0)**n:1.0-0.5*1/(t+1.0)**n
NO_ANIMATION = 1
fourinvsqrtpi = 2.25675833419103
invpi = 0.318309886183791
invsqrt2pi = 0.398942280401433
log2 = 0.693147180559945
sqrt2 = 1.4142135623731
sqrt2invpi = 0.797884560802865
twopi = 6.28318530717959
save_encoding = "utf8"
eps = 1e-10
xmin = -4.0
xmax = 4.0
ymin = -5
ymax = 0.877673016883152
r = 8
mu = 2.0
sigma = 0.5
p = 0.4
q = 3.0
n = 2
a = 1.5
b = 1.0
k = 2
lambda = 2.0
l1 = 1.0
l2 = 0.5
alpha = 1.0
u = 0.0
df1 = 5.0
df2 = 9.0
rho = 6.0
s = 4.66878227507107
N = 75
C = 25
d = 10
m = 3.08021684891803
plot [xmin:xmax] [0:ymax] normal(x, 0, 1.0) title "mean = 0,
```

```
sigma = 1.0", normal(x, 2, 0.5) title "mean = 2, sigma =
0.5", normal(x, 1, 2.0) title "mean = 1, sigma = 2.0"
```

**C++ Code 287:** *probabilitydensityfunction-normal*

The plotting command is located on the last line, while the lines at the middle starting from **Binv(p,q)** till **gsampfunc(t,n)** are the function definitions of each of probability density functions available that we can plot, besides normal there are chi-square, weibull, poisson, pareto, logistic, each with different parameters. Probability and Statistics are necessary especially in experimental physics we would like to know about the profile of the data, especially if we only do experimental physics with limited sample, like try to swing double pendulum at angle of $\pi/6$ for 5 minutes, compared with angle of $\pi/8$ for 10 minutes, they are all only samples of experimentation which the result of the data then can be processed with statistics and probability methods.

**Figure 31.4:** *The probability density function for normal or bell-curved with certain parameters of $\mu$ and $\sigma$.*

[DF*] **Cumulative Density Function Plot**

Open gnuplot and type:

**load 'cumulativedistributionfunction-normal**

the code can be seen below or you can also get it from the repository.

```
#!/usr/local/bin/gnuplot -persist
# set terminal pngcairo transparent enhanced font "arial,10" fontscale
    1.0 size 600, 400
# set output 'prob.39.png'
set format x "%.1f"
set format y "%.1f"
set key fixed left top vertical Right noreverse enhanced
    autotitle box lt black linewidth 1.000 dashtype solid
unset parametric
set style data lines
set xzeroaxis
set yzeroaxis
set zzeroaxis
```

```
set title "normal (also called Gauss or bell-curved) CDF"
set trange [ -0.200000 : 3.20913 ] noreverse nowriteback
set xlabel "x"
set xrange [ 0.00000 : 2.50000 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "cumulative distribution"
set yrange [ 0.00000 : 1.10000 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zrange [ * : * ] noreverse writeback
set cbrange [ * : * ] noreverse writeback
set rrange [ * : * ] noreverse writeback
set colorbox vertical origin screen 0.9, 0.2 size screen 0.05,
    0.6 front noinvert bdefault
isint(x)=(int(x)==x)
Binv(p,q)=exp(lgamma(p+q)-lgamma(p)-lgamma(q))
arcsin(x,r)=r<=0?1/0:abs(x)>r?0.0:invpi/sqrt(r*r-x*x)
beta(x,p,q)=p<=0||q<=0?1/0:x<0||x>1?0.0:Binv(p,q)*x**(p-1.0)
    *(1.0-x)**(q-1.0)
binom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0||x>n?0.0:exp(lgamma(n+1)-lgamma(n-x+1)-lgamma(x+1) +x*
    log(p)+(n-x)*log(1.0-p))
cauchy(x,a,b)=b<=0?1/0:b/(pi*(b*b+(x-a)**2))
chisq(x,k)=k<=0||!isint(k)?1/0: x<=0?0.0:exp((0.5*k-1.0)*log(x
    )-0.5*x-lgamma(0.5*k)-k*0.5*log2)
erlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0: x<0?0.0:x
    ==0?(n==1?real(lambda):0.0):exp(n*log(lambda)+(n-1.0)*log(
    x)-lgamma(n)-lambda*x)
extreme(x,mu,alpha)=alpha<=0?1/0:alpha*(exp(-alpha*(x-mu)-exp
    (-alpha*(x-mu))))
f(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0: Binv(0.5*
    d1,0.5*d2)*(real(d1)/d2)**(0.5*d1)*x**(0.5*d1-1.0)/(1.0+(
    real(d1)/d2)*x)**(0.5*(d1+d2))
gmm(x,rho,lambda)=rho<=0||lambda<=0?1/0: x<0?0.0:x==0?(rho
    >1?0.0:rho==1?real(lambda):1/0): exp(rho*log(lambda)+(rho
    -1.0)*log(x)-lgamma(rho)-lambda*x)
geometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==1?(x
    ==0?1.0:0.0):exp(log(p)+x*log(1.0-p))
halfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:sqrt2invpi/sigma*exp
    (-0.5*(x/sigma)**2)
hypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x>d||x>C||x<0||x<d-(N-C)
    ?0.0:exp(lgamma(C+1)-lgamma(C-x+1)-lgamma(x+1)+ lgamma(N-
    C+1)-lgamma(d-x+1)-lgamma(N-C-d+x+1)+lgamma(N-d+1)+
    lgamma(d+1)-lgamma(N+1))
laplace(x,mu,b)=b<=0?1/0:0.5/b*exp(-abs(x-mu)/b)
logistic(x,a,lambda)=lambda<=0?1/0:lambda*exp(-lambda*(x-a))
    /(1.0+exp(-lambda*(x-a)))**2
lognormal(x,mu,sigma)=sigma<=0?1/0: x<0?0.0:invsqrt2pi/sigma/x*
```

```
    exp(-0.5*((log(x)-mu)/sigma)**2)
maxwell(x,a)=a<=0?1/0:x<0?0.0:fourinvsqrtpi*a**3*x*x*exp(-a*a*
    x*x)
negbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
    <0?0.0:p==1?(x==0?1.0:0.0):exp(lgamma(r+x)-lgamma(r)-
    lgamma(x+1)+ r*log(p)+x*log(1.0-p))
nexp(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*exp(-lambda*x)
normal(x,mu,sigma)=sigma<=0?1/0:invsqrt2pi/sigma*exp(-0.5*((x-
    mu)/sigma)**2)
pareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0:real(b)/x*(real(a)
    /x)**b
poisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:exp(x*log(mu)-
    lgamma(x+1)-mu)
rayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*2.0*x*exp(-
    lambda*x*x)
sine(x,f,a)=a<=0?1/0: x<0||x>=a?0.0:f==0?1.0/a:2.0/a*sin(f*pi*x
    /a)**2/(1-sin(twopi*f))
t(x,nu)=nu<0||!isint(nu)?1/0: Binv(0.5*nu,0.5)/sqrt(nu)*(1+real
    (x*x)/nu)**(-0.5*(nu+1.0))
triangular(x,m,g)=g<=0?1/0:x<m-g||x>=m+g?0.0:1.0/g-abs(x-m)/
    real(g*g)
uniform(x,a,b)=x<(a<b?a:b)||x>=(a>b?a:b)?0.0:1.0/abs(b-a)
weibull(x,a,lambda)=a<=0||lambda<=0?1/0: x<0?0.0:x==0?(a>1?0.0:
    a==1?real(lambda):1/0): exp(log(a)+a*log(lambda)+(a-1)*log
    (x)-(lambda*x)**a)
carcsin(x,r)=r<=0?1/0:x<-r?0.0:x>r?1.0:0.5+invpi*asin(x/r)
cbeta(x,p,q)=ibeta(p,q,x)
cbinom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0?0.0:x>=n?1.0:ibeta(n-x,x+1.0,1.0-p)
ccauchy(x,a,b)=b<=0?1/0:0.5+invpi*atan((x-a)/b)
cchisq(x,k)=k<=0||!isint(k)?1/0:x<0?0.0:igamma(0.5*k,0.5*x)
cerlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0:x<0?0.0:
    igamma(n,lambda*x)
cextreme(x,mu,alpha)=alpha<=0?1/0:exp(-exp(-alpha*(x-mu)))
cf(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0:1.0-ibeta
    (0.5*d2,0.5*d1,d2/(d2+d1*x))
cgmm(x,rho,lambda)=rho<=0||lambda<=0?1/0:x<0?0.0:igamma(rho,x*
    lambda)
cgeometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==0?0.0:p
    ==1?1.0:1.0-exp((x+1)*log(1.0-p))
chalfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:erf(x/sigma/sqrt2)
chypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x<0||x<d-(N-C)?0.0:x>d||x>C
    ?1.0:hypgeo(x,N,C,d)+chypgeo(x-1,N,C,d)
claplace(x,mu,b)=b<=0?1/0:(x<mu)?0.5*exp((x-mu)/b):1.0-0.5*exp
    (-(x-mu)/b)
clogistic(x,a,lambda)=lambda<=0?1/0:1.0/(1+exp(-lambda*(x-a)))
clognormal(x,mu,sigma)=sigma<=0?1/0:x<=0?0.0:cnormal(log(x),mu,
```

```
    sigma)
cnormal(x,mu,sigma)=sigma<=0?1/0:0.5+0.5*erf((x-mu)/sigma/
    sqrt2)
cmaxwell(x,a)=a<=0?1/0:x<0?0.0:igamma(1.5,a*a*x*x)
cnegbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
    <0?0.0:ibeta(r,x+1,p)
cnexp(x,lambda)=lambda<=0?1/0:x<0?0.0:1-exp(-lambda*x)
cpareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0.0:1.0-(real(a)/x
    )**b
cpoisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:1-igamma(x+1.0,
    mu)
crayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:1.0-exp(-lambda*x*x)
csine(x,f,a)=a<=0?1/0: x<0?0.0:x>a?1.0:f==0?real(x)/a:(real(x)/
    a-sin(f*twopi*x/a)/(f*twopi))/(1.0-sin(twopi*f)/(twopi*f)
    )
ct(x,nu)=nu<0||!isint(nu)?1/0:0.5+0.5*sgn(x)*(1-ibeta(0.5*nu
    ,0.5,nu/(nu+x*x)))
ctriangular(x,m,g)=g<=0?1/0: x<m-g?0.0:x>=m+g?1.0:0.5+real(x-m
    )/g-(x-m)*abs(x-m)/(2.0*g*g)
cuniform(x,a,b)=x<(a<b?a:b)?0.0:x>=(a>b?a:b)?1.0:real(x-a)/(b-
    a)
cweibull(x,a,lambda)=a<=0||lambda<=0?1/0:x<0?0.0:1.0-exp(-(
    lambda*x)**a)
gsampfunc(t,n) = t<0?0.5*1/(-t+1.0)**n:1.0-0.5*1/(t+1.0)**n
NO_ANIMATION = 1
fourinvsqrtpi = 2.25675833419103
invpi = 0.318309886183791
invsqrt2pi = 0.398942280401433
log2 = 0.693147180559945
sqrt2 = 1.4142135623731
sqrt2invpi = 0.797884560802865
twopi = 6.28318530717959
save_encoding = "utf8"
eps = 1e-10
xmin = -4.0
xmax = 4.0
ymin = -5
ymax = 0.877673016883152
r = 8
mu = 2.0
sigma = 0.5
p = 0.4
q = 3.0
n = 2
a = 1.5
b = 1.0
k = 2
lambda = 2.0
```

```
l1 = 1.0
l2 = 0.5
alpha = 1.0
u = 0.0
df1 = 5.0
df2 = 9.0
rho = 6.0
s = 4.66878227507107
N = 75
C = 25
d = 10
m = 3.08021684891803
plot [xmin:xmax] [0:1.1] mu = 0, sigma = 1.0, cnormal(x, mu,
    sigma) title "mean = 0, sigma = 1.0", \
mu = 2, sigma = 0.5, cnormal(x, mu, sigma) title "mean = 2,
    sigma = 0.5", \
mu = 1, sigma = 2.0, cnormal(x, mu, sigma) title "mean = 1,
    sigma = 2.0"
```

**C++ Code 288:** *cumulativedistributionfunction-normal*

The code for the beginning and middle part are the same as the probability density function code, the only difference are the title, the label, and the function we use here is **cnormal** not **normal**.

**Figure 31.5:** *The cumulative distribution function for normal or bell-curved with certain parameters of $\mu$ and $\sigma$.*

[DF*] **Vector Field Plot**

When the quantity that depends on $x$ and $y$ has both a magnitude and a direction it can be represented by an arrow whose length is proportional to the magnitude. If we divide the $x - y$ plane into a grid and draw an arrow at each grid point representing the direction and magnitude at that point, we have a vector plot. As we are associating two quantities, the magnitude and direction(or, equivalently, $\triangle x$ and $\triangle y$) for each value of $x$ and $y$, we can think of this type of visualization as a $4D$ plot. Vector plots are used for representing fluid

flows, electric fields, and many other physical systems.

Create a file that store the data that we want to plot as vector field, it should be 2 columns of data. Name it **data**, the one provided is obtained from Box2D simulation of a kinetic work example for moving a cutting tool along $x$ axis for 46 meters. Then at the same working directory create a file name **vectorfieldplot**.

```
set xrange [-23:23]
set yrange [0:2]
# only integer x-coordinates
set samples 11
# only integer y-coordinates
set isosamples 11
# we need data, so we use the special filename "data", which
# produces x,y-pairs
set title "Vector Field"
plot "data" using 1:2:1:(2.*$2) with vectors title 'F(x,y) = <x
    , 2y> Kinetic Work '
```

**C++ Code 289:** *vectorfieldplot*

Open gnuplot and type:
**load 'vectorfieldplot'**

Another alternative to plot the vector field is

```
set xrange [-5:5]
set yrange [-5:5]
dx(x) = x
dy(x) = 2*x
set title 'Vector field F(x,y) = <x, 2y>'
plot "data" using 1:2:(dx($1)):(dy($2)) w vec title 'F(x,y) = <
    x, 2y>'
```

**C++ Code 290:** *vectorfieldplot1*

The third alternative to plot vector field on a plane is Another alternative to plot the vector field is

```
set xrange [-3*pi:3*pi]
set yrange [-pi:pi]
set iso 20
set samp 20
unset key
a = 2
plot "data" using 1:2:(-a*sin($1)*cos($2)):(a*cos($1)*sin($2))
     \
with vectors size .06, 15 filled
```

**C++ Code 291:** *vectorfieldplot2*

**Figure 31.6:** *Vector field plot of $F(x, y) =< x, 2y >$ from Kinetic Work data of $x$ and $y$ position of a cutting tool that is moved 46 meters.*

How to figure out the arrows' location, magnitude and direction:
**plot "data" using 1:2:(-a*sin($1)*cos($2)):(a*cos($1)*sin($2)) with vectors size .06, 15 filled**

with the general syntax should look like this:
**plot "data" using (first variable / $x$ location of the arrow):(second variable / $y$ location of the arrow):(function or constant to determine direction of each arrow): (function or constant to determine magnitude of each arrow)**
We are going to examine that command for gnuplot, here are the explanations:

- **using 1:2**
  Means the initial location / the tail of each arrow is $(x, y)$ with $x$ is the data from the first column(1) and $y$ is the data from the second column(2). If the initial location is like this **using 2:1**, then the tail will be $(y, x)$.
- **:(-a*sin($1)*cos($2)):**
  The direction of the arrow, the slope of the arrow. It is located in the middle after we determine the tail of the arrow, we determine the direction, it is a sine cosine function of the data, with $1 represents column 1 or $x$, and $2 represents column 2 or $y$, the direction for the $i$-th arrow will be toward $-a * sin(x_i) * cos(y_i)$, this explains why each arrow can have different direction.
- **:(a*cos($1)*sin($2))**
  The magnitude for each arrow, again it is a sine cosine function toward $x$ and $y$, this explains why each arrow may have different magnitude.

Basically, gnuplot can only plot vector fields when reading data from a file. In standard, your file will have to have 4 columns, x, y, delta x and delta y, and gnuplot will then plot a vector from (x,y) to (x+delta x, y+delta y) for each line in the file, then type in gnuplot:

**plot "file.dat" using 1:2:3:4 with vectors head filled lt 2**

**Figure 31.7:** *Vector field plot of* $F(x,y) =< -2sin(x)\ \cos(y), 2\cos(x)\ \sin(y) >$ *from Kinetic Work data of x and y position of a cutting tool that is moved 46 meters.*

But with modification like the examples above, we can also use 2 columns of data.

[DF*] **Parametric Plot**

Parametric plot is a plot where the $x$ and $y$ values each depend on a third variable, called a parameter. The set parametric command changes the meaning of plot (splot) from normal functions to parametric functions. For 3-d plotting, the surface is described as $x = f(u,v),\ y = g(u,v),\ z = h(u,v)$. Therefore a triplet of functions is required. Create a file named **parametricplot**:

```
unset key
set parametric
set samp 1000
set view 60, 45
set urange [0: 10]
splot u*cos(10*u), u*sin(10*u), u lw 3
```

**C++ Code 292:** *3dparametricplot-invertedcone*

Open gnuplot and type:
**load '3dparametricplot-invertedcone'**
Another example for plotting a sphere with parametric plot:

```
unset key
set parametric
set samp 1000
set view 60, 45
set xrange [—pi/2:pi/2]
set yrange [—pi/2:pi/2]
set xlabel 'x'
set ylabel 'y'
set zlabel 'z'
```

**Figure 31.8:** *Parametric plot of function* $(x, y) = (u * cos(10 * u), u * sin(10 * u), u)$ *with* $u = [0, 10]$ ***(DFSimulatorC/Source Codes/C++/Gnuplot/3dparametricplot-invertedcone)****.*

```
set urange [-10: 10]
set vrange [-10: 10]
splot cos(u)*cos(v),cos(u)*sin(v),sin(u) lw 3
```

**C++ Code 293:** *parametricplotsphere*

**Figure 31.9:** *Parametric plot of function* $(x, y, z) = \cos(u)\cos(v), \cos(u)\sin(v), \sin(u)$ *with* $u = [-10, 10]$, $v = [-10, 10]$ ***(DFSimulatorC/Source Codes/C++/Gnuplot/3dparametricplot-sphere)****.*

The command **splot** can display a surface as a collection of points, or by connecting those points, splot can interpolate points having a common amplitude with **set contour**. The surface can be made opaque with **set hidden3d**.

**Figure 31.10:** *Parametric plot of function* $(x, y, z) = 2 - v \sin(u/2)\sin(u), 2 - v \sin(u/2)\cos(u), v \cos(u/2)$ *, this function is known as Moebius strip (**DFSimulatorC/-Source Codes/C++/Gnuplot/3dparametricplot-moebiusstrip**).*

[DF*] **Contour Plot**

A contour plot traces value by drawing curves, just as in the topographical maps for the hikers. Create a file named **contourplot**.

```
set xrange [-4:4]
set yrange [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set cntrparam levels auto 9
set view map
splot sin(y**2+x**2) - cos(x**2) title "sin(x^{2} + y^{2}) -
    cos(x^{2})"
```

**C++ Code 294:** *contourplot*

Open gnuplot and type:
**load 'contourplot'**

**Figure 31.11:** *Contour plot of function* $\sin(x^2 + y^2) - \cos(x^2)$.

[DF*] **Complex Plot**

When we observe the physical phenomena we can plot them and see the relation thus create a formula out of it, there are more beyond the conservative relation, when you have learn modern physics and more advanced physics, there are relation between each variable that can only be explained with complex functions.

Create a file named **3dcomplexfunctionplot**, or you can retrieve it from the repository, the code can be seen below.

```
#!/usr/local/bin/gnuplot -persist
# set terminal pngcairo background "#ffffff" enhanced font "times"
    fontscale 1.0 size 640, 480
# set output 'complex_trig.11.png'
set border -1 front lt black linewidth 1.000 dashtype solid
set grid nopolar
set grid xtics nomxtics ytics nomytics noztics nomztics nortics
    nomrtics \
nox2tics nomx2tics noy2tics nomy2tics nocbtics nomcbtics
set grid layerdefault lt 0 linecolor 0 linewidth 0.500, lt 0
    linecolor 0 linewidth 0.500
unset key
unset parametric
set view 66, 336, 1.2, 1.2
set view equal xyz
set isosamples 100, 100
set size ratio 1 1,1
set style data lines
set xyplane at 0
set xtics norangelimit
```

```
set xtics ("-pi/2" -1.57080, "-pi/4" -0.785398, "0" 0.00000,
    "pi/4" 0.785398, "pi/2" 1.57080)
set ytics norangelimit
set ytics ("-pi/2" -1.57080, "-pi/4" -0.785398, "0" 0.00000,
    "pi/4" 0.785398, "pi/2" 1.57080)
unset ztics
set cbtics norangelimit
set cbtics ("0" -3.14159, "2pi" 3.14159)
set title "atanh( x + iy )"
set urange [ -1.57080 : 1.57080 ] noreverse nowriteback
set vrange [ -1.57080 : 1.57080 ] noreverse nowriteback
set xlabel "Real"
set xlabel offset character 0, -2, 0 font "" textcolor lt -1
    rotate parallel
set xrange [ -1.57080 : 1.57080 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "Imaginary"
set ylabel offset character 0, -2, 0 font "" textcolor lt -1
    rotate parallel
set yrange [ -1.57080 : 1.57080 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zlabel "magnitude"
set zlabel offset character 3, 0, 0 font "" textcolor lt -1
    rotate
set zrange [ * : * ] noreverse writeback
set cblabel "Phase Angle"
set cblabel offset character -2, 0, 0 font "" textcolor lt -1
    rotate
set cbrange [ -3.14159 : 3.14159 ] noreverse nowriteback
set rrange [ * : * ] noreverse writeback
set palette positive nops_allcF maxcolors 0 gamma 1.5 color
    model HSV
set palette defined ( 0 0 1 1, 1 1 1 1 )
set colorbox user
set colorbox vertical origin screen 0.85, 0.2 size screen 0.05,
     0.6 front noinvert bdefault
Hue(x,y) = (pi + atan2(-y,-x)) / (2*pi)
phase(x,y) = hsv2rgb( Hue(x,y), sqrt(x**2+y**2), 1. )
rp(x,y) = real(f(x,y))
f(x,y) = atanh(x + y*{0,1})
ip(x,y) = imag(f(x,y))
color(x,y) = hsv2rgb( Hue( rp(x,y), ip(x,y) ), abs(f(x,y)), 1.
    )
NO_ANIMATION = 1
save_encoding = "utf8"
## Last datafile plotted: "++"
splot '++' using 1:2:(abs(f($1,$2))):(color($1,$2)) with pm3d
    lc rgb variable
```

**C++ Code 295:** *3dcomplexfunctionplot*

Open gnuplot and type:
**load '3dcomplexfunctionplot'**

**Figure 31.12:** *3D plot of complex function* $\text{atan}\,(x + iy)$ *(**DFSimulatorC/Source Codes/C++/Gnuplot/3dcomplexfunctionplot**).*

**Figure 31.13:** *2D polar plot of complex function* $\operatorname{asin}(x + iy)$ *(**DFSimulatorC/Source Codes/C++/Gnuplot/2dpolarcomplexfunctionplot**).*

# Bibliography


[1] Boyce, William E., DiPrima, Richard C. (2010) Elementary Differential Equations and Boundary Value Problems 9th Edition, John Wiley & Sons, Hoboken, New Jersey, United States.

[2] Burden, Richard L., Faires, J. Douglas (2001) Numerical Analysis 7th Edition, Brooks/Cole, Pacific Grove, California, United States.

[3] Fletcher, Glena. (1994) Mathematical Methods in Physics, Wm. C. Brown Publishers, Dubuque, VA, United States.

[4] Gordon, Scott V., Clevenger, John (2019) Computer Graphics Programming in OpenGL with C++, Mercury Learning and Information, Dulles, VA, United States.

[5] Haberman, Richard. (1977) Mathematical Models: Mechanical Vibrations, Population Dynamics, and Traffic Flow, Prentice-Hall, Englewood Cliffs, New Jersey, United States.

[6] Walker, Jearl, Halliday, David, Resnick, Robert. (2014) Fundamental of Physics, John Wiley & Sons, Hoboken, New Jersey, United States.

[7] Hardy, Y, Tan Kiat Shi and Steeb, W.-H. (2008) Computer Algebra with SymbolicC++, World Scientific Publishing, Singapore.

[8] Hayt, William H. (2019) Engineering Circuit Analysis 9th Edition, McGraw-Hill Education, New York, United States.

[9] Judd, Kenneth L. (1999) Numerical Methods In Economics 2nd Edition, MIT Press, Cambridge, Massachusetts, United States.

[10] Phillips, Lee (2020) Gnuplot 5 Second Edition, Alogus Publishing.

[11] Anton, Rorres (2006) Elementary Linear Algebra with Supplemental Applications 10th Edition, Wiley, Boston, United States.

[12] Shekar, Siddharth (2019) C++ Game Development By Example, Packt, Birmingham, United Kingdom.

[13] Simmons, George F. (2017)Differential Equations with Applications and Historical Notes,CRC Press, Florida, United States.

[14] Strang, Gilbert (2007)Computational Science and Engineering, Wellesley-Cambridge Press, Wellesley, United States.

[15] Trefethen, Lloyd N., III, David Bau (1997) Numerical Linear Algebra, SIAM, 3600 University City Science Center, Philadelphia, United States.

[16] Vries, Joey de. (2020) Learn OpenGL - Graphics Programming, Kendall & Welling.

[17] Verzani, J., (2023) Calculus With Julia

# Listings


1 test.cpp "Hey Beautiful Goddess." . . . 22
2 main.cpp "Green Screen" . . . 23
3 main.cpp "SOIL GLM Rotating Images" . . . 26
4 main.cpp "SFML Keyboard Event" . . . 34
5 main.cpp "Render Mesh with Lighting" . . . 38
6 Camera.cpp "Render Mesh with Lighting" . . . 41
7 Camera.h "Render Mesh with Lighting" . . . 42
8 CMakeLists.txt "Render Mesh with Lighting" . . . 42
9 LightRenderer.cpp "Render Mesh with Lighting" . . . 43
10 LightRenderer.h "Render Mesh with Lighting" . . . 45
11 Mesh.cpp "Render Mesh with Lighting" . . . 46
12 Mesh.h "Render Mesh with Lighting" . . . 50
13 ShaderLoader.cpp "Render Mesh with Lighting" . . . 50
14 ShaderLoader.h "Render Mesh with Lighting" . . . 52
15 main.cpp "Cube and Moving Camera" . . . 55
16 camera.fs "Cube and Moving Camera" . . . 64
17 camera.vs "Cube and Moving Camera" . . . 64
18 main.cpp "Yaw Pitch Roll for 3D Cube" . . . 66
19 fragShader.glsl "Yaw Pitch Roll for 3D Cube" . . . 72
20 Utils.cpp "Yaw Pitch Roll for 3D Cube" . . . 73
21 Utils.h "Yaw Pitch Roll for 3D Cube" . . . 77
22 vertShader.glsl "Yaw Pitch Roll for 3D Cube" . . . 78
23 main.cpp . . . 98
24 vertexshader.vs . . . 99
25 main.cpp "Hello Box2D" . . . 106
26 main.cpp "Bullet Example" . . . 114
27 main.cpp "Slope plotting Gnuplot C++" . . . 119
28 gnuplot_i.hpp . . . 121
29 Makefile "Gnuplot Slope Makefile" . . . 122
30 Makefile "Gnuplot Slope Makefile" . . . 123
31 main.cpp "Function plotting Gnuplot C++" . . . 124
32 main.cpp "3D Surface plotting Gnuplot C++" . . . 126
33 main.cpp "3D Curve plotting Gnuplot C++" . . . 128
34 matrix.txt "2D Plot from Textfile with Gnuplot through C++" . . . 131
35 main.cpp "2D Plot from Textfile with Gnuplot through C++" . . . 131
36 Makefile "2D Plot from Textfile with Gnuplot through C++" . . . 133
37 main.cpp "2D Plot and Fit Curve from Textfile with Gnuplot through C++" . . . 134

38 Makefile "2D Plot and Fit Curve from Textfile with Gnuplot through C++" . . . . . 135
39 derivation.cpp "Symbolic Derivation for Function in One Variable C++" . . . . . 139
40 integral.cpp "Symbolic Integration for Function in One Variable C++" . . . . . 140
41 Makefile "SymbolicC++ Example" . . . . . 140
42 Compile Eigen . . . . . 144
43 Compile CLN . . . . . 145
44 Compile GiNaC . . . . . 145
45 Compile OpenBLAS . . . . . 147
46 Compile LAPACK . . . . . 147
47 Compile Armadillo . . . . . 148
48 Compile GSL . . . . . 148
49 Compile Tomsolver in GFreya OS in Qemu . . . . . 150
50 main.cpp . . . . . 152
51 Compile Assimp . . . . . 154
52 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 158
53 tests/projectile_dropped.cpp "Projectile Motion Box2D" . . . . . 166
54 tests/circular_motion.cpp "Uniform Circular Motion Box2D" . . . . . 180
55 tests/newton_firstlaw.cpp "Newton's First Law Box2D" . . . . . 190
56 tests/newton_firstlaw2dforces.cpp "Newton's First Law and 2D Forces Box2D" . . . 200
57 tests/pulley_jointtriangle.cpp "Applying Newton's Law Box2D" . . . . . 210
58 tests/frictional_forces.cpp "Static Frictional Forces Box2D" . . . . . 220
59 tests/pulley_kineticfriction.cpp "Pulley Kinetic Friction Box2D" . . . . . 227
60 tests/dragforce_skier.cpp "Drag Force Skier Box2D" . . . . . 235
61 tests/circular_motionairplane.cpp "Circular Motion Airplane Box2D" . . . . . 245
62 tests/kineticenergy_collision.cpp "Kinetic Energy Train Crash Box2D" . . . . . 252
63 tests/kineticenergy_work.cpp "Kinetic Energy Work Box2D" . . . . . 261
64 tests/work_gravitationalforce.cpp "Push and Pull along a Ramp Box2D" . . . . . 269
65 tests/spring_work.cpp "Work Done by A Spring Force Box2D" . . . . . 277
66 tests/work_generalvariable.cpp "Work Done by a General Variable Box2D" . . . . . 284
67 tests/work_pulleyelevator.cpp "Work Pulley Elevator Box2D" . . . . . 294
68 tests/work_cratependulum.cpp "Crate Pendulum Box2D" . . . . . 303
69 tests/work_verticalspringupward.cpp "Block Dropped onto Relaxed Spring Box2D" 309
70 tests/work_conveyorbelt.cpp "Work and Conveyor Belt Box2D" . . . . . 317
71 tests/potentialenergy_looptheloop.cpp "Potential Energy Frictionless Loop-the-Loop Box2D" . . . . . 333
72 tests/potentialenergy_verticalspringupward.cpp "Marble Fired Upward with Spring Box2D" . . . . . 345
73 tests/potentialenergy_runawaytruck.cpp "Runaway Truck with Failed Brakes Box2D" 353
74 tests/potentialenergy_springincline.cpp "Spring and Breadbox on an Incline Box2D" 361
75 tests/potentialenergy_marblespringgun.cpp "Marble Spring Gun Box2D" . . . . . 369
76 tests/conservationofenergy_blockslidestohigherlevel.cpp "Block Slides to Higher Level Box2D" . . . . . 396
77 tests/conservationofenergy_elevatorsnaps.cpp "Elevator Snaps Box2D" . . . . . 402
78 tests/conservationofenergy_particleslidestillstop.cpp "Particle Slides till Stop Box2D" 411
79 tests/centerofmass_threeparticles.cpp "Center of Mass Three Particles Box2D" . . . 421
80 tests/centerofmass_atomicbomb.cpp "Center of Mass Atomic Bomb Box2D" . . . . 432
81 tests/centerofmass_racecarwallcollision.cpp "Center of Mass Three Particles Box2D" 453
82 tests/centerofmass_airplaneexplosion.cpp "Center of Mass Airplane Explosion Box2D" 466

83 tests/centerofmass_bulletshotupward.cpp "Center of Mass Bullet Shot Upward Box2D" . . . 479
84 tests/centerofmass_elasticcollisionsteelball.cpp "Center of Mass Elastic Collision Steel Ball on Pendulum Box2D" . . . 492
85 tests/centerofmass_elasticcollisiontwoballs.cpp "Center of Mass Elastic Collision Baseball Basketball Box2D" . . . 498
86 tests/centerofmass_threeparticles.cpp "Center of Mass Three Particles Box2D" . . . 507
87 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 509
88 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 511
89 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 513
90 main.cpp "Gravity with 3 Bodies" . . . 517
91 tests/gravity_check.cpp "Gravity Check Box2D" . . . 526
92 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 533
93 tests/simple_pendulum.cpp "Simple Pendulum Box2D" . . . 543
94 tests/spring_test.cpp "Horizontal Spring Mass System Box2D" . . . 551
95 tests/verticalspring_test.cpp "Simple Vertical Spring Box2D" . . . 561
96 tests/spring_twomasses.cpp "Two Masses Spring-Mass System Box2D" . . . 577
97 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 587
98 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 589
99 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 591
100 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . 593
101 multiplicationofmatrices.cpp "Computation: Multiplication of matrices" . . . 606
102 main.cpp "Matrix times a Vector from Textfile Data" . . . 608
103 "vector1.txt" . . . 611
104 "matrix1.txt" . . . 611
105 main.cpp "Matrix Sum and Subtraction with Armadillo" . . . 614
106 main.cpp "Matrices Multiplication" . . . 616
107 main.cpp "Compute Determinant of a Square Matrix" . . . 623
108 main.cpp "Inverse of a Matrix With Adjoint" . . . 628
109 main.cpp "Symbolic Inverse of a Matrix" . . . 635
110 main.cpp "Solving Linear Systems With Cramer's Rule" . . . 637
111 main.cpp "Gauss-Jordan Elimination" . . . 644
112 main.cpp "2D Vector Addition" . . . 665
113 main.cpp "3D Plot Addition" . . . 668
114 main.cpp "Linear Combination" . . . 671
115 main.cpp "Norm Angle Dot Product and Distance" . . . 672
116 main.cpp "Line and Its Normal" . . . 675
117 main.cpp "Plane and Its Normal" . . . 679
118 main.cpp "Distance Between a Point and the Plane in 3D" . . . 683
119 main.cpp "Direction Cosines in 3D" . . . 688
120 main.cpp "Parametric Equation of Plane in 3D" . . . 692
121 main.cpp "Numerical Orthogonal Projection" . . . 695
122 main.cpp "Orthogonal Projection" . . . 700
123 main.cpp "Cross Product in 3D" . . . 701
124 main.cpp "Wronskian for Symbolic Matrix" . . . 782
125 main.cpp "3D Plot of Standard Basis Reflected toward a Line into New Bases" . . . 789
126 main.cpp "Particular and General Solution of a Linear System" . . . 795
127 main.cpp "General Solution of a Linear System Ax . . . 804

128 main.cpp "Basis for a Column Space by Row Reduction" . . . . . 810
129 main.cpp "Basis for the Row and Column Space by Row Reduction" . . . . . 818
130 main.cpp "Basis for a Row Space by Row Reduction and Matrix Transpose" . . . . . 829
131 main.cpp "Row Space and Column Space Application in Material Science" . . . . . 839
132 main.cpp "Rank Nullity and Basis for Null Space" . . . . . 845
133 main.cpp "Gauss-Jordan Elimination for Overdetermined Symbolic Matrix" . . . . . 856
134 main.cpp "Automatic Gauss-Jordan Elimination for Overdetermined Linear System" 862
135 main.cpp "Gauss-Jordan Elimination Rank and Basis for Null Space" . . . . . 869
136 main.cpp "2D Plot of Solution Space Ax . . . . . 885
137 main.cpp "Reflection in 2 dimensional space" . . . . . 897
138 main.cpp "Reflection in 3 dimensional space" . . . . . 904
139 main.cpp "Orthogonal Projection in 2 dimensional space" . . . . . 910
140 main.cpp "Orthogonal projection in 3 dimensional space" . . . . . 915
141 main.cpp "Rotation in 2 dimensional space" . . . . . 922
142 main.cpp "Rotations in 3 dimensional space about positive axis" . . . . . 928
143 main.cpp "Rotation about arbitrary unit vector in 3 dimensional space" . . . . . 936
144 main.cpp "Contraction and dilation on 2 dimensional space" . . . . . 941
145 main.cpp "Compression and expansion of 2 dimensional space" . . . . . 947
146 main.cpp "Shears in 2 dimensional space" . . . . . 953
147 main.cpp "Orthogonal Projection on lines through origin" . . . . . 963
148 main.cpp "Reflection about lines through the origin in 2 dimensional space" . . . . 968
149 main.cpp "Find the angle of rotation in 2 dimensional space" . . . . . 973
150 main.cpp "Angle of rotation and trace of matrix in 3 dimensional space" . . . . . 979
151 main.cpp "Determine axis of rotation and angle of rotation in 3 dimensional space" 983
152 main.cpp "Composition of two transformations in two dimensional space" . . . . . 989
153 main.cpp "Composition of three transformations in 3 dimensional space" . . . . . 993
154 main.cpp "Find inverse of standard matrix in symbolic terms" . . . . . 1001
155 main.cpp "Rotation transformation for 2 dimensional picture" . . . . . 1003
156 main.cpp "Translation transformation for 2 dimensional picture" . . . . . 1012
157 main.cpp "Compression and expansion transformation for 2 dimensional picture" . 1020
158 main.cpp "Reflection transformation for 2 dimensional picture" . . . . . 1027
159 main.cpp "Shear transformation for 2 dimensional picture" . . . . . 1037
160 main.cpp "Composition transformation for 2 dimensional picture" . . . . . 1045
161 main.cpp "Custom transformation with matrix multiplication for 2 dimensional picture" . . . . . 1054
162 main.cpp "Image of a Line" . . . . . 1064
163 main.cpp "The images of a line under various transformations" . . . . . 1070
164 main.cpp "Shear transformation for 3 dimensional cube with camera moving" . . . 1077
165 main.cpp "3D Object Load with assimp and opengl" . . . . . 1088
166 main.cpp "Apply transrotation to 3d object with opengl" . . . . . 1099
167 main.cpp "Dynamical system with power matrix" . . . . . 1113
168 main.cpp "Determine steady state vector for two states markov chain" . . . . . 1116
169 main.cpp "State vectors for three states markov chain" . . . . . 1120
170 main.cpp "Steady state vector for three states markov chain" . . . . . 1121
171 main.cpp "Compute characteristic polynomial" . . . . . 1150
172 main.cpp "Compute eigenvalues and eigenvectors" . . . . . 1151
173 main.cpp "Eigenvectors or bases for eigenspaces" . . . . . 1155
174 main.cpp "Solving Linear System with Equations Less than Unknowns" . . . . . 1159

175 main.cpp "Polynomial factorization" . . . . . 1162
176 main.cpp "Plot two Polynomial functions" . . . . . 1163
177 main.cpp "Bisection and Newton Raphson method" . . . . . 1164
178 main.cpp "Finding a matrix P that diagonalizes matrix A and the diagonal matrix" 1169
179 main.cpp "Powers of a diagonalizable matrix" . . . . . 1173
180 main.cpp "Complex conjugate real and imaginary parts of vectors and matrices" . . 1197
181 main.cpp "Complex euclidean inner product and norm" . . . . . 1201
182 main.cpp "Geometric interpretation of complex eigenvalues" . . . . . 1204
183 main.cpp "Matrix factorization using complex eigenvalues" . . . . . 1212
184 main.cpp "Power sequences of a matrix transformation on a vector" . . . . . 1218
185 main.cpp "Solution of a simple first order linear system of differential equations with initial conditions" . . . . . 1235
186 main.cpp "Solution of a general first order linear system of differential equations with initial conditions" . . . . . 1240
187 main.cpp "Solution of a second order linear differential equation" . . . . . 1249
188 main.cpp "Solution of a third order linear differential equation" . . . . . 1256
189 main.cpp "Evaluation inner product on polynomials" . . . . . 1283
190 main.cpp "Inner product or definite integral for polynomials" . . . . . 1286
191 main.cpp "Inner product or definite integral for trigonometry and transcendental functions" . . . . . 1289
192 main.cpp "Gauss-jordan elimination to obtain basis for an orthogonal complement" 1292
193 main.cpp "Orthonormal basis for null space of an underdetermined matrix" . . . . 1295
194 main.cpp "Plane in 3d and the orthogonal complement vectors" . . . . . 1304
195 main.cpp "Line in 3d and the orthogonal complement vectors" . . . . . 1310
196 main.cpp "Intersecting line of two planes" . . . . . 1314
197 main.cpp "Coordinate vector relative to an orthonormal basis" . . . . . 1322
198 main.cpp "Orthogonal projection of vector u on plane W" . . . . . 1325
199 main.cpp "Gram-Schmidt process in R2" . . . . . 1332
200 main.cpp "Gram-Schmidt process in R3" . . . . . 1340
201 main.cpp "Gram-Schmidt process in R3 with weighted inner product" . . . . . 1350
202 main.cpp "Legendre polynomials" . . . . . 1363
203 main.cpp "QR-decomposition" . . . . . 1368
204 main.cpp "Rotation of axes in 2 dimensional space" . . . . . 1383
205 main.cpp "Rotation of axes in 3 dimensional space around z-axis" . . . . . 1389
206 main.cpp "Rotation of axes in 3 dimensional space around z-axis then around y′-axis"1399
207 main.cpp "Orthogonally diagonalizing a symmetric matrix" . . . . . 1409
208 main.cpp "A geometric interpretation of a spectral decomposition" . . . . . 1416
209 main.cpp "Determine symmetric matrix from known eigenvalues and eigenvectors" 1424
210 main.cpp "Expressing quadratic forms in matrix notation" . . . . . 1451
211 main.cpp "Principal axes theorem" . . . . . 1453
212 src/ellipse.cpp "2D Ellipse Simulation" . . . . . 1459
213 src/hyperbola.cpp "2D hyperbola type 1 simulation" . . . . . 1464
214 src/hyperbola.cpp "2D hyperbola type 2 simulation" . . . . . 1470
215 main.cpp "Identifying a conic" . . . . . 1476
216 main.cpp "Ellipsoids in 3 dimensional space with gnuplot" . . . . . 1484
217 src/main.cpp "3D Simulation of ellipsoid and sphere" . . . . . 1486
218 main.cpp "Finding constrained extrema" . . . . . 1502
219 main.cpp "A constrained extremum problem" . . . . . 1507

220 src/ellipse.cpp "Finding constrained extrema with level curves" . . . . . 1513
221 main.cpp "Determine critical points" . . . . . 1525
222 main.cpp "Classify relative extrema with Hessian" . . . . . 1530
223 main.cpp "Compute eigenvalues and eigenvectors for 2 dimensional matrix" . . . . 1538
224 main.cpp "Compute Critical Points for Nonlinear System of Equations" . . . . . 1541
225 main.cpp "Classify Critical Point with second derivative test" . . . . . 1544
226 main.cpp "Eigenvalues and eigenvectors of a hermitian matrix" . . . . . 1568
227 main.cpp "Inverse of a unitary matrix" . . . . . 1570
228 main.cpp "Unitary diagonalization of a hermitian matrix" . . . . . 1574
229 main.cpp "Computing with Images of Basis Vectors" . . . . . 1634
230 main.cpp "Compute an inverse transformation from R3 to R3" . . . . . 1638
231 main.cpp "Finding T(x) indirectly" . . . . . 1642
232 main.cpp "Matrix for a linear transformation from R2 to R3" . . . . . 1646
233 main.cpp "Linear Operator on P2" . . . . . 1651
234 main.cpp "Composition of linear transformations" . . . . . 1655
235 main.cpp "Compute matrix T relative to the basis B′ for linear transformation" . . . 1657
236 main.cpp "Compute matrix T relative to the basis B′ for rotation" . . . . . 1661
237 main.cpp "Compute matrix T relative to the basis B′ for orthogonal projection on xy-plane" . . . . . 1664
238 main.cpp "Compute matrix T relative to the basis B′ for polynomial linear operator" 1668
239 main.cpp "Find a basis for R2 relative to matrix T that is diagonal" . . . . . 1673
240 main.cpp "Find Eigenvalues and Bases for Eigenspaces for Polynomial Linear Operator" . . . . . 1676
241 main.cpp "Polynomial interpolation with Vandermonde Matrix and Gauss-Jordan Elimination" . . . . . 1683
242 main.cpp "Gaussian Elimination" . . . . . 1694
243 main.cpp "Gaussian Elimination for Symbolic Matrix" . . . . . 1699
244 main.cpp "Least squares solution of an inconsistent linear system" . . . . . 1709
245 main.cpp "Orthogonal projection on a subspace" . . . . . 1716
246 main.cpp "Basis and standard matrix for an orthogonal projection on a plane" . . . 1720
247 main.cpp "Least Square Straight Line Fit" . . . . . 1731
248 main.cpp "Least Squares Straight Line Fit with Armadillo" . . . . . 1741
249 main.cpp "Least Squares Quadratic Polynomial Fit" . . . . . 1746
250 main.cpp "Plot for Least Squares Quadratic Polynomial Fit" . . . . . 1750
251 main.cpp "Least Squares Cubic Polynomial Fit" . . . . . 1762
252 main.cpp "Plot for Least Squares Cubic Polynomial Fit" . . . . . 1766
253 main.cpp "Least squares approximations with fourier series" . . . . . 1785
254 main.cpp "Plot of least squares approximations with fourier series" . . . . . 1786
255 main.cpp "Compute fourier series with known f(x)" . . . . . 1790
256 main.cpp "Plot 2D fourier series with known f(x)" . . . . . 1792
257 main.cpp "Plot fourier series with two different f(x) for square wave" . . . . . 1796
258 src/fourier.cpp "Fourier series simulation for sawtooth and square wave" . . . . . 1799
259 main.cpp "LU Decomposition" . . . . . 1810
260 main.cpp "The power method with Euclidean scaling" . . . . . 1827
261 main.cpp "The power method with maximum entry scaling" . . . . . 1834
262 main.cpp "A ranking procedure for search engine" . . . . . 1841
263 main.cpp "Singular value decomposition" . . . . . 1853
264 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1858

265 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1858
266 main.cpp "Bisection method" . . . . . 1878
267 main.cpp "Fixed point iteration" . . . . . 1882
268 main.cpp "Newton-Raphson method" . . . . . 1886
269 main.cpp "Secant method" . . . . . 1889
270 main.cpp "Compute Numerical Derivative with Autodiff" . . . . . 1896
271 main.cpp "2D Plot of Derivative of Univariate Function" . . . . . 1900
272 Makefile "2D Plot of Derivative of Univariate Function" . . . . . 1909
273 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1915
274 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1915
275 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1915
276 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1916
277 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1916
278 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1916
279 main.cpp "2D plot of complex function" . . . . . 1918
280 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1923
281 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1925
282 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1927
283 tests/projectile_motion.cpp "Projectile Motion Box2D" . . . . . 1929
284 pendulumdata . . . . . 1933
285 histogramplot . . . . . 1933
286 histogramplot . . . . . 1934
287 probabilitydensityfunction-normal . . . . . 1934
288 cumulativedistributionfunction-normal . . . . . 1938
289 vectorfieldplot . . . . . 1943
290 vectorfieldplot1 . . . . . 1943
291 vectorfieldplot2 . . . . . 1943
292 3dparametricplot-invertedcone . . . . . 1945
293 parametricplotsphere . . . . . 1945
294 contourplot . . . . . 1947
295 3dcomplexfunctionplot . . . . . 1948