Foundations of Linear Algebra

Overview

Quantum Mechanics is based upon theory of linear algebra. In order to understand its distinguishable traits with respect to classical mechanics, and not feeling overwhelmed by the results that it gives rise to, one must has the essential toolkit. In this notebook we're going to work on foundational concepts regarding linear algebra, and various methods that allows us to use computers for computations regarding this theory.

This notebook would first present complex numbers and geometric vectors, and the representation of the first in the latter. Thereafter, we start constructing linear algebra, for which we learn about inner product, function spaces, norms, and orthogonality. This would help us find the Gram-Schmidt Orthonormalization lemma. Later on, we would discuss vector spaces, linear maps, kernels, and images.

The next important steps then is to investigate matrices and linear systems. The next chapters are devoted into understanding factorization, eigenvalue problems, approximation and iterative methods, and stability. We would also have a look at Tensor spaces, wavelets and multiresolution analysis

Complex Numbers and Geometric Vectors

Complex Numbers and Functions

Complex numbers and analysis based on complex variable theory have become extremely important and valuable tools for the mathematical analysis of physical theory. Though the results of the measurement of physical quantities must, we firmly believe, ultimately be described by real numbers, there is ample evidence that successful theories predicting the results of those measurements require the use of complex numbers and analysis.

Basic Properties

A Complex number is nothing more than an ordered pair of two real numbers, (a, b). Similarly, a complex variable is an ordered pair of two real variables $z \equiv (x, y)$. The ordering is significant. We continue writing real numbers $(x, 0)$ as simply x and define the imaginary unit $i \equiv (0, 1)$.

Addition

Addition is defined in complex numbers in terms of their Cartesian components as:

$$z_1 + z_2 = (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2).$$

Multiplication

Multiplication of complex numbers is defined as:

$$z_1 z_2 = (x_1, y_1)(x_2, y_2) = \left(x_1 x_2 - y_1 y_2, x_1 y_2 + x_2 y_1\right).$$

It is obvious that multiplication is not just the multiplication of corresponding components. Using this definition we can verify that: (do it as an example by implementing the definition)

```
In[]:= ⅈ^2
Out[]= -1
```

Formal Properties

The space of complex numbers, sometimes denoted by Z has the following formal properties:

1. It is closed under addition and multiplication.

2. It has a unique zero number, which when added to any complex number leaves it unchanged and which, when multiplied with any complex number yields zero.

3. It has a unique unit number, 1, which when multiplied with any complex number leaves it unchanged.

4. Every complex number has an inverse under addition, and an inverse under multiplication.

5. It is closed under exponentiation, meaning that if u, v are complex numbers u^v is also a complex number.

Complex Conjugation

Like all complex numbers, i has an inverse under addition, denoted $-i$. Given a complex number

```
In[]:= z = x + ⅈ y
Out[]= x + ⅈ y
```

It is useful to define another complex number

```
In[]:= zStar = x - ⅈ y
Out[]= x - ⅈ y
```

which we call the complex conjugate of z forming:

```
In[]:= z zStar // Expand
Out[]= x^2 + y^2
```

we see that $z z^*$ is real; we define the absolute value of z, denoted $|z|$ as $\sqrt{z z^*}$.

Functions in the Complex Domain

Since the fundamental operations in the complex numbers domain obey the same rules as those for arithmetic in the space of real numbers, it is natural to define functions so that their real and complex incarnations are similar. Applying this concept to the exponential, we define:

```
In[∘]:= (*Step 1:Create Taylor series for e^x*)
series = Series[Exp[x], {x, 0, 10}] // Normal

(*Step 2:Substitute x=I θ*)
seriesComplex = series /. x → I θ

(*Step 3:Separate into real and imaginary parts*)
realPart = ComplexExpand[Re[seriesComplex]]
imagPart = ComplexExpand[Im[seriesComplex]]

(*Step 4:Simplify*)
realPartSimp = Simplify[realPart]
imagPartSimp = Simplify[imagPart]

(*Step 5:Show that as a→∞,these approach cos(θ) and sin(θ)*)
(*For this,we can compute the exact expressions*)
exactCosine = Series[Cos[θ], {θ, 0, 10}] // Normal
exactSine = Series[Sin[θ], {θ, 0, 10}] // Normal

(*Compare with our real and imaginary parts*)
Simplify[realPartSimp - exactCosine]
Simplify[imagPartSimp - exactSine]
```

Therefore, proving that

$$e^{i\theta} = \cos\theta + i\sin\theta$$

The Argand Plane and Geometric Interpretation

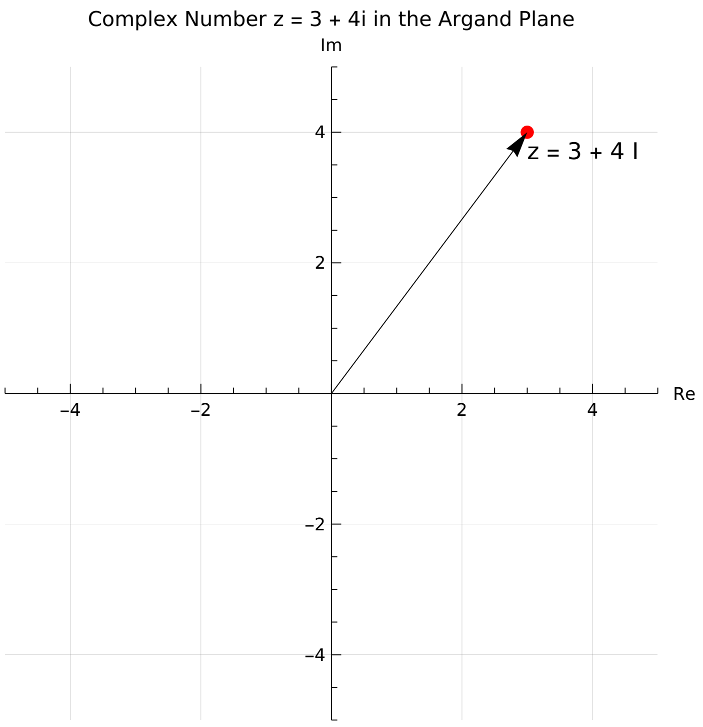
The Argand plane (also known as the complex plane) provides a geometric representation of complex numbers . In this representation, a complex number $z = a + b\,i$ is plotted as a point (a, b) in a two - dimensional coordinate system, where:

- The horizontal axis represents the real part of the complex number
- The vertical axis represents the imaginary part of the complex number

Basic Representation

In[∘]:=
```
(*Define a complex number*)
z = 3 + 4 * I;

(*Visualize a single complex number in the Argand plane*)
argandPoint =
 ListPlot[{{Re[z], Im[z]}}, PlotStyle → {Red, PointSize[0.02]}, PlotRange → {{-5, 5}, {-5, 5}},
  AspectRatio → 1, AxesLabel → {"Re", "Im"}, GridLines → Automatic, Epilog →
   {Arrow[{{0, 0}, {Re[z], Im[z]}}], Text[Style["z = " <> ToString[z], 12], {Re[z], Im[z]}, {-1, 1}]},
  PlotLabel → "Complex Number z = 3 + 4i in the Argand Plane"]
```

Out[∘]=



Polar Representation

A complex number can also be represented in polar form as $z = r\, e^{i\,\theta}$, where:

- r is the magnitude (or modulus) of the complex number
- θ is the argument (or phase) of the complex number

In Mathematica, we can convert between rectangular and polar forms:

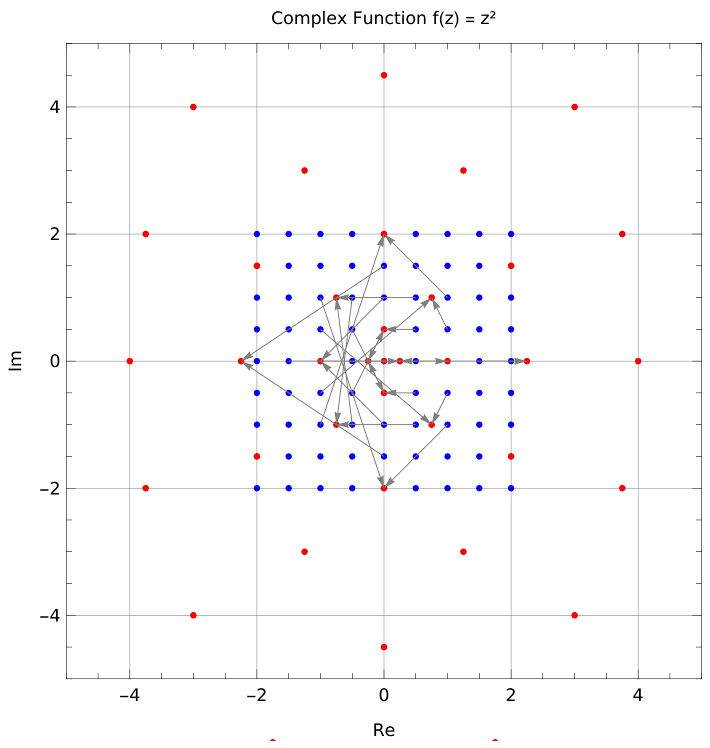
```
In[]:= (*Convert from rectangular to polar form*)
z = 3 + 4 * I;
r = Abs[z];
theta = Arg[z];

(*Display in exponential form*)
polarForm = r * Exp[I * theta];
polarFormString = ToString[r] <> " e^(" <> ToString[theta, InputForm] <> "i)";

(*Verify using Euler's formula*)
eulersFormula = r * (Cos[theta] + I * Sin[theta])
Simplify[eulersFormula - z] (*Should be 0*)

Out[]= 3 + 4 ⅈ

Out[]= 0
```

Visualizing Complex Functions

We can visualize how complex functions transform the Argand plane:

```
In[•]:= (*Create a grid of complex numbers*)
gridPoints = Table[a + b * I, {a, -2, 2, 0.5}, {b, -2, 2, 0.5}];
gridPoints = Flatten[gridPoints];

(*Define a complex function*)
f[z_] := z ^ 2;

(*Apply the function to each point*)
transformedPoints = f /@ gridPoints;

(*Visualize the transformation*)
transformation = Graphics[{
    (*Original grid points*)
    Blue, PointSize[0.01], Point[{Re[#], Im[#]} & /@ gridPoints],
    (*Transformed grid points*)
    Red, PointSize[0.01], Point[{Re[#], Im[#]} & /@ transformedPoints],
    (*Connect corresponding points with arrows*)
    Gray, Arrowheads[0.02],
    Arrow[{{Re[#], Im[#]}, {Re[f[#]], Im[f[#]]}}] & /@ Select[gridPoints, Abs[#] ≤ 1.5 &]
   },
   PlotRange → {{-5, 5}, {-5, 5}}, AspectRatio → 1, Frame → True,
   FrameLabel → {{"Im", None}, {"Re", "Complex Function f(z) = z²"}}, GridLines → Automatic]
```

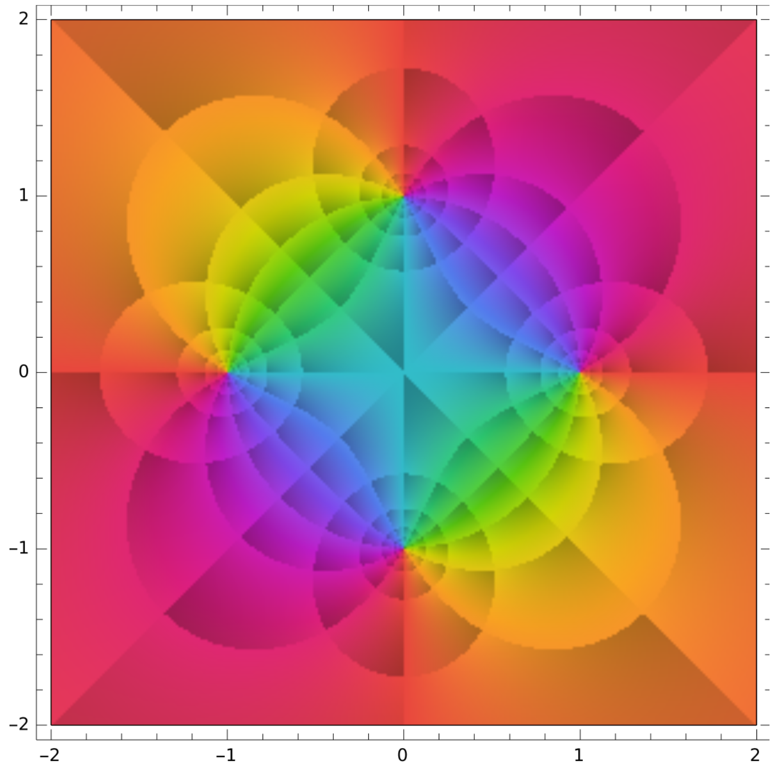
Out[•]=



Or more maturely we can use the built-in function in Mathematica:

```
In[∘]:= ComplexPlot[(z^2 + 1) / (z^2 - 1), {z, -2 - 2 I, 2 + 2 I}, ColorFunction → "CyclicLogAbsArg"]
```

Out[∘]=



Complex Exponential and Trigonometric Functions

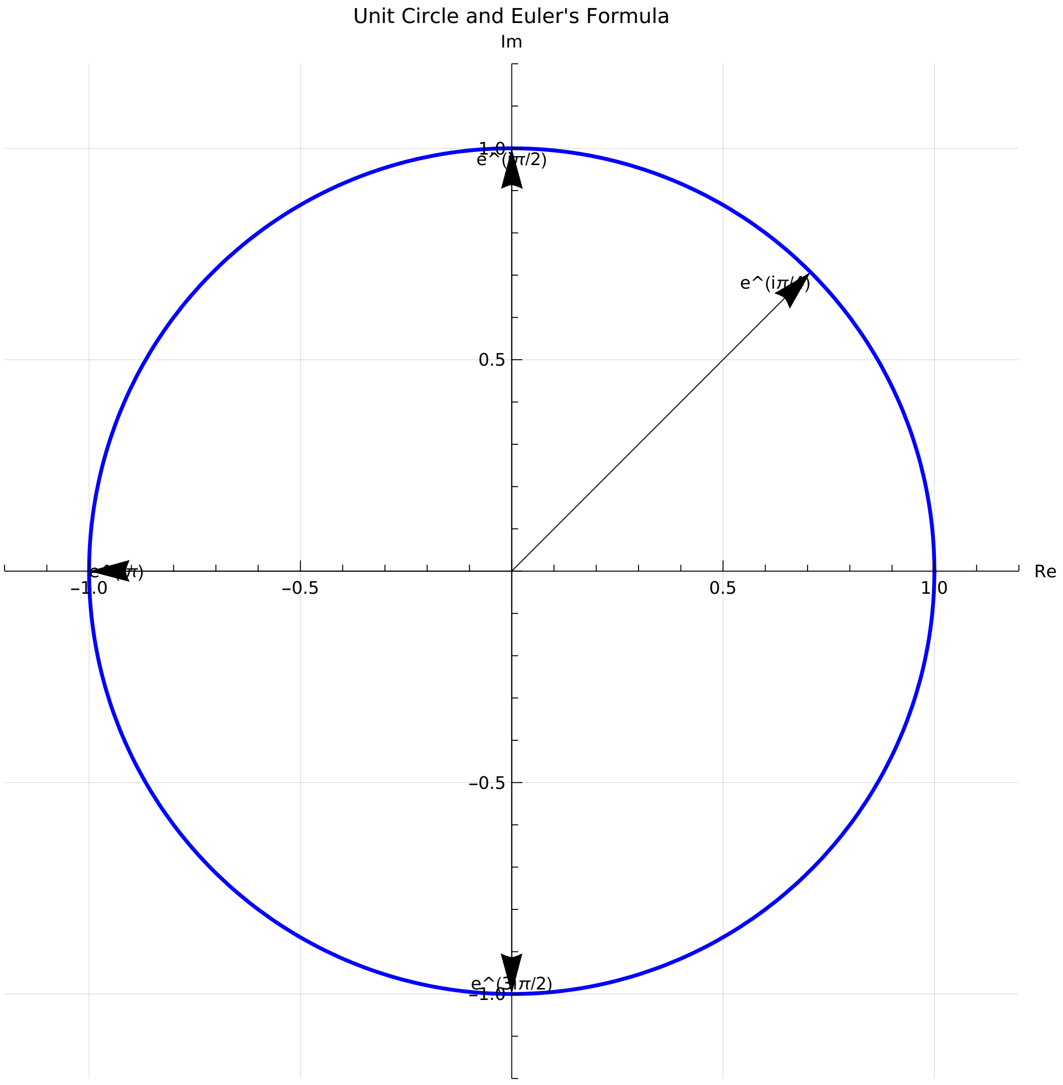
Using Euler' s identity, we can express trigonometric functions in terms of complex exponentials :

```
In[∘]:= (*Euler's identity*)
eulerIdentity = Exp[I * θ] == Cos[θ] + I * Sin[θ];

(*Solving for cosine and sine*)
cosTheta = Simplify[(Exp[I * θ] + Exp[-I * θ]) / 2]
sinTheta = Simplify[(Exp[I * θ] - Exp[-I * θ]) / (2 * I)]

(*Visualizing the relationship with a unit circle*)
unitCircle = ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2 * Pi},
  PlotStyle → {Thick, Blue}, PlotRange → {{-1.2, 1.2}, {-1.2, 1.2}},
  AspectRatio → 1, AxesLabel → {"Re", "Im"}, GridLines → Automatic,
  Epilog → {Arrow[{{0, 0}, {Cos[Pi / 4], Sin[Pi / 4]}}], Text["e^(iπ/4)", {Cos[Pi / 4], Sin[Pi / 4]}, {1, 1}],
    Arrow[{{0, 0}, {Cos[Pi / 2], Sin[Pi / 2]}}], Text["e^(iπ/2)", {Cos[Pi / 2], Sin[Pi / 2]}, {0, 1}],
    Arrow[{{0, 0}, {Cos[Pi], Sin[Pi]}}], Text["e^(iπ)", {Cos[Pi], Sin[Pi]}, {-1, 0}], Arrow[
     {{0, 0}, {Cos[3 * Pi / 2], Sin[3 * Pi / 2]}}], Text["e^(3iπ/2)", {Cos[3 * Pi / 2], Sin[3 * Pi / 2]}, {0, -1}]},
  PlotLabel → "Unit Circle and Euler's Formula"]
```

Out[ ]= $$\frac{1}{2}\left(e^{-i\theta}+e^{i\theta}\right)$$

Out[ ]= $$-\frac{1}{2}\, i\, e^{-i\theta}\left(-1+e^{2 i\theta}\right)$$

Out[ ]=



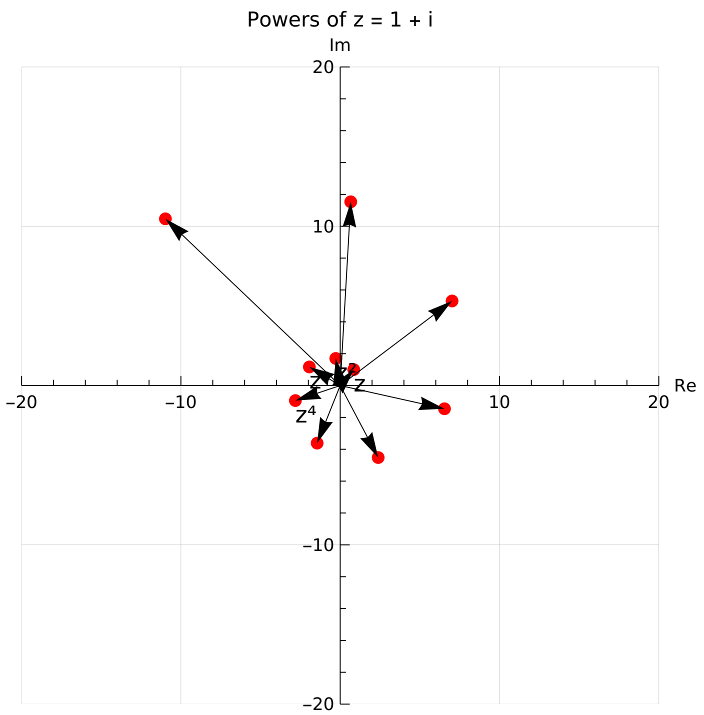
De Moivre’s Formula

De Moivre’s formula provides a simple way to compute powers of complex numbers:

In[∘]:=
```
(*De Moivre's formula:(r e^(iθ))^n=r^n e^(inθ)*)
deMoivreFormula = Simplify[(r * Exp[I * θ]) ^ n == r ^ n * Exp[I * n * θ]]

(*Example:Computing powers of complex numbers*)
z = 0.85 + I; (*r=√2,θ=π/4*)
powers = Table[z ^ k, {k, 1, 10}];

(*Show the powers on the Argand plane*)
powerPlot = ListPlot[{Re[#], Im[#]} & /@ powers,
   PlotStyle → {Red, PointSize[0.02]}, PlotRange → {{-20, 20}, {-20, 20}},
   AspectRatio → 1, AxesLabel → {"Re", "Im"}, GridLines → Automatic, Epilog →
    {Arrow[{{0, 0}, {Re[#], Im[#]}}] & /@ powers, Text[Style["z", 12], {Re[z], Im[z]}, {-1, 1}], Text[
      Style["z²", 12], {Re[z ^ 2], Im[z ^ 2]}, {-1, 1}], Text[Style["z³", 12], {Re[z ^ 3], Im[z ^ 3]}, {-1, 1}],
     Text[Style["z⁴", 12], {Re[z ^ 4], Im[z ^ 4]}, {-1, 1}]}, PlotLabel → "Powers of z = 1 + i"]
```

Out[∘]= $5^n \left(e^{i\,\theta}\right)^n == 5^n\, e^{i\,n\,\theta}$

Out[∘]=



Complex Roots

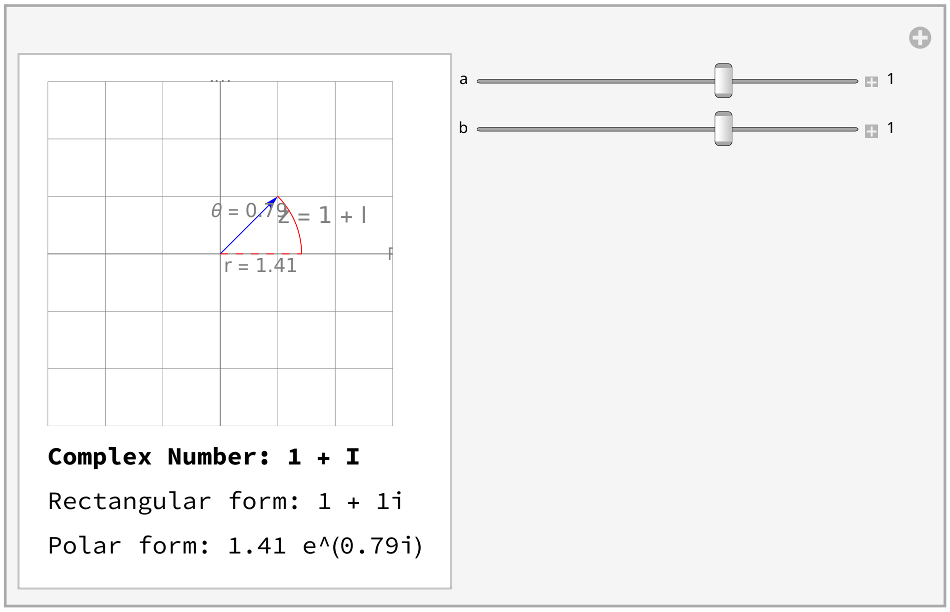
We can find the n - th roots of a complex number and visualize them :

```
In[ ]:= (*Find nth roots of a complex number*)
findRoots[z_, n_] := Table[Abs[z]^(1/n) * Exp[I * (Arg[z] + 2 * Pi * k) / n], {k, 0, n - 1}]

(*Example:Find the 5th roots of 1*)
roots = findRoots[1, 5];

(*Visualize the roots*)
rootsPlot = ListPlot[{Re[#], Im[#]} & /@ roots,
  PlotStyle → {Red, PointSize[0.02]}, PlotRange → {{-1.2, 1.2}, {-1.2, 1.2}},
  AspectRatio → 1, AxesLabel → {"Re", "Im"}, GridLines → Automatic,
  Epilog → {Circle[], (*Unit circle*)Arrow[{{0, 0}, {Re[#], Im[#]}}] & /@ roots, MapIndexed[
     Text[Style["ω" <> ToString[First[#2] - 1], 12], {Re[#1], Im[#1]}, {Sign[Re[#1]], Sign[Im[#1]]}] &,
     roots]}, PlotLabel → "The 5th Roots of Unity"]
```

Interactive Exploration of Complex Numbers

```
In[•]:= Manipulate[
  Column[
   {
    Graphics[
     {
      (*Argand plane*)
      Gray, Line[{{-3, 0}, {3, 0}}], Line[{{0, -3}, {0, 3}}], Text["Re", {3.1, 0}], Text["Im", {0, 3.1}],
      (*Vector representation*)
      {
       Blue, Arrow[{{0, 0}, {a, b}}]}, Text[Style["z = " <> ToString[a + b * I], 12], {a, b}, {-1, 1}],
      (*Polar form information*)
      {Red, Circle[{0, 0}, Sqrt[a ^ 2 + b ^ 2], {0, ArcTan[a, b]}]},
      {Red, Dashed, Line[{{0, 0}, {Sqrt[a ^ 2 + b ^ 2], 0}}]},
      Text[Style["r = " <> ToString[Round[Sqrt[a ^ 2 + b ^ 2], 0.01]], 10], {Sqrt[a ^ 2 + b ^ 2] / 2, -0.2}],
      Text[Style["θ = " <> ToString[Round[ArcTan[a, b], 0.01]], 10], {a / 2, b / 2}, {0, -1}]
     },
     PlotRange → {{-3, 3}, {-3, 3}}, AspectRatio → 1, GridLines → Automatic
    ],
    Style["Complex Number: " <> ToString[a + b * I], Bold],
    "Rectangular form: " <> ToString[a] <> " + " <> ToString[b] <> "i", "Polar form: " <>
     ToString[Round[Sqrt[a ^ 2 + b ^ 2], 0.01]] <> " e^(" <> ToString[Round[ArcTan[a, b], 0.01]] <> "i)"}
  ], {{a, 1}, -3, 3, 0.1, Appearance → "Labeled"},
  {{b, 1}, -3, 3, 0.1, Appearance → "Labeled"}, ControlPlacement → Right
 ]
```

Out[∘]=



```
Complex Number: 1 + I
Rectangular form: 1 + 1i
Polar form: 1.41 e^(0.79i)
```

In[∘]:=
```
TextCell["This gives us the L²–norm of the sine function on
   [0, π]. Now, the distance between two vectors/functions is defined as:

d(f, g) = ||f – g||

Try computing distance between Sin[x] and Cos[x]:", "Text"]
```

Out[∘]= This gives us the L^2–norm of the sine function on $[0, \pi]$. Now, the distance between two vectors/functions is defined as:

d(f, g) = ||f – g||

Try computing distance between Sin[x] and Cos[x]:

Vector Spaces

Lists

Suppose n is a nonnegative integer. A list of length n is an ordered collection of n elements. Two lists are equal if and only if they have the same length and the same elements in the same order (notice the difference between lists and sets).

Lists are often written as elements separated by commas and surrounded by parentheses. In Mathematica however, we use curly brackets to show lists:

```
In[∘]:= A = {1, 2, 3, 4};
```

Coordinate

F^n is the set of all lists of length n of elements of F.

```
In[∘]:= F[A_, n_] := Tuples[A, n];
        F[A, 2]

Out[∘]= {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 2}, {2, 3},
          {2, 4}, {3, 1}, {3, 2}, {3, 3}, {3, 4}, {4, 1}, {4, 2}, {4, 3}, {4, 4}}
```

If F be the real numbers for $n = 2, 3$ we get the usual R^2 and R^3. We can also define F to be the set of complex numbers, thus creating a complex coordinate.

Addition

Addition in F^n is defined by adding corresponding coordinates:

```
In[∘]:= X = F[A, 2];
        X[[1]] + X[[2]] == {X[[1]][[1]] + X[[2]][[1]], X[[1]][[2]] + X[[2]][[2]]}
Out[∘]= True
```

It is obvious to show that this definition of sum is commutative. Do it here as an exercise. Also one can define the additive inverse of any element of F^n by negative sign as:

```
In[∘]:= Fn = Flatten[{Tuples[A, 2] , Tuples[-A, 2]}, 1]
Out[∘]= {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 2}, {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 3}, {3, 4}, {4, 1},
          {4, 2}, {4, 3}, {4, 4}, {-1, -1}, {-1, -2}, {-1, -3}, {-1, -4}, {-2, -1}, {-2, -2}, {-2, -3},
          {-2, -4}, {-3, -1}, {-3, -2}, {-3, -3}, {-3, -4}, {-4, -1}, {-4, -2}, {-4, -3}, {-4, -4}}
```

Scalar Multiplication

Having dealt with addition in F^n, we now turn to multiplication. We could define a multiplication in F^n in a similar fashion.

The product of a number λ and a vector in F^n is computed by multiplying each coordinate of the vector by λ:

```
In[•]:= λ Fn[[1]]
Out[•]= {λ, λ}
```

Scalar multiplication has a nice geometric interpretation in R^2. If $\lambda > 0$ and $x \in R^2$, then $\lambda\, x$ is a vector that points in the same direction as x.

Digression on Fields

A field is a set containing at least two distinct elements called 0 and 1 along with operations of addition and multiplication satisfying all properties below:

1. Commutativity
2. Associativity
3. Identities
4. Additive inverse
5. Multiplication inverse
6. distributive property

Thus R, C are fields, as is the set of rational numbers along with the usual operations of addition and multiplication.

Vector Space

The motivation for the definition of a vector space comes from properties of addition and scalar multiplication in F^n: Addition is commutative, associative and has an identity. Every element has an additive inverse. Scalar multiplication is associative. Scalar multiplication by 1 acts as expected. Addition and scalar multiplication are connected by distributive properties.

We will define a vector space to be a set V with an addition and a scalar multiplication on V that satisfy the properties above.

Addition and Scalar Multiplication

An addition on set V is a function that assigns an element $u + v \in V$ to each pair of elements $u, v \in V$.

A scalar multiplication on a set V is a function that assigns an element $\lambda\, v \in V$ to each $\lambda \in F$ and each $v \in V$.

Definition of Vector Space

A vector space is a set V along with an addition on V and a scalar multiplication on V such that the following properties hold.

Commutativity: $\forall\, u, v \in V : v + u = u + v$

Associativity: $\forall\, a, b \in F \wedge \forall\, v, u, w \in V : (u + v) + w = u + (v + w) \wedge (a\, b)\, v = a\,(b\, v)$

Additive Identity: $\exists\, 0 \in V$ such that $\forall\, v \in V : v + 0 = 0 + v = v$.

Additive Inverse:$\forall\, v \in V\, \exists\, w \in V : w + v = 0$.

Multiplicative Identity: $\forall\, v \in V\, \exists\, 1 \in F : 1\, v = v$.

Distributive properties: $\forall\, a, b \in F \wedge \forall\, u, v \in V : a\,(u + v) = a\, u + a\, v\, (a + b)\, v = a\, v + b\, v$.

Note: Elements of a vector space are called *vectors or points.*

The scalar multiplication in a vector space depends on F. But usually the choice of F is either clear from the context or irrelevant. Thus we often assume that F is lurking in tha background without specifically mentioning it. With the usual operations of addition and scalar multiplication F^n is a vector space over F, as you should verify.

Subspaces

A subset U of V is called a subspace of V if U is also a vector space with the same additive identity, addition, and scalar multiplication as on V.

Sum of Subspaces

Suppose $V_1, \ldots, V_m$ are subspaces of V. The sum of $V_1, \ldots, V_m$, denoted by $V_1 + \ldots + V_m$, is the set of all possible sums of elements of $V_1, \ldots, V_m$. More precisely,

$$V_1 + \ldots + V_m = \{v_1 + \ldots + v_m : v_1 \in V_1, \ldots, v_m \in V_m\}.$$

Direct Sum

Suppose V_is are subspaces of V.

- The sum of subspaces $\sum_i V_i$ is called a direct sum if each element of the sum can be written in only one way as a sum $\sum_i v_i$, where $v_i \in V_i$. For direct sums we use the notation

$\bigoplus$.

Basis Vectors

Definition

A list of vectors in a vector space that is small enough to be linearly independent and big enough so the linear combinations of the list fill up the vector space is called a basis of the vector space. We will see that every basis of a vector space has the same length, which will allow us to define the dimension of a vector space

A basis of V, is a list of vectors in V that is linearly independent and spans V.

Linear Combination

A linear combination of a list $v_1, \ldots, v_m$ of vectors in V, is a vector of the form $\sum_i a_i v_i$ where $a_i \in F$.

Span

The set of all linear combination of a list of vectors v_i in V is called the span of v_i, denoted by span$(v_1, \ldots, v_n)$. If span of a list of vectors equals V, we say that the list spans V. A vector space is called finite-dimensional if some list of vectors in it spans the space.

Finite-Dimensional Vector Space

A vector space is called finite-dimensional if a finite set of vectors can span it. For example:

```
In[∘]:= Fn
Out[∘]= {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 2}, {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 3}, {3, 4}, {4, 1},
         {4, 2}, {4, 3}, {4, 4}, {-1, -1}, {-1, -2}, {-1, -3}, {-1, -4}, {-2, -1}, {-2, -2}, {-2, -3},
         {-2, -4}, {-3, -1}, {-3, -2}, {-3, -3}, {-3, -4}, {-4, -1}, {-4, -2}, {-4, -3}, {-4, -4}}
```

Can be spanned by:

```
In[∘]:= V1 = {1, 0}
        V2 = {0, 1}

Out[∘]= {1, 0}

Out[∘]= {0, 1}
```

Because for each member we can find:

```
In[∘]:= Flatten[Table[Solve[a V1 + b V2 == Fn[[i]]], {i, 1, Length[Fn]}], 1]

Out[∘]= {{a → 1, b → 1}, {a → 1, b → 2}, {a → 1, b → 3}, {a → 1, b → 4}, {a → 2, b → 1}, {a → 2, b → 2},
          {a → 2, b → 3}, {a → 2, b → 4}, {a → 3, b → 1}, {a → 3, b → 2}, {a → 3, b → 3}, {a → 3, b → 4},
          {a → 4, b → 1}, {a → 4, b → 2}, {a → 4, b → 3}, {a → 4, b → 4}, {a → -1, b → -1},
          {a → -1, b → -2}, {a → -1, b → -3}, {a → -1, b → -4}, {a → -2, b → -1}, {a → -2, b → -2},
          {a → -2, b → -3}, {a → -2, b → -4}, {a → -3, b → -1}, {a → -3, b → -2}, {a → -3, b → -3},
          {a → -3, b → -4}, {a → -4, b → -1}, {a → -4, b → -2}, {a → -4, b → -3}, {a → -4, b → -4}}
```

An infinite-dimensional vector space can also be defined when one proves that the vector space is not finite-dimensional.

Linear Independent

A set of vectors is called linearly independent when the equation below satisfies only if the coefficients of the vector are zero, namely:

```
In[∘]:= Solve[a V1 + b V2 == 0]

Out[∘]= {{a → 0, b → 0}}
```

So the vectors defined above are linearly independent.

Criterion for Basis

A list $v_1, \ldots, v_n$ of vectors in V is a basis of V if and only if every $v \in V$ can be written uniquely in the form: $v = \sum_i a_i v_i$ where $a_i \in F$.

Every spanning list in a vector space can be reduced to a basis of the vector space. And every finite-dimensional vector space has at least one basis. Every linearly independent list extends to a basis (either it is a basis or by adding vectors to it we can make a basis from it). Any two bases of a finite-dimensional vector space have the same length. The dimension of a finite-dimensional vector space is the length of any basis of the vector space. We use the notation dim V.

Linearly independent list of the right length is a basis.

Suppose V is a finite-dimensional vector space. Then every linearly independent list of vectors in V of length dim V is a basis of V.

Spanning list of the right length is a basis

Suppose V is finite-dimensional. Then every spanning list of vectors in V of length dim V is a basis of V.

Linear Maps

Introduction

So far our attention has focused on vector spaces. No one gets excited about vector spaces. The interesting part of linear algebra is the subject to which we now turn linear maps.

We will frequently use the powerful fundamental theorem of linear maps, which states that the dimension of the domain of a linear map equals the dimension of the subspace that gets sent to 0 plus the dimension of the range. This will imply the striking result that a linear map from a finite-dimensional vector space to itself is one-to-one if and only if its range is the whole space.

A major concept that we will introduce in this chapter is the matrix associated with a linear map and with a basis of the domain space and a basis of the target space. This correspondence between linear maps and matrices provides much insight into key aspects of linear algebra.

Definition

A Linear Map from V to W is a function $T : V \to W$ with the following properties:
1. **Additivity**: $T\,(u + v) = T\,u + T\,v \,\forall\, v,\, u \in V$
2. **Homogeneity**: $T\left(\lambda\, v\right) = \lambda\left(T\,v\right) \forall\, \lambda \in F \wedge \forall\, v \in V$

Note that for linear maps we often use the notation $T\,v$ as well as the usual function notation $T(v)$.

The set of linear maps from V to W is denoted by $L(V, W)$. And the set of linear maps from a vector space V to itself is simply $L(V)$.

Addition and Scalar Multiplication on $L(V, W)$

Suppose $S, T \in L(V, W)$ and $\lambda \in F$. The sum $S + T$ and the product λT are the linear maps from V to W defined by:

$(S + T)(v) = S v + T v$ and $\left(\lambda T\right)(v) = \lambda\left(T v\right)$

Because we took the trouble to define addition and scalar multiplication on $L(V, W)$, the next result should not be a surprise.

With the operations of addition and scalar multiplication as defined above, $L(V, W)$ is a vector space.

The Product of Linear Maps

If $T \in L(U, V)$ and $S \in L(V, W)$, then the product $S T \in L(U, W)$ is defined by:

$S T(u) = S\left(T u\right)$

note that the domain and range of functions are connected with each other.

Fundamental Theorem of Linear Maps

Suppose V is a finite dimensional and $T \in L(V, W)$. Then the range is finite dimensional and:

$\dim V = \dim \operatorname{null} T + \dim \operatorname{range} T.$

Matrices

Representing a Linear Map by a Matrix

We know that if $v_1, \ldots, v_n$ is a basis of V and $T : V \to W$ is linear, then the values of $T v_1, \ldots, T v_n$ determine the values of T on arbitrary vectors in V. Matrices provide an efficient method of recording the values of $T\,v_k$'s in terms of a basis of W.

Suppose m and n are nonnegative integers. An m-by-n matrix A is a rectangular array of elements of F with m rows and n columns:

```
In[∘]:= A = (a11 a12 a13
             a21 a22 a23
             a31 a32 a33)

Out[∘]= {{a11, a12, a13}, {a21, a22, a23}, {a31, a32, a33}}

In[∘]:= % // MatrixForm
Out[∘]//MatrixForm=
        (a11  a12  a13
         a21  a22  a23
         a31  a32  a33)
```

Matrix of a Linear Map

Suppose $T \in L(V, W)$ and v_i are a basis of V and w_i are a basis of W. The matrix of T with respect to these bases are defined by:

$$T\,v_k = \sum_{i=1}^{m} A_{i,k}\,w_i$$

Addition and Scalar Multiplication of Matrices

The sum of two matrices of the same size is the matrix obtained by adding corresponding entries in the matrices:

```
In[∘]:= A = {{1, 1}, {2, 3}};
        B = {{2, 1}, {3, 4}};
        A // MatrixForm
        B // MatrixForm
        A . B // MatrixForm
Out[∘]//MatrixForm=
        (1  1
         2  3)
Out[∘]//MatrixForm=
        (2  1
         3  4)
Out[∘]//MatrixForm=
        ( 5   5
         13  14)
```

For scalar multiplication, one can use the same means of vector-scalar multiplication,

```
In[•]:= λ A

Out[•]= {{λ, λ}, {2 λ, 3 λ}}
```

I won’t go in anymore details about matrices but I gave some references at the end of this notebook.

Matrix Factorization

Matrix factorization (or decomposition) refers to the process of expressing a matrix as a product of two or more matrices. These factorizations are extremely useful in numerical linear algebra, as they simplify complex operations, increase computational efficiency, and provide insights into the mathematical structure of the original matrix.

We’ll explore five fundamental matrix factorization techniques:

- LU Decomposition
- QR Factorization
- LDL Decomposition
- Singular Value Decomposition (SVD)
- Low Rank Approximations

Each technique has specific applications and advantages depending on the problem at hand.

LU Decomposition

$L\,U$ decomposition factors a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U:
$A = L\,U$
This decomposition is particularly useful for solving systems of linear equations, computing determinants, and inverting matrices.

For a square matrix A, the $L\,U$ decomposition works by using Gaussian elimination to transform A into an upper triangular matrix U, while keeping track of the elimination operations in a lower triangular matrix L.
The process requires that A can be factorized without row exchanges (pivoting). If pivot-

ing is needed, we get a slightly modified form: $P A = L U$, where P is a permutation matrix.

```
In[•]:= (*Create a sample matrix*)
        A = {{1, 3, 2}, {2, 2, 3}, {3, 4, 4}};
        MatrixForm[A]

        (*Perform LU decomposition*)
        {lu, p, c} = LUDecomposition[A];

        L = LowerTriangularize[lu, -1] + IdentityMatrix[3];
        U = UpperTriangularize[lu];
        L // MatrixForm
        U // MatrixForm
        L.U // MatrixForm
        Equal[A, L.U]
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 2 & 3 \\ 3 & 4 & 4 \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{5}{4} & 1 \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 3 & 2 \\ 0 & -4 & -1 \\ 0 & 0 & -\frac{3}{4} \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 2 & 3 \\ 3 & 4 & 4 \end{pmatrix}$$

```
Out[•]= True
```

Example: Solving Linear System with LU Decomposition

```
In[∘]:= (*Define a linear system Ax=b*)
        A = {{4, 3, 2}, {2, 1, 3}, {3, 4, 1}};
        b = {13, 12, 19};

        (*Perform LU decomposition*)
        {lu, p, c} = LUDecomposition[A];
        L =  LowerTriangularize[lu, -1] + IdentityMatrix[3];
        U = UpperTriangularize[lu];
        (*Solve the system using LU decomposition*)

        x = Inverse[U].Inverse[L]. b

        Print["Solution to the system Ax = b:"];
        x

        (*Verify the solution*)
        Print["Verification: Ax = b?"];
        A.x
        b
        Sort[A.x] == Sort[b]
         (* We have to sort to make sure we're treating it like a set not like a list.*)
```

Out[∘]= $\left\{-\frac{66}{13}, \frac{94}{13}, \frac{69}{13}\right\}$

```
Solution to the system Ax = b:
```

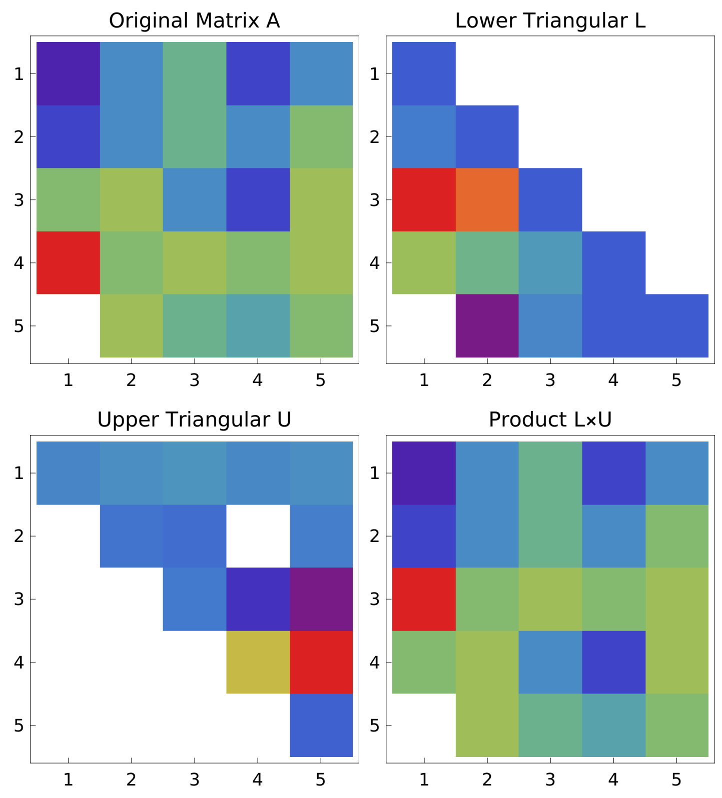
Out[∘]= $\left\{-\frac{66}{13}, \frac{94}{13}, \frac{69}{13}\right\}$

```
Verification: Ax = b?
Out[∘]= {12, 13, 19}
Out[∘]= {13, 12, 19}
Out[∘]= True
```

Visualization of LU Decomposition

```
In[∘]:= (*Create a function to visualize matrices*)visualizeMatrix[mat_, title_] :=
         ArrayPlot[mat, ColorRules → {0 → White}, ColorFunction → "Rainbow", Frame → True,
          FrameTicks → {Range[1, Length[mat]], Range[1, Length[mat[[1]]]]}, PlotLabel → title]


        A = {
           {1, 4, 6, 2, 4},
           {2, 4, 6, 4, 7},
           {7, 8, 4, 2, 8},
           {14, 7, 8, 7, 8},
           {0, 8, 6, 5, 7}
          };
        (*Perform LU decomposition*)
        {lu, p, c} = LUDecomposition[A];

        (*Extract L and U*)
        L = LowerTriangularize[lu, -1] + IdentityMatrix[5];
        U = UpperTriangularize[lu];
        (*Visualize A,L,U,and L×U*)
        Grid[{{visualizeMatrix[A, "Original Matrix A"], visualizeMatrix[L, "Lower Triangular L"]},
          {visualizeMatrix[U, "Upper Triangular U"], visualizeMatrix[L.U, "Product L×U"]}}]
```

Out[ ]=



QR Factorization

$Q\,R$ factorization decomposes a matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R:

$A = Q^T\,R.$

This decomposition is particularly useful for solving least squares problems, computing eigenvalues, and orthogonalizing sets of vectors.

There are several methods for computing $Q\,R$ decomposition including:

1. Gram-Schmidt orthogonalization
2. Householder reflections
3. Givens rotations

The $Q\,R$ decomposition has the useful property that $Q^T\,Q = I$ (where Q^T is the transpose of Q), which simplifies many matrix operations.

In[∘]:=
```
(*Create a sample matrix*)
A = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
MatrixForm[A]

(*Perform QR decomposition*)
{Q, R} = QRDecomposition[A];

(*Display Q and R*)
Print["Orthogonal matrix Q:"];
MatrixForm[Q]

Print["Upper triangular matrix R:"];
MatrixForm[R]

(*Verify decomposition*)
Print["Verification: Q×R equals original matrix A?"];
MatrixForm[A] == MatrixForm[ConjugateTranspose[Q].R]
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
Orthogonal matrix Q:
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} \frac{1}{\sqrt{66}} & 2\sqrt{\frac{2}{33}} & \frac{7}{\sqrt{66}} \\ \frac{3}{\sqrt{11}} & \frac{1}{\sqrt{11}} & -\frac{1}{\sqrt{11}} \end{pmatrix}$$

```
Upper triangular matrix R:
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} \sqrt{66} & 13\sqrt{\frac{6}{11}} & 15\sqrt{\frac{6}{11}} \\ 0 & \frac{3}{\sqrt{11}} & \frac{6}{\sqrt{11}} \end{pmatrix}$$

```
Verification: Q×R equals original matrix A?
```

Out[∘]= True

Example: Solving Least Squares Problem

```
In[•]:= (*Create an overdetermined system*)
A = {{1, 1}, {2, 1}, {3, 1}, {4, 1}};
b = {6, 5, 7, 10};

(*QR decomposition approach to least squares*)
{Q, R} = QRDecomposition[A];
y = Q.b;
x = LinearSolve[R, y[[1 ;; Length[R]]]]

Print["Solution to the least squares problem:"];
x

(*Compare with direct least squares solution*)
lsqSolution = LinearSolve[Transpose[A].A, Transpose[A].b]
Print["Direct least squares solution:"];
lsqSolution

(*Calculate and display residual*)
residual = Norm[A.x - b]
Print["Residual norm: ", residual]

(*Visualization of the fit for a line y=mx+c*)
model[t_, {m_, c_}] := m * t + c;
fitPlot = Show[ListPlot[Transpose[{A[[All, 1]], b}],
    PlotStyle → {Red, PointSize[0.02]}, PlotLabel → "Least Squares Fit"],
  Plot[model[t, x], {t, 0, 5}, PlotStyle → Blue], Frame → True, PlotRange → All,
  FrameLabel → {"x", "y"}, PlotLegends → {"Data Points", "Fitted Line"}]
```

Out[•]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

```
Solution to the least squares problem:
```

Out[•]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Out[•]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

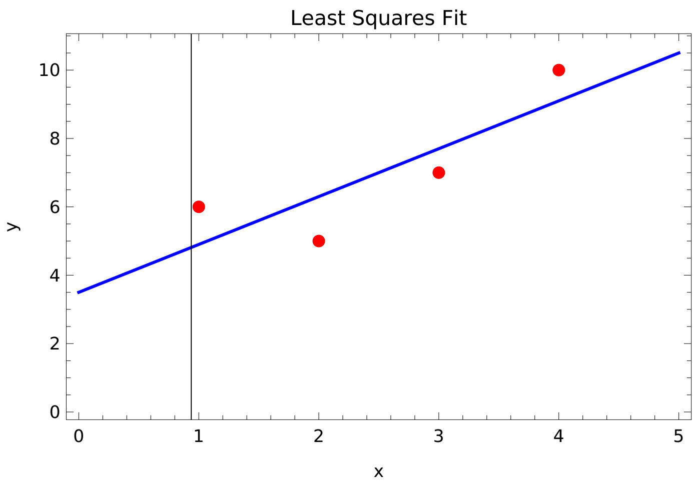
```
Direct least squares solution:
```

Out[•]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Out[∘]= $\sqrt{\frac{21}{5}}$

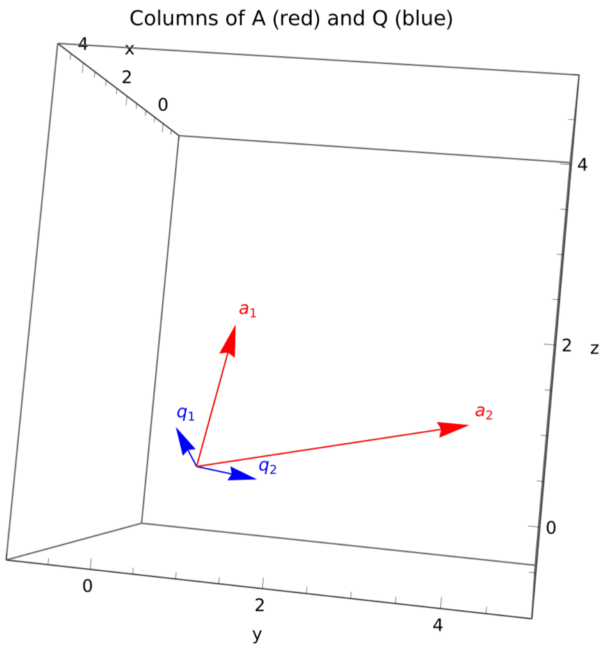
```
Residual norm:
```
$\sqrt{\frac{21}{5}}$

Out[∘]=



Visualization: QR Decomposition

In[∘]:=
```
(*Create a 3×2 matrix for visualization*)
A = {{3, 2, 4}, {1, 4, 5}, {2, 1, 2}};

(*Perform QR decomposition*)
{Q, R} = QRDecomposition[A];

(*Create 3D vectors representing the columns of A and Q*)
aVectors = Transpose[A];
qVectors = Transpose[Q];

(*Visualization in 3D space*)
vectorPlot =
  Graphics3D[{{Red, Arrow[{{0, 0, 0}, aVectors[[1]]}], Text["a1", aVectors[[1]] + {0.2, 0.2, 0.2}]},
    {Red, Arrow[{{0, 0, 0}, aVectors[[2]]}], Text["a2", aVectors[[2]] + {0.2, 0.2, 0.2}]},
    {Blue, Arrow[{{0, 0, 0}, qVectors[[1]]}], Text["q1", qVectors[[1]] + {0.2, 0.2, 0.2}]},
    {Blue, Arrow[{{0, 0, 0}, qVectors[[2]]}], Text["q2", qVectors[[2]] + {0.2, 0.2, 0.2}]}},
   PlotRange → {{-1, 5}, {-1, 5}, {-1, 5}}, BoxRatios → {1, 1, 1}, Axes → True,
   AxesLabel → {"x", "y", "z"}, PlotLabel → "Columns of A (red) and Q (blue)"];

Show[vectorPlot]
```

Out[∘]=


LDL Decomposition and Positive Definite

Matrices

$L\,D\,L$ decomposition is a variant of $L\,U$ decomposition for symmetric matrices
where:
$A = L\,D\,L^T$
Here, L is a lower triangular matrix with 1’s on the diagonal, and D is a diagonal matrix.

For a symmetric matrix A, the $L\,D\,L$ decomposition provides a more efficient and numerically stable alternative to Cholesky decomposition, especially when A is not guaranteed to be positive definite.

If A is positive definite, we can further simplify to the Cholesky decomposition, where A = $L\,L^T$.

```
In[•]:= (*Create a symmetric matrix*)
        A = {{4, 2, 2}, {2, 4, 4}, {2, 4, 4}};
        MatrixForm[A]

        (*Verify that A is symmetric*)
        Print["Is A symmetric? ", A == Transpose[A]]

        (*Perform LDL decomposition manually*)
        n = Length[A];
        L = IdentityMatrix[n];
        Diag = ConstantArray[0, n]

        For[j = 1, j ≤ n, j++, (*Calculate diagonal element of D*)
         Diag[[j]] = A[[j, j]] - Sum[L[[j, k]]^2 * Diag[[k]], {k, 1, j - 1}];
         (*Calculate elements of L below the diagonal*)
         For[i = j + 1, i ≤ n, i++, L[[i, j]] = (A[[i, j]] - Sum[L[[i, k]] * L[[j, k]] * Diag[[k]], {k, 1, j - 1}]) / Diag[[j]];]]

        (*Create diagonal matrix from D*)
        Dmat = DiagonalMatrix[Diag];

        (*Display L and D*)
        Print["Lower triangular matrix L:"];
        MatrixForm[L]

        Print["Diagonal matrix D:"];
        MatrixForm[Dmat]

        (*Verify decomposition*)
        Print["Verification: L×D×L^T equals original matrix A?"];
        MatrixForm[L.Dmat.Transpose[L]] == MatrixForm[A]
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 4 & 2 & 2 \\ 2 & 4 & 4 \\ 2 & 4 & 4 \end{pmatrix}$$

```
Is A symmetric? True
```

Out[•]= {0, 0, 0}

```
Lower triangular matrix L:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & 1 & 1 \end{pmatrix}$$

```
Diagonal matrix D:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

```
Verification: L×D×L^T equals original matrix A?
```

Out[•]=
```
True
```

Positive Definite Matrices and Cholesky Decomposition

In[•]:=
```
(*Check if A is positive definite*)
eigenvalues = Eigenvalues[A];
Print["Eigenvalues of A: ", eigenvalues];
Print["Is A positive definite? ", And @@ (eigenvalues > 0)]

(*If A is positive definite,we can use Cholesky decomposition*)
If[And @@ (eigenvalues > 0),
 (*Perform Cholesky decomposition*)L = CholeskyDecomposition[A];
 Print["Cholesky factor L:"];
 MatrixForm[L];
 (*Verify Cholesky decomposition*)
 Print["Verification: L×L^T equals original matrix A?"];
 MatrixForm[L.Transpose[L]];
 MatrixForm[A];
 Equal[L.Transpose[L], A],
 Print["Matrix is not positive definite, cannot perform Cholesky decomposition"]]
```

Eigenvalues of A: $\left\{2\left(3+\sqrt{3}\right),\ 2\left(3-\sqrt{3}\right),\ 0\right\}$

Is A positive definite? $\left\{2\left(3+\sqrt{3}\right),\ 2\left(3-\sqrt{3}\right),\ 0\right\}$ && 0

Out[•]=
```
If[{2 (3 + √3), 2 (3 - √3), 0} && 0, L = CholeskyDecomposition[A];
 Print[Cholesky factor L:];
 L;
 Print[Verification: L×L^T equals original matrix A?];
 L.Transpose[L];
 A;
 L.Transpose[L] == A,
 Print[Matrix is not positive definite, cannot perform Cholesky decomposition]]
```

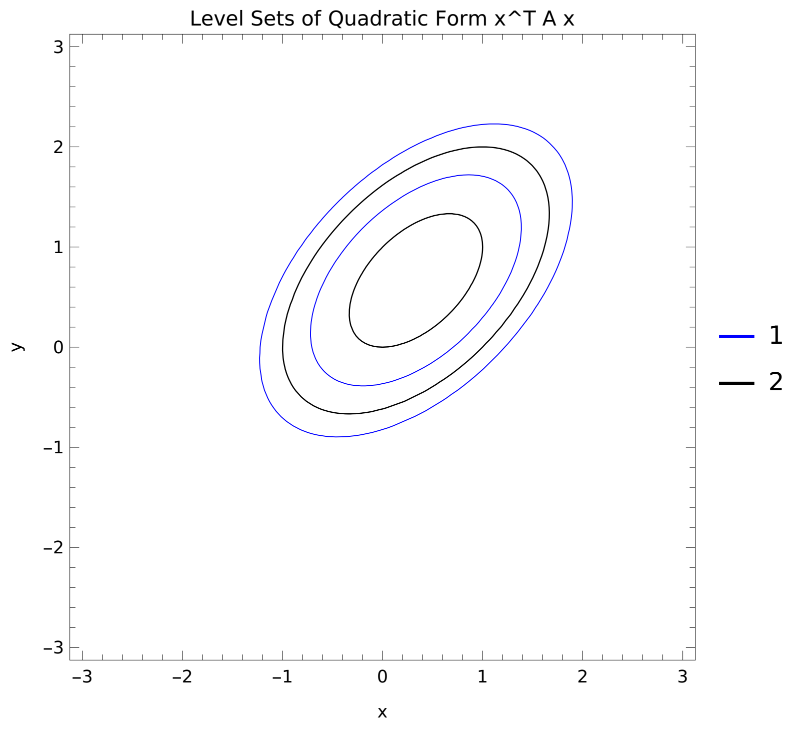
Visualization: Positive Definite Matrices and Ellipsoids

```
In[∘]:= (*Create a positive definite matrix*)A = {{2, -1, 0}, {-1, 2, -1}, {0, -1, 2}};

        (*Check positive definiteness*)
        eigenvalues = Eigenvalues[A];
        Print["Eigenvalues: ", eigenvalues];
        Print["Is A positive definite? ", And @@ (eigenvalues > 0)]

        (*Create a quadratic form function*)
        quadraticForm[x_, mat_] := x.mat.x

        (*Visualize a 2D ellipse corresponding to the quadratic form*)
        ellipsePlot = ContourPlot[quadraticForm[{x, y, 1}, A], {x, -3, 3}, {y, -3, 3},
          Contours → {1, 2, 3, 4, 5}, ContourShading → None, ContourStyle → {Blue, Thickness[0.002]},
          PlotLabel → "Level Sets of Quadratic Form x^T A x",
          FrameLabel → {"x", "y"}, PlotTheme → "Detailed"]

        (*Visualize the eigenvalues and eigenvectors*)
        {vals, vecs} = Eigensystem[A[[1 ;; 2, 1 ;; 2]]];
        eigenvectorPlot =
          Graphics[{Red, Arrow[{{0, 0}, vecs[[1]] * Sqrt[vals[[1]]]}], Arrow[{{0, 0}, vecs[[2]] * Sqrt[vals[[2]]]}],
            Text["λ₁ = " <> ToString[vals[[1]]], vecs[[1]] * Sqrt[vals[[1]]] * 1.1],
            Text["λ₂ = " <> ToString[vals[[2]]], vecs[[2]] * Sqrt[vals[[2]]] * 1.1]}];

        Show[ellipsePlot, eigenvectorPlot]
```

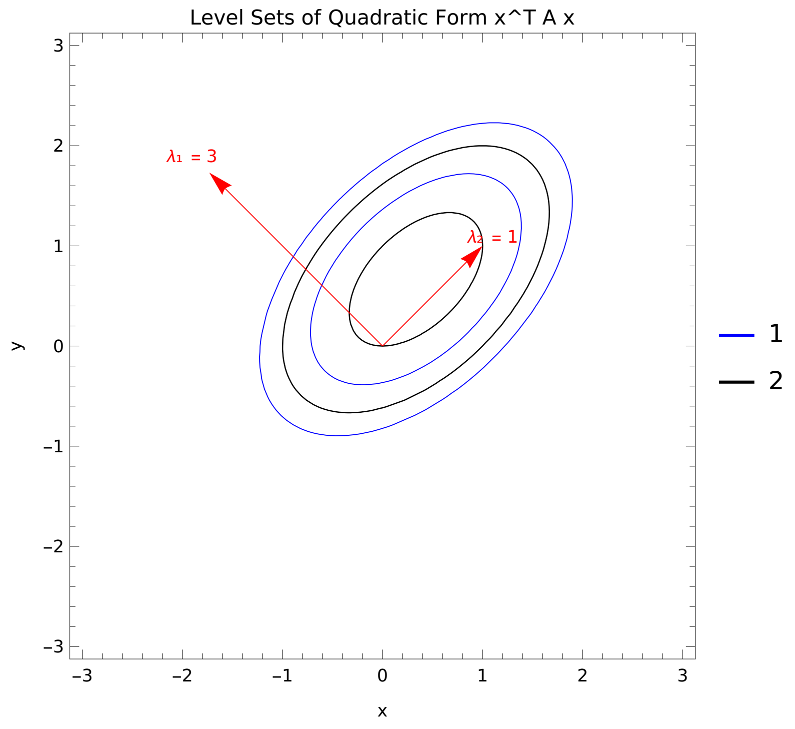
Eigenvalues: $\{2+\sqrt{2}, 2, 2-\sqrt{2}\}$

Is A positive definite? $\{2+\sqrt{2}, 2, 2-\sqrt{2}\}$ && 0

Out[ ]=



Out[ ]=



Singular Value Decomposition

Singular Value Decomposition (SVD) is one of the most important matrix factorizations, expressing any m×n matrix A as:

$A = U \Sigma V^T$

where:

- U is an $m \times m$ orthogonal matrix containing the left singular vectors
- Σ is an $m \times m$ diagonal matrix containing singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$.
- V^T is the transpose of an $n \times n$ orthogonal matrix V containing the right singular vectors.

SVD reveals fundamental properties of a linear transformation represented by matrix A:

- The singular values indicate the “stretching” in each direction.
- The left singular vectors (columns of U) form an orthonormal basis for the column space.
- The right singular vectors (columns of V) form an orthonormal basis for the row space.

```
In[∘]:= (*Create a sample matrix*)
A = {{3, 2, 2}, {2, 3, -2}};
MatrixForm[A]

(*Perform SVD*)
{U, S, V} = SingularValueDecomposition[A];

(*Display the components*)
Print["Left singular vectors (U):"];
MatrixForm[U]

Print["Singular values (Σ):"];
MatrixForm[S]

Print["Right singular vectors (V):"];
MatrixForm[V]

(*Verify decomposition*)
Print["Verification: U×Σ×V^T equals original matrix A?"];
MatrixForm[U.S.Transpose[V]]
MatrixForm[A]
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$$

```
Left singular vectors (U):
```

Out[•]//MatrixForm=

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

```
Singular values (Σ):
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}$$

```
Right singular vectors (V):
```

Out[•]//MatrixForm=

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{3\sqrt{2}} & -\frac{2}{3} \\ \frac{1}{\sqrt{2}} & \frac{1}{3\sqrt{2}} & \frac{2}{3} \\ 0 & -\frac{2\sqrt{2}}{3} & \frac{1}{3} \end{pmatrix}$$

```
Verification: U×Σ×V^T equals original matrix A?
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$$

Example: Computing Pseudoinverse

In[∘]:=
```
(*Create a non-square matrix*)
A = {{1, 2}, {3, 4}, {5, 6}};
MatrixForm[A]

(*Compute pseudoinverse using SVD*)
{U, S, V} = SingularValueDecomposition[A];

(*Replace non-zero singular values with their reciprocals*)
SInv = S;
For[i = 1, i ≤ Min[Dimensions[S]], i++, If[S[[i, i]] > 10^-10, SInv[[i, i]] = 1/S[[i, i]], SInv[[i, i]] = 0]]

(*Calculate pseudoinverse:A⁺=V·Σ⁺·U^T*)
Aplus = V.Transpose[SInv].Transpose[U];
Print["Pseudoinverse of A:"];
MatrixForm[Aplus]

(*Verify the pseudoinverse properties*)
Print["Verification: A·A⁺·A = A?"];
MatrixForm[A.Aplus.A]
MatrixForm[A]
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

```
Pseudoinverse of A:
```

Out[∘]//MatrixForm=

$$\left(\begin{array}{l} \frac{(-21+\sqrt{8185})\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)}{44\sqrt{2(91+\sqrt{8185})\left(1+\frac{(-21+\sqrt{8185})^2}{7744}\right)\left(\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)^2+\left(4+\frac{3}{88}(-21+\sqrt{8185})\right)^2+\left(6+\frac{5}{88}(-21+\sqrt{8185})\right)^2\right)}}+\frac{1}{4}\left(-\frac{14}{11}+\frac{12}{11(91+\sqrt{8185})}\right)\left(\frac{8}{11}+\right. \\ \left(2+\frac{1}{88}\left(-21+\sqrt{8185}\right)\right)\sqrt{\frac{2}{(91+\sqrt{8185})\left(1+\frac{(-21+\sqrt{8185})^2}{7744}\right)\left(\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)^2+\left(4+\frac{3}{88}(-21+\sqrt{8185})\right)^2+\left(6+\frac{5}{88}(-21+\sqrt{8185})\right)^2\right)}}+\frac{1}{4}\left(\frac{8}{11}+\right. \end{array}\right.$$

```
Verification: A·A⁺·A = A?
```

Out[∘]//MatrixForm=

$$\left(\begin{array}{l}
\frac{(-21+\sqrt{8185})\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)}{44\sqrt{2(91+\sqrt{8185})\left(1+\frac{(-21+\sqrt{8185})^2}{7744}\right)\left(\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)^2+\left(4+\frac{3}{88}(-21+\sqrt{8185})\right)^2+\left(6+\frac{5}{88}(-21+\sqrt{8185})\right)^2\right)}}+\frac{1}{4}\left(-\frac{14}{11}+\frac{12}{11(91+\sqrt{8185})}\right)\Big(\\
4\left(2+\frac{1}{88}\left(-21+\sqrt{8185}\right)\right)\sqrt{\frac{2}{(91+\sqrt{8185})\left(1+\frac{(-21+\sqrt{8185})^2}{7744}\right)\left(\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)^2+\left(4+\frac{3}{88}(-21+\sqrt{8185})\right)^2+\left(6+\frac{5}{88}(-21+\sqrt{8185})\right)^2\right)}}+\frac{1}{4}\Big(\\
6\left(2+\frac{1}{88}\left(-21+\sqrt{8185}\right)\right)\sqrt{\frac{2}{(91+\sqrt{8185})\left(1+\frac{(-21+\sqrt{8185})^2}{7744}\right)\left(\left(2+\frac{1}{88}(-21+\sqrt{8185})\right)^2+\left(4+\frac{3}{88}(-21+\sqrt{8185})\right)^2+\left(6+\frac{5}{88}(-21+\sqrt{8185})\right)^2\right)}}+\frac{1}{4}\Big(
\end{array}\right.$$

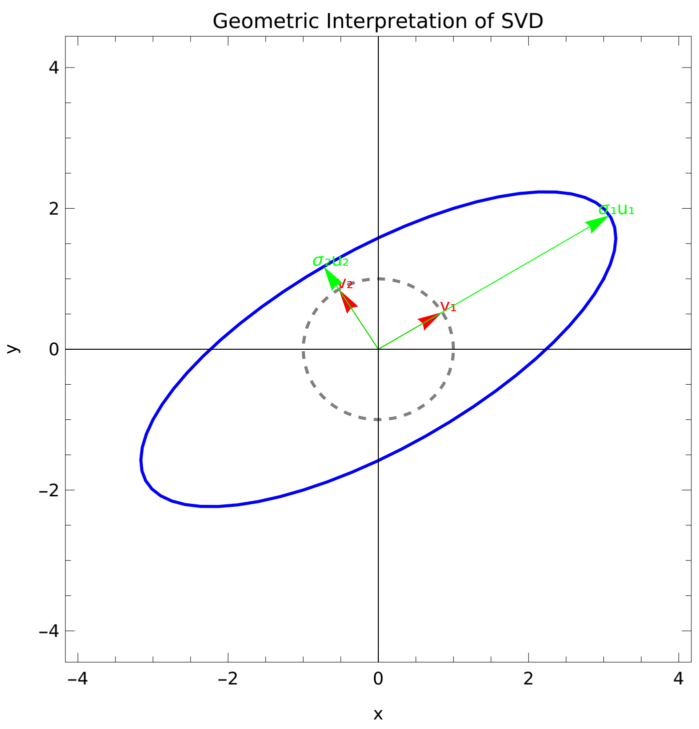
Out[∘]//MatrixForm=

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

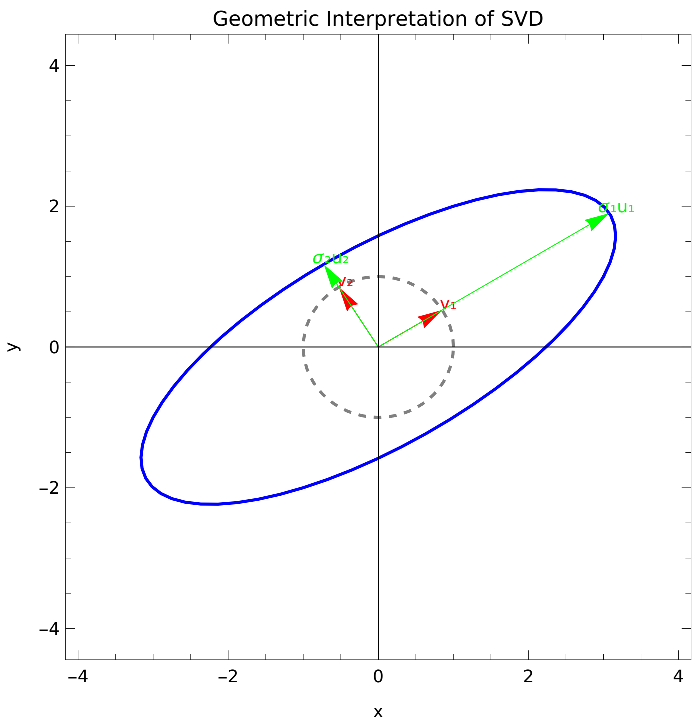
Visualization: Geometric Interpretation of SVD

```
In[∘]:= (*Create a 2×2 matrix for visualization*)
A = {{3, 1}, {1, 2}};

(*Perform SVD*)
{U, S, V} = SingularValueDecomposition[A];
s1 = S⟦1, 1⟧;
s2 = S⟦2, 2⟧;

(*Define unit circle and its image under A*)
unitCircle = Table[{Cos[t], Sin[t]}, {t, 0, 2 Pi, Pi / 30}];
transformedCircle = Map[A.# &, unitCircle];

(*Define vectors for visualization*)
v1 = V⟦All, 1⟧;
v2 = V⟦All, 2⟧;
u1 = U⟦All, 1⟧;
u2 = U⟦All, 2⟧;

(*Create the visualization*)
svdPlot =
 Show[(*Unit circle*)ListLinePlot[unitCircle, AspectRatio → 1, PlotStyle → {Gray, Dashed}],
  (*Image of unit circle under A*)ListLinePlot[transformedCircle, PlotStyle → {Blue}],
  (*Right singular vectors*)Graphics[{{Red, Arrow[{{0, 0}, v1}], Text["v₁", v1 + {0.1, 0.1}]},
    {Red, Arrow[{{0, 0}, v2}], Text["v₂", v2 + {0.1, 0.1}]}}],
  (*Left singular vectors scaled by singular values*)
  Graphics[{{Green, Arrow[{{0, 0}, s1 * u1}], Text["σ₁u₁", s1 * u1 + {0.1, 0.1}]},
    {Green, Arrow[{{0, 0}, s2 * u2}], Text["σ₂u₂", s2 * u2 + {0.1, 0.1}]}}], PlotRange → {{-4, 4}, {-4, 4}},
  PlotLabel → "Geometric Interpretation of SVD", Frame → True, FrameLabel → {"x", "y"}]

svdPlot
```

Out[∘]=



Out[∘]=



Low Rand Approximations

Low rank approximations use truncated SVD to represent a matrix with fewer parameters while minimizing the approximation error.

The Eckart-Young theorem states that the best rank-k approximation to a matrix A in terms of the Frobenius norm is given by:

$A_k = U_k \Sigma_k V_k^T$.

Where:

- U_k contains the first k columns of U.
- Σ_k is the $k \times k$ top-left submatrix of Σ.
- V_k contains the first k columns of V.

The approximation error is
$|| A - A_k || F = \sqrt{\sigma_{k+1}^2 + \ldots + \sigma_r^2}$

```
In[∘]:= (*Create a sample matrix*)
A = {{3, 1, 1, 1}, {1, 3, 1, 1}, {1, 1, 3, 1}, {1, 1, 1, 3}};
MatrixForm[A]

(*Perform SVD*)
{U, S, V} = SingularValueDecomposition[A];

(*Display singular values*)
singularValues = Table[S[[i, i]], {i, 1, Min[Dimensions[S]]}];
Print["Singular values: ", singularValues];

(*Function to compute rank-k approximation*)
rankKApproximation[U_, S_, V_, k_] := U[[All, 1 ;; k]].S[[1 ;; k, 1 ;; k]].Transpose[V[[All, 1 ;; k]]];

(*Compute different rank approximations*)
A1 = rankKApproximation[U, S, V, 1];
A2 = rankKApproximation[U, S, V, 2];
A3 = rankKApproximation[U, S, V, 3];

(*Display the approximations*)
Print["Rank-1 approximation:"];
MatrixForm[A1]

Print["Rank-2 approximation:"];
MatrixForm[A2]

Print["Rank-3 approximation:"];
```

```
MatrixForm[A3]

(*Calculate approximation errors*)
error1 = Norm[A - A1, "Frobenius"];
error2 = Norm[A - A2, "Frobenius"];
error3 = Norm[A - A3, "Frobenius"];

Print["Approximation errors:"];
Print["Error for rank-1: ", error1];
Print["Error for rank-2: ", error2];
Print["Error for rank-3: ", error3];

(*Verify that the error matches the discarded singular values*)
theoreticalError1 = Sqrt[Total[singularValues[[2 ;;]]^2]];
theoreticalError2 = Sqrt[Total[singularValues[[3 ;;]]^2]];
theoreticalError3 = Sqrt[Total[singularValues[[4 ;;]]^2]];

Print["Do errors match theoretical predictions?"];
Print["Rank-1: ", Abs[error1 - theoreticalError1] < 10^-10];
Print["Rank-2: ", Abs[error2 - theoreticalError2] < 10^-10];
Print["Rank-3: ", Abs[error3 - theoreticalError3] < 10^-10];
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 3 & 1 & 1 & 1 \\ 1 & 3 & 1 & 1 \\ 1 & 1 & 3 & 1 \\ 1 & 1 & 1 & 3 \end{pmatrix}$$

```
Singular values: {6, 2, 2, 2}

Rank-1 approximation:
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \end{pmatrix}$$

```
Rank-2 approximation:
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \frac{5}{2} & \frac{3}{2} & \frac{3}{2} & \frac{1}{2} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{1}{2} & \frac{3}{2} & \frac{3}{2} & \frac{5}{2} \end{pmatrix}$$

```
Rank-3 approximation:
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} \frac{17}{6} & \frac{3}{2} & \frac{5}{6} & \frac{5}{6} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{2} & \frac{3}{2} \\ \frac{5}{6} & \frac{3}{2} & \frac{17}{6} & \frac{5}{6} \\ \frac{5}{6} & \frac{3}{2} & \frac{5}{6} & \frac{17}{6} \end{pmatrix}$$

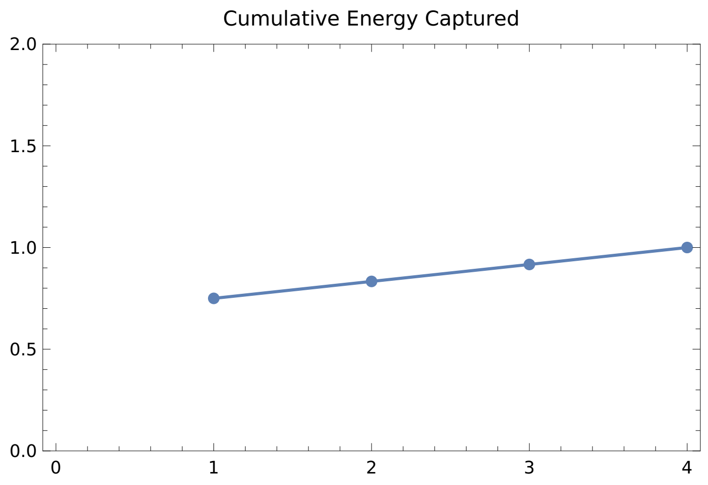
```
Approximation errors:

Error for rank-1: 2 √3

Error for rank-2: 2 √2

Error for rank-3: 2

Do errors match theoretical predictions?

Rank-1: True

Rank-2: True

Rank-3: True
```

Visualization: Energy Captured in Low-Rank Approximations

In[∘]:=
```
(*Calculate the "energy" captured by each singular value*)
totalEnergy = Total[singularValues^2];
energyVector = Accumulate[singularValues^2] / totalEnergy;

(*Plot the cumulative energy*)
energyPlot =
  ListPlot[energyVector, PlotMarkers → Automatic, Joined → True, PlotRange → {0, 2},
   PlotLabel → "Cumulative Energy Captured", AxesLabel → {"k (rank)", "Energy Fraction"},
   GridLines → {ranks, None}, PlotTheme → "Detailed"];

(*Find how many singular values needed for 90%,95%,99% energy*)
k90 = FirstPosition[energyVector, x_ /; x ≥ 0.9][[1]];
k95 = FirstPosition[energyVector, x_ /; x ≥ 0.95][[1]];
k99 = FirstPosition[energyVector, x_ /; x ≥ 0.99][[1]];

Print["Number of singular values needed for:"];
Print["90% energy: ", k90];
Print["95% energy: ", k95];
Print["99% energy: ", k99];

energyPlot
```

```
Number of singular values needed for:

90% energy: 3

95% energy: 4

99% energy: 4
```

Out[•]=



Eigenvalues, Eigenvectors and Spectral Theorems

Definition

Eigenvalues and eigenvectors are fundamental concepts in linear algebra that provide deep insights into the behavior of linear transformations.

Basic Definition

For a square matrix A, a non-zero vector v is an eigen-vector of A if there exists a scalar λ (lambda) such that $A \cdot v = \lambda \cdot v$.

The scalar λ is called the eigen value corresponding to the eigenvector v. This equation tells us that when A acts on v, it only changes its magnitude by a factor of λ, not its direction.

Characteristic Polynomial and Eigenvalue Computation

Eigen values can be found by solving the characteristic equation:

```
In[]:= (*Define a sample matrix*)A = {{4, 2}, {1, 3}};
MatrixForm[A]

(*Compute the characteristic polynomial*)
charPoly = CharacteristicPolynomial[A, λ]
Print["Characteristic polynomial: ", charPoly]

(*Find the eigenvalues by solving the characteristic equation*)
eigenvalues = Solve[charPoly == 0, λ]
Print["Eigenvalues: ", eigenvalues]

(*Use built-in function to find eigenvalues*)
eigvals = Eigenvalues[A]
Print["Eigenvalues (using built-in function): ", eigvals]
```

Out[]//MatrixForm=

$$\begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix}$$

Out[]= $10 - 7\lambda + \lambda^2$

Characteristic polynomial: $10 - 7\lambda + \lambda^2$

Out[]= {{λ → 2}, {λ → 5}}

```
Eigenvalues: {{λ → 2}, {λ → 5}}
```

Out[]= {5, 2}

```
Eigenvalues (using built-in function): {5, 2}
```

This equation produces the characteristic polynomial, whose roots are the eigenvalues of A.

Computing Eigenvectors

Once we know an eigenvalue, we can find the corresponding eigenvector v by solving: $(A - \lambda I) \cdot v = 0$

```
In[•]:= (*Find eigenvectors for each eigenvalue*)
eigenvectors = {};
For[i = 1, i ≤ Length[eigvals], i++,
 (*Set up the homogeneous system (A-λI)v=0*)λ = eigvals⟦i⟧;
 system = A - λ * IdentityMatrix[Length[A]];
 (*Solve the homogeneous system*)solution = NullSpace[system];
 AppendTo[eigenvectors, solution⟦1⟧];]

(*Display eigenvectors*)
Print["Eigenvectors:"];
Do[Print["For eigenvalue λ = ", eigvals⟦i⟧, ": ", eigenvectors⟦i⟧], {i, 1, Length[eigvals]}]

(*Use built-in function to compute eigenvectors*)
{vals, vecs} = Eigensystem[A];
Print["Eigenvalues and eigenvectors (using built-in function):"];
Do[Print["λ = ", vals⟦i⟧, ", v = ", vecs⟦i⟧], {i, 1, Length[vals]}]

(*Verify the eigenvector equation A·v=λ·v*)
Print["Verification of A·v = λ·v:"];
Do[v = vecs⟦i⟧;
 λ = vals⟦i⟧;
 Print["For eigenpair ", i, ":", MatrixForm[A.v], " ≈ ", MatrixForm[λ * v]], {i, 1, Length[vals]}]
```

```
Eigenvectors:
For eigenvalue λ = 5: {2, 1}
For eigenvalue λ = 2: {-1, 1}
Eigenvalues and eigenvectors (using built-in function):
λ = 5, v = {2, 1}
λ = 2, v = {-1, 1}
Verification of A·v = λ·v:
```

For eigenpair 1: $\begin{pmatrix} 10 \\ 5 \end{pmatrix} \approx \begin{pmatrix} 10 \\ 5 \end{pmatrix}$

For eigenpair 2: $\begin{pmatrix} -2 \\ 2 \end{pmatrix} \approx \begin{pmatrix} -2 \\ 2 \end{pmatrix}$

Properties of Eigenvalues and Eigenvectors

Several important properties related to eigenvalues and eigenvectors:

- The trace of A equals the sum of its eigenvalues

```
In[∘]:= Print["Trace of A: ", Tr[A]];
        Print["Sum of eigenvalues: ", Total[eigvals]];

        Trace of A: 7

        Sum of eigenvalues: 7
```

- The determinant of A equals the product of its eigenvalues

```
In[∘]:= Print["Determinant of A: ", Det[A]];
        Print["Product of eigenvalues: ", Times @@ eigvals];

        Determinant of A: 10

        Product of eigenvalues: 10
```

- A is invertible if and only if all its eigenvalues are non-zero
- Similar matrices have the same eigenvalues

```
In[∘]:= P = {{2, 1}, {1, 1}};
        Pinv = Inverse[P];
        B = P.A.Pinv;

        Print["Similar matrix B = P·A·P⁻¹:"];
        MatrixForm[B]

        Print["Eigenvalues of B: ", Eigenvalues[B]];
        Print["Eigenvalues of A: ", eigvals];

        Similar matrix B = P·A·P⁻¹:
Out[∘]//MatrixForm=
```

$$\begin{pmatrix} 2 & 5 \\ 0 & 5 \end{pmatrix}$$

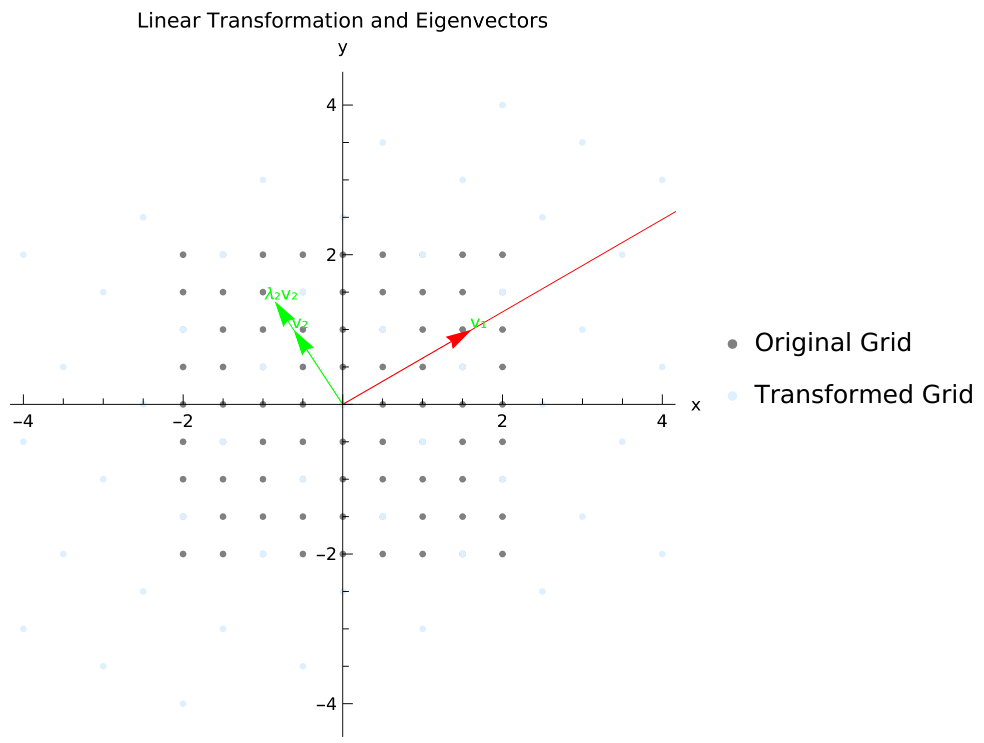
```
        Eigenvalues of B: {5, 2}

        Eigenvalues of A: {5, 2}
```

Visualizing Eigenvectors and Eigenvalues in 2D

For a 2 × 2 matrix, we can visualize how the linear transformation affects vectors in the plane, highlighting the special role of eigenvectors.

```
In[•]:= (*Create a visualization of eigenvectors and the transformation*)
A = {{3, 1}, {1, 2}};
{vals, vecs} = Eigensystem[A];

(*Create a grid of points*)
gridPoints = Flatten[Table[{i, j}, {i, -2, 2, 0.5}, {j, -2, 2, 0.5}], 1];
transformedGrid = Map[A.# &, gridPoints];

(*Define the eigenvectors and their images*)
ev1 = vecs[[1]];
ev2 = vecs[[2]];
λ1 = vals[[1]];
λ2 = vals[[2]];

(*Create the visualization*)
eigenvectorPlot =
  Show[(*Draw the grid and its transformation*)ListPlot[{gridPoints, transformedGrid},
    PlotStyle → {{Gray, PointSize[0.01]}, {LightBlue, PointSize[0.01]}},
    PlotLegends → {"Original Grid", "Transformed Grid"}],
   (*Draw the eigenvectors and their transformations*)
   Graphics[{Red, Arrow[{{0, 0}, ev1}], Arrow[{{0, 0}, λ1 * ev1}], Green, Arrow[{{0, 0}, ev2}],
     Arrow[{{0, 0}, λ2 * ev2}], Text["v₁", ev1 + {0.1, 0.1}], Text["λ₁v₁", λ1 * ev1 + {0.1, 0.1}],
     Text["v₂", ev2 + {0.1, 0.1}], Text["λ₂v₂", λ2 * ev2 + {0.1, 0.1}]}], PlotRange → {{-4, 4}, {-4, 4}},
   AspectRatio → 1, PlotLabel → "Linear Transformation and Eigenvectors",
   AxesLabel → {"x", "y"}];

eigenvectorPlot
```

Out[∘]=



The Spectral Theorem for Symmetric Matrices

The spectral theorem is a fundamental result that provides a canonical form for certain classes of matrices.

Statement of the Spectral Theorem

For a symmetric matrix A, the spectral theorem states that.

- All eigenvalues of A are real numbers.
- Eigenvectors corresponding to distinct eigenvalues are orthogonal.
- A can be diagonalized by an orthogonal matrix $Q: A = Q D Q^T$, where D is a diagonal matrix containing the eigenvalues of A.

Verification of the Spectral Theorem

In[∘]:=
```
(*Create a symmetric matrix*)
A = {{1, -1}, {-1, 1}}
MatrixForm[A]

(*Verify that A is symmetric*)
```

```
Print["Is A symmetric? ", A == Transpose[A]]

(*Find eigenvalues and eigenvectors*)
{vals, vecs} = Eigensystem[A];
Print["Eigenvalues: ", vals];

(*Check that eigenvalues are real*)
Print["Are all eigenvalues real? ", And @@ (Im[#] == 0 & /@ vals)]

(*Normalize eigenvectors*)
normVecs = Map[Normalize, vecs];

(*Check orthogonality of eigenvectors*)
Print["Dot products between eigenvectors:"];
Table[normVecs[[i]].normVecs[[j]], {i, 1, Length[normVecs]}, {j, 1, Length[normVecs]}] // MatrixForm

(*Construct the orthogonal matrix Q*)
Q = Transpose[normVecs];
Print["Orthogonal matrix Q:"];
MatrixForm[Q]

(*Verify that Q is orthogonal:Q^T·Q=I*)
Print["Q^T·Q = "];
MatrixForm[Transpose[Q].Q]

(*Construct the diagonal matrix D*)
Diag = DiagonalMatrix[vals];
Print["Diagonal matrix D:"];
MatrixForm[Diag]

(*Verify the spectral decomposition:A=Q·D·Q^T*)
Print["Q·D·Q^T = "];
MatrixForm[Q.Diag.Transpose[Q]]

Print["Original matrix A:"];
MatrixForm[A]

Print["Is A = Q·D·Q^T? ", And @@ Flatten[
   Table[Abs[A[[i, j]] - (Q.Diag.Transpose[Q])[[i, j]]] < 10^-10, {i, 1, Length[A]}, {j, 1, Length[A]}]]]
```

```
Out[•]= {{1, -1}, {-1, 1}}
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

```
Is A symmetric? True

Eigenvalues: {2, 0}

Are all eigenvalues real? True

Dot products between eigenvectors:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
Orthogonal matrix Q:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

```
Q^T·Q =
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
Diagonal matrix D:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

```
Q·D·Q^T =
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

```
Original matrix A:
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

```
Is A = Q·D·Q^T? True
```

Spectral Decomposition

The spectral decomposition expresses a symmetric matrix A as:

$A = \sum_{i=1}^{n} \lambda_i \, v_i \, v_i^T$, where λ_i are the eigenvalues and v_i are the corresponding normalized eigenvectors.

```
In[∘]:= (*Compute the spectral decomposition explicitly*)
        A = {{1, -1}, {-1, 1}};
        {vals, vecs} = Eigensystem[A];
        normVecs = Map[Normalize, vecs];

        (*Construct the spectral decomposition*)
        spectralSum = Sum[vals[[i]] * KroneckerProduct[normVecs[[i]], normVecs[[i]]], {i, 1, Length[vals]}];

        Print["Spectral decomposition:"];
        MatrixForm[spectralSum]

        Print["Original matrix:"];
        MatrixForm[A]

        Print["Difference (should be approximately zero):"];
        MatrixForm[A - spectralSum] // FullSimplify

        Spectral decomposition:
Out[∘]//MatrixForm=
```

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

```
        Original matrix:
Out[∘]//MatrixForm=
```

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

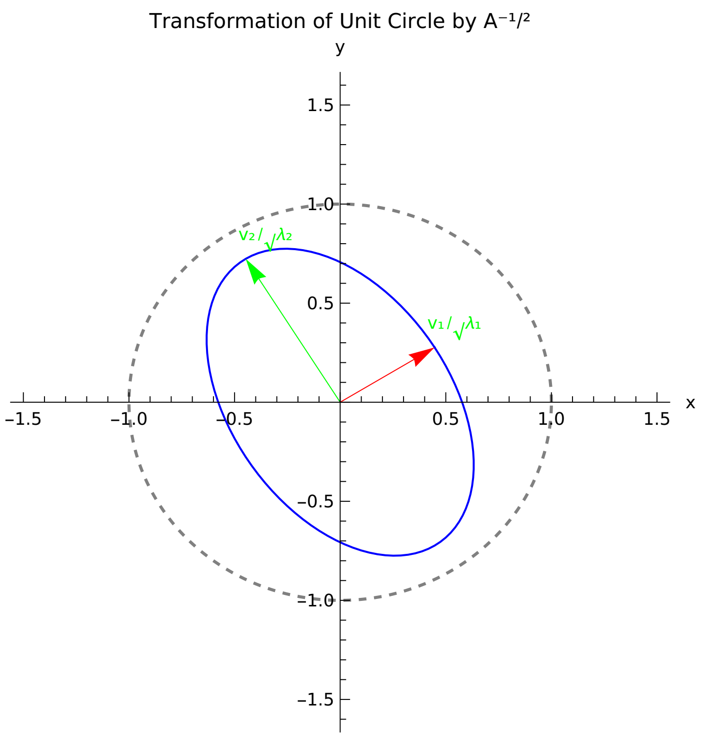
```
        Difference (should be approximately zero):
Out[∘]//MatrixForm=
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

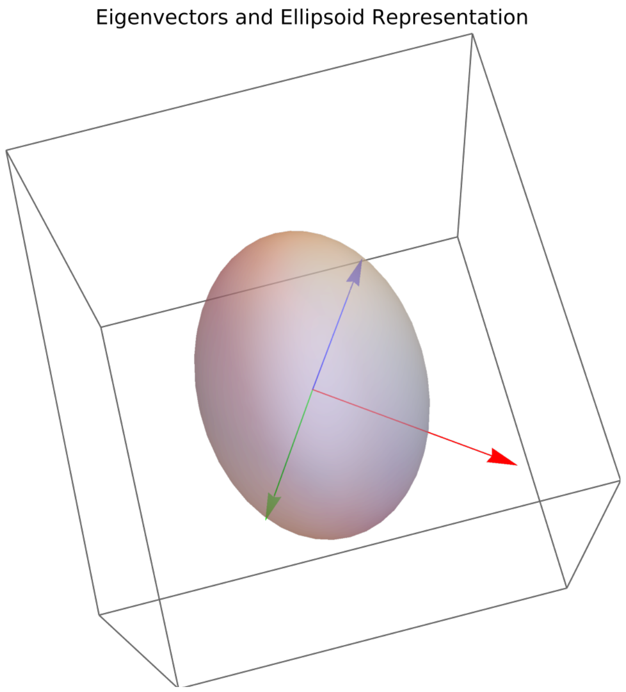
Visualizing the Spectral Theorem in 2D and 3D

```
In[∘]:= (*Visualize spectral theorem in 2D–Ellipse axes aligned with eigenvectors*)
A = {{3, 1}, {1, 2}};
{vals, vecs} = Eigensystem[A];
normVecs = Map[Normalize, vecs];

(*Define an ellipse using the eigenvectors and eigenvalues*)
ellipsePoints = Table[Sqrt[1 / vals[[1]]] * normVecs[[1]] * Cos[t] +
    Sqrt[1 / vals[[2]]] * normVecs[[2]] * Sin[t], {t, 0, 2 Pi, Pi / 50}];

(*Create visualization*)
spectralEllipsePlot =
 Show[(*Draw the ellipse*)ListLinePlot[ellipsePoints, PlotStyle → {Blue, Thickness[0.003]}],
  (*Draw the eigenvectors scaled by eigenvalue*)
  Graphics[{Red, Arrow[{{0, 0}, Sqrt[1 / vals[[1]]] * normVecs[[1]]}],
    Green, Arrow[{{0, 0}, Sqrt[1 / vals[[2]]] * normVecs[[2]]}],
    Text["v₁/√λ₁", Sqrt[1 / vals[[1]]] * normVecs[[1]] + {0.1, 0.1}],
    Text["v₂/√λ₂", Sqrt[1 / vals[[2]]] * normVecs[[2]] + {0.1, 0.1}]}],
  (*Draw the unit circle for reference*)ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2 Pi},
   PlotStyle → {Gray, Dashed}], PlotRange → {{-1.5, 1.5}, {-1.5, 1.5}}, AspectRatio → 1,
  PlotLabel → "Transformation of Unit Circle by A⁻¹/²", AxesLabel → {"x", "y"}]

(*Visualize in 3D–Eigenvectors of a 3×3 symmetric matrix*)
A3D = {{3, 1, 0}, {1, 2, 0.5}, {0, 0.5, 2}};
{vals3D, vecs3D} = Eigensystem[A3D];
normVecs3D = Map[Normalize, vecs3D];

(*Create a 3D ellipsoid*)
ellipsoid3D = Graphics3D[
   {Opacity[0.5], Ellipsoid[{0, 0, 0}, {1 / Sqrt[vals3D[[1]]], 1 / Sqrt[vals3D[[2]]], 1 / Sqrt[vals3D[[3]]]}]}];

(*Create arrows for the eigenvectors*)
arrows3D = Graphics3D[{Red, Arrow[{{0, 0, 0}, normVecs3D[[1]]}], Green,
    Arrow[{{0, 0, 0}, normVecs3D[[2]]}], Blue, Arrow[{{0, 0, 0}, normVecs3D[[3]]}]}];

(*Display both together*)
Show[ellipsoid3D, arrows3D, PlotRange → {{-1, 1}, {-1, 1}, {-1, 1}},
 BoxRatios → {1, 1, 1}, PlotLabel → "Eigenvectors and Ellipsoid Representation"]
```

Out[∘]=



Out[∘]=



Applications

Eigenvalues and eigenvectors have numerous applications across mathematics, physics, engineering, and data science.

Principal Component Analysis

PCA uses the eigen-decomposition of the covariance matrix to find the principal directions of variation in the data .

```
In[•]:= (*Generate sample 2D data with correlation*)
SeedRandom[42];
covMatrix = {{2, 1.5}, {1.5, 3}};
data = RandomVariate[MultinormalDistribution[{0, 0}, covMatrix], 200];

(*Compute the sample covariance matrix*)
sampleCov = Covariance[data];
Print["Sample covariance matrix:"];
MatrixForm[sampleCov]

(*Find eigenvalues and eigenvectors of the covariance matrix*)
{pcaVals, pcaVecs} = Eigensystem[sampleCov];
Print["Eigenvalues (variances along principal components): ", pcaVals];

(*Sort eigenvalues and eigenvectors in descending order*)
indices = Ordering[pcaVals, All, Greater];
pcaVals = pcaVals⟦indices⟧;
pcaVecs = pcaVecs⟦indices⟧;

(*Visualize the data and principal components*)
pcaPlot = Show[ListPlot[data, PlotStyle → {Black, PointSize[0.01]}, PlotLabel → "PCA Example"],
  (*Add arrows for principal components*)
  Graphics[{Red, Arrow[{{0, 0}, 2 * Sqrt[pcaVals⟦1⟧] * pcaVecs⟦1⟧}],
    Green, Arrow[{{0, 0}, 2 * Sqrt[pcaVals⟦2⟧] * pcaVecs⟦2⟧}],
    Text["PC1", 2 * Sqrt[pcaVals⟦1⟧] * pcaVecs⟦1⟧ + {0.2, 0.2}],
    Text["PC2", 2 * Sqrt[pcaVals⟦2⟧] * pcaVecs⟦2⟧ + {0.2, 0.2}]}],
  PlotRange → {{-5, 5}, {-5, 5}}, AspectRatio → 1, AxesLabel → {"x", "y"}]

(*Project the data onto principal components*)
projectionMatrix = Transpose[pcaVecs];
projectedData = Map[projectionMatrix.# &, data];

(*Visualize projected data*)
projectedPlot = ListPlot[projectedData, PlotStyle → {Red, PointSize[0.01]},
  PlotLabel → "Data Projected onto Principal Components",
  AxesLabel → {"PC1", "PC2"}, AspectRatio → 1, PlotRange → {{-5, 5}, {-5, 5}}]

(*Show both plots side by side*)
```

```
Grid[{{pcaPlot, projectedPlot}}]

(*Reconstruct data using only the first principal component*)
reconstructionPC1 = Map[pcaVecs⟦1⟧ * (pcaVecs⟦1⟧.#) &, data];

(*Visualize reconstruction with one principal component*)
reconstructionPlot = Show[ListPlot[data, PlotStyle → {Gray, PointSize[0.01]}],
   ListPlot[reconstructionPC1, PlotStyle → {Red, PointSize[0.01]}],
   PlotLabel → "Reconstruction Using First Principal Component",
   AxesLabel → {"x", "y"}, PlotRange → {{-5, 5}, {-5, 5}}, AspectRatio → 1];

reconstructionPlot
```

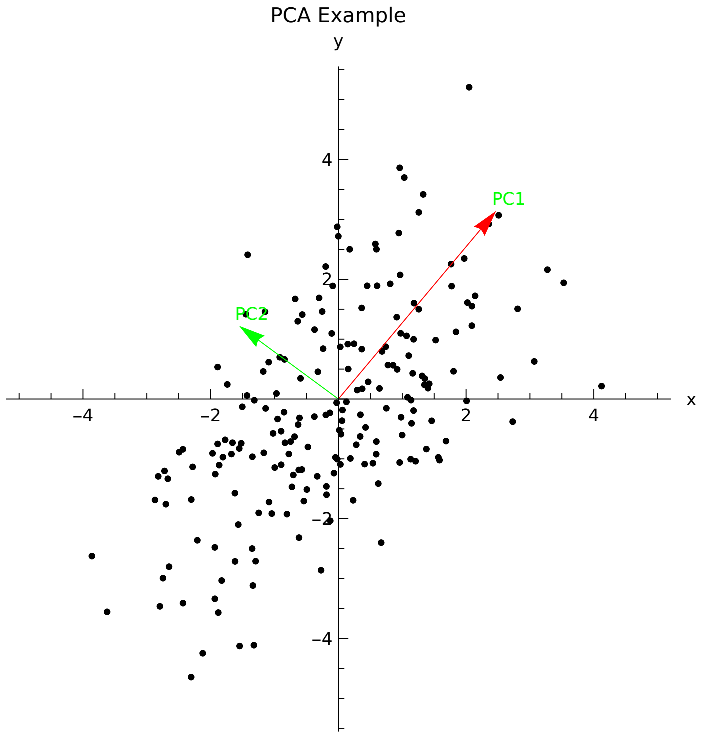
```
Sample covariance matrix:
```

Out[∘]//MatrixForm=

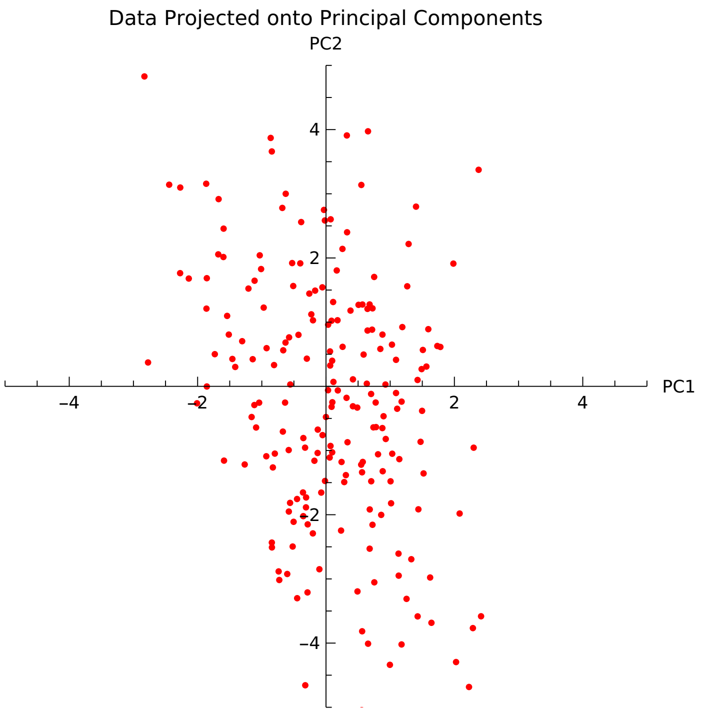
$$\begin{pmatrix} 2.13111 & 1.46471 \\ 1.46471 & 2.84316 \end{pmatrix}$$

```
Eigenvalues (variances along principal components): {3.99449, 0.979775}
```

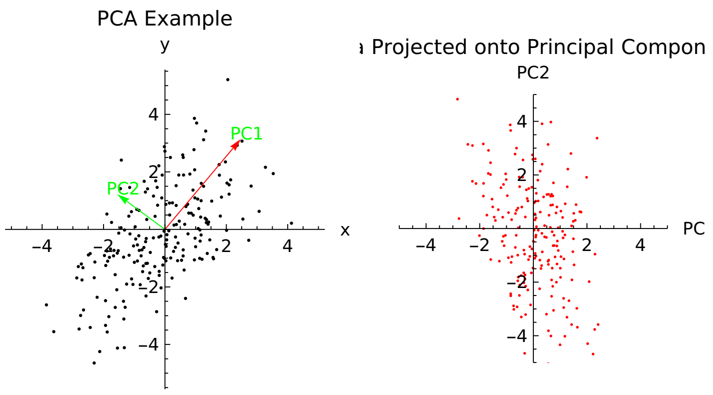
Out[∘]=

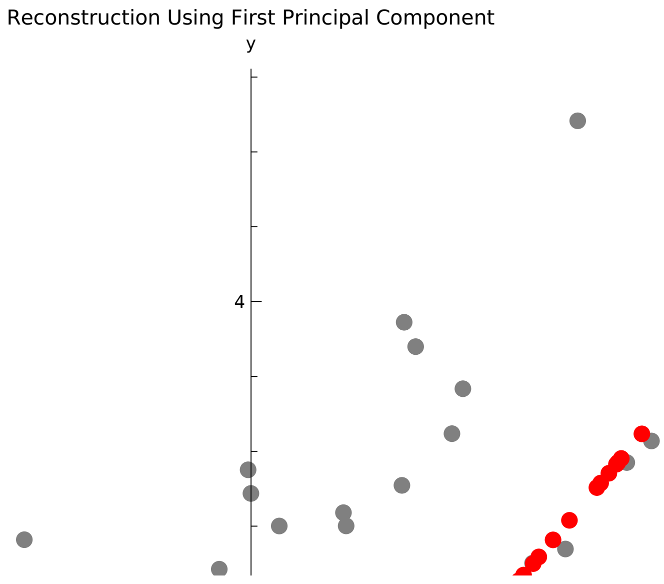


Out[•]=

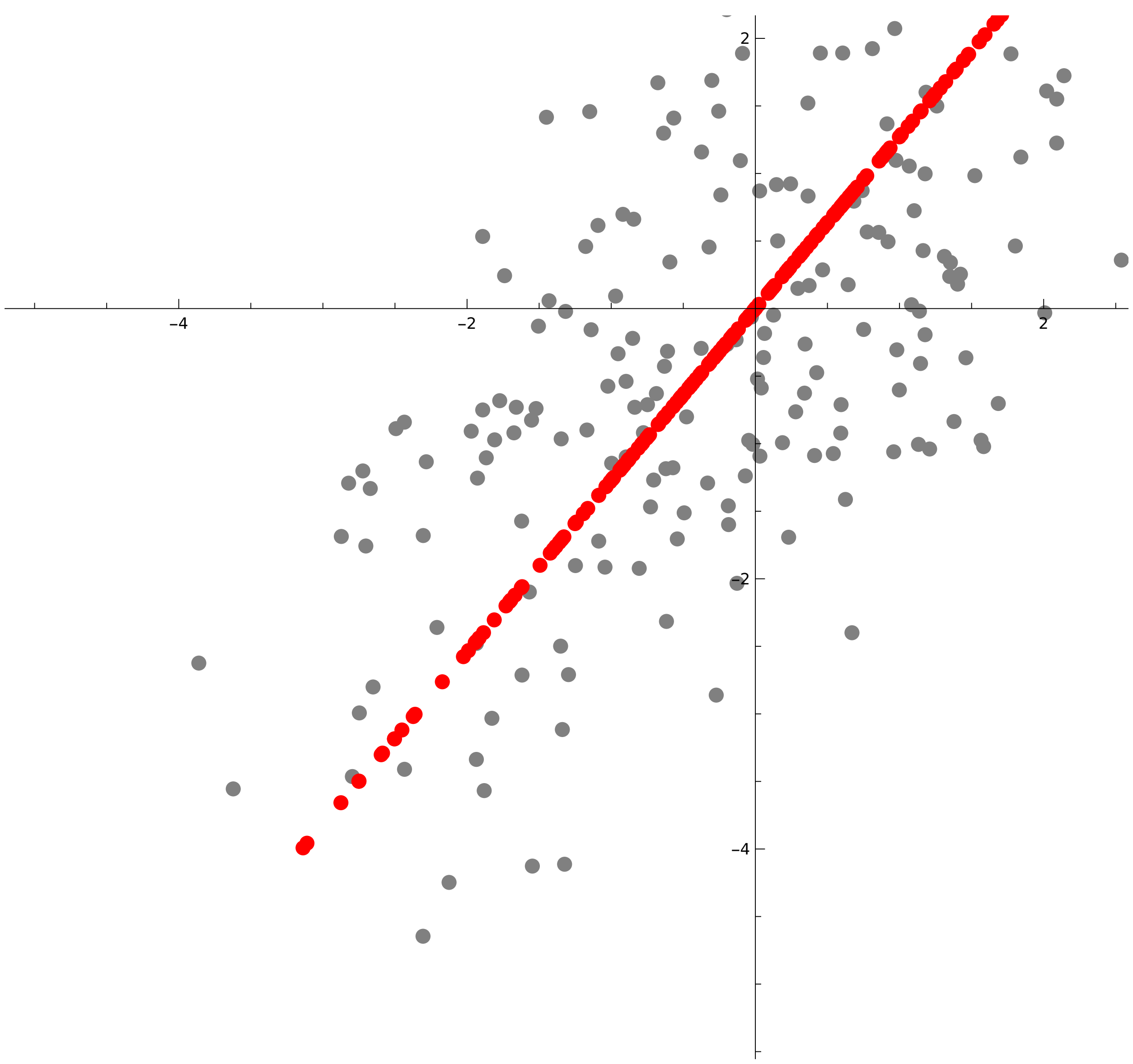

Data Projected onto Principal Components
PC2
PC1
−4
−2
2
4
4
2
−2
−4

Out[•]=


PCA Example
y
x
PC1
PC2
Projected onto Principal Compon
PC2
PC


Reconstruction Using First Principal Component
y
4

Out[•]=



Dynamical Systems

Eigenvalues determine the stability of fixed points in dynamical systems.

In[•]:=

```
ClearAll[x, y]
(*Define a simple linear dynamical system:dx/dt=Ax*)
A = {{2, 1}, {-2, -1}};
MatrixForm[A]

(*Find eigenvalues and eigenvectors*)
{vals, vecs} = Eigensystem[A];
Print["Eigenvalues: ", vals];
Print["Eigenvectors: "];
```

```
Map[MatrixForm, vecs]

classification = Which[AllTrue[Re[vals], # < 0 &],
    "Stable node/spiral (all eigenvalues have negative real parts)", AllTrue[Re[vals],
     # > 0 &], "Unstable node/spiral (all eigenvalues have positive real parts)",
    AnyTrue[Re[vals], # == 0 &], "Center/neutral (at least one eigenvalue has zero real part)",
    True, "Saddle point (mixed positive and negative real parts)"];

Print["Classification of fixed point at origin: ", classification];


(*Define the vector field*)
vectorField[{x_, y_}] := A.{x, y};

(*Create a stream plot of the vector field*)
streamPlot = StreamPlot[vectorField[{x, y}], {x, -2, 2},
  {y, -2, 2}, StreamPoints → Fine, StreamScale → Small, StreamStyle → Blue,
  PlotLabel → "Phase Portrait of Linear System", AxesLabel → {"x", "y"}]

(*Add eigenvectors to the plot*)
dynamicsPlot =
 Show[streamPlot, Graphics[{Red, Arrow[{{0, 0}, vecs[[1]]}], Green, Arrow[{{0, 0}, vecs[[2]]}],
    Text["v₁", vecs[[1]] + {0.1, 0.1}], Text["v₂", vecs[[2]] + {0.1, 0.1}]}]]

(*Solve the system for some initial conditions*)
tmax = 3;
initialConditions = {{1, 1}, {-1, 1}, {1, -1}, {-1, -1}, {0.5, -0.2}};

trajectories = Table[NDSolve[{x '[t] == A[[1, 1]] * x[t] + A[[1, 2]] * y[t], y '[t] == A[[2, 1]] * x[t] + A[[2, 2]] * y[t],
      x[0] == ic[[1]], y[0] == ic[[2]]}, {x, y}, {t, 0, tmax}], {ic, initialConditions}];

(*Plot trajectories*)
trajectoriesPlot =
  Show[dynamicsPlot, Table[ParametricPlot[Evaluate[{x[t], y[t]} /. trajectories[[i, 1]]],
     {t, 0, tmax}, PlotStyle → {Orange, Thickness[0.003]}],
    {i, 1, Length[initialConditions]}], PlotRange → {{-2, 2}, {-2, 2}}];

trajectoriesPlot
```

Out[∘]//MatrixForm=

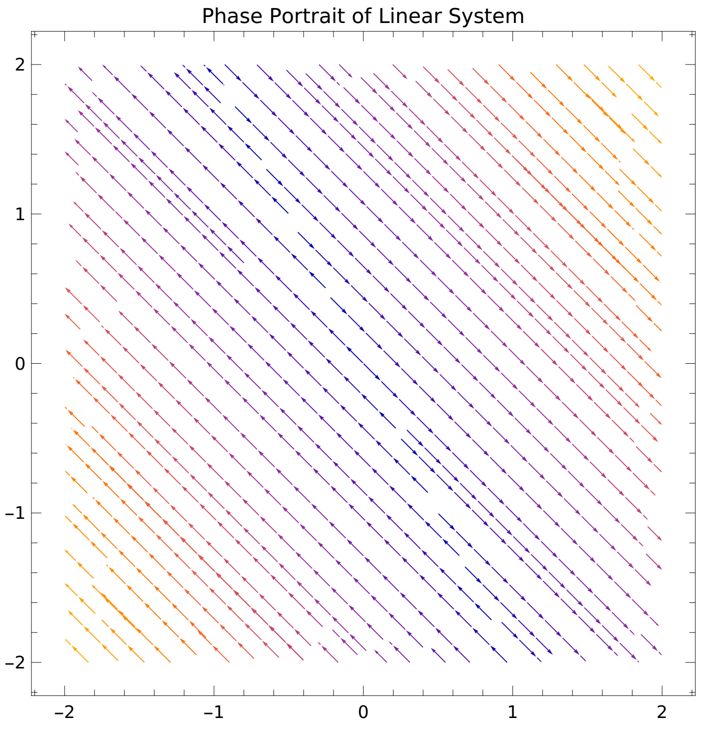
$$\begin{pmatrix} 2 & 1 \\ -2 & -1 \end{pmatrix}$$

```
Eigenvalues: {1, 0}

Eigenvectors:
```

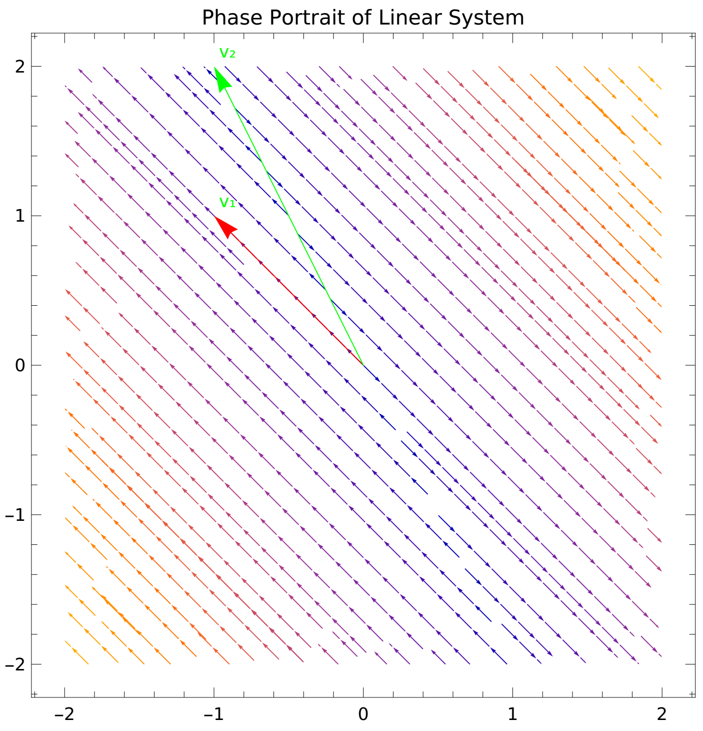
Out[∘]= $\left\{\begin{pmatrix}-1\\1\end{pmatrix},\begin{pmatrix}-1\\2\end{pmatrix}\right\}$

```
Classification of fixed point at origin:
 Center/neutral (at least one eigenvalue has zero real part)
```

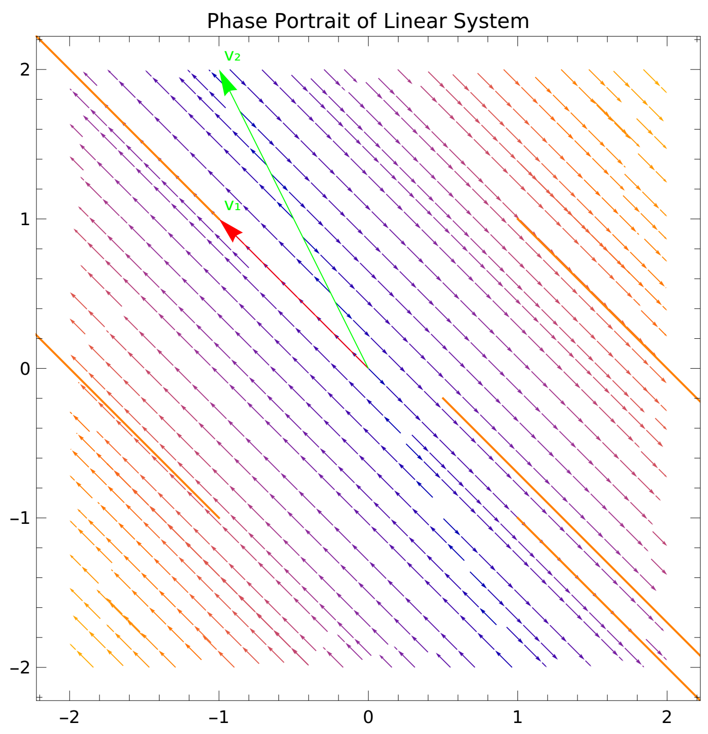
Out[∘]=



Out[∘]=



Out[∘]=



Differential Equations - Modal Analysis

Eigenvalues and eigenvectors can be used to solve systems of differential equations.

In[∘]:=
```
(*Consider a coupled spring-mass system modeled as d²x/dt²=Ax*)
M = {{2, 0}, {0, 3}}; (*Mass matrix*)
K = {{3, -1}, {-1, 2}}; (*Stiffness matrix*)
A = -Inverse[M].K; (*System matrix*)

Print["System matrix A = -M⁻¹K:"];
MatrixForm[A]

(*Find eigenvalues and eigenvectors*)
{vals, vecs} = Eigensystem[A];
ω = Sqrt[vals]; (*Natural frequencies*)

Print["Eigenvalues: ", vals];
Print["Natural frequencies: ", ω];
Print["Eigenvectors (mode shapes): "];
Map[MatrixForm, vecs]

(*Modal matrix (matrix of eigenvectors)*)
P = Transpose[vecs];
```

```
Print["Modal matrix P:"];
MatrixForm[P]

(*Verify that P diagonalizes A*)
Diag = P.A.Inverse[P];
Print["Diagonalized system matrix P·A·P⁻¹:"];
MatrixForm[Diag]

(*Visualize the mode shapes*)
modeShapePlot = GraphicsRow[
  Table[Graphics[{Line[{{0, 0}, {1, 0}}], (*Base line*)Red, PointSize[0.02], Point[{0.25, 0}],
     Point[{0.75, 0}], (*Equilibrium positions*)Blue, Arrow[{{0.25, 0}, {0.25, vecs[[i, 1]] / 2}}],
     (*Displacement of mass 1*)Blue, Arrow[{{0.75, 0}, {0.75, vecs[[i, 2]] / 2}}]
     (*Displacement of mass 2*)}, PlotRange → {{0, 1}, {-0.5, 0.5}}, PlotLabel →
     "Mode " <> ToString[i] <> " (ω = " <> ToString[N[ω[[i]]]] <> ")", AspectRatio → 1], {i, 1, 2}]]

(*Simulate the response of the system*)
tmax = 20;
x0 = {1, 0}; (*Initial displacements*)
v0 = {0, 0}; (*Initial velocities*)

(*Transform initial conditions to modal coordinates*)
modalIC = Inverse[P].x0;
modalVel = Inverse[P].v0;

(*Modal solution*)
modalSol[t_] :=
  Table[modalIC[[i]] * Cos[ω[[i]] * t] + modalVel[[i]] / ω[[i]] * Sin[ω[[i]] * t], {i, 1, Length[vals]}];

(*Transform back to physical coordinates*)
physicalSol[t_] := P.modalSol[t];

(*Plot the solution*)
responsePlot = Plot[Evaluate[physicalSol[t]], {t, 0, tmax}, PlotStyle → {Red, Blue},
  PlotLegends → {"Mass 1", "Mass 2"}, PlotLabel → "Response of Coupled Spring-Mass System",
  AxesLabel → {"Time", "Displacement"}, PlotRange → All]

Grid[{{modeShapePlot}, {responsePlot}}]
```

```
System matrix A = -M⁻¹K:
```

Out[∘]//MatrixForm=

$$\begin{pmatrix} -\frac{3}{2} & \frac{1}{2} \\ \frac{1}{3} & -\frac{2}{3} \end{pmatrix}$$

Eigenvalues: $\left\{-\frac{5}{3}, -\frac{1}{2}\right\}$

Natural frequencies: $\left\{i \sqrt{\frac{5}{3}}, \frac{i}{\sqrt{2}}\right\}$

Eigenvectors (mode shapes):

Out[∘]= $\left\{\begin{pmatrix} -3 \\ 1 \end{pmatrix}, \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix}\right\}$

Modal matrix P:

Out[∘]//MatrixForm=

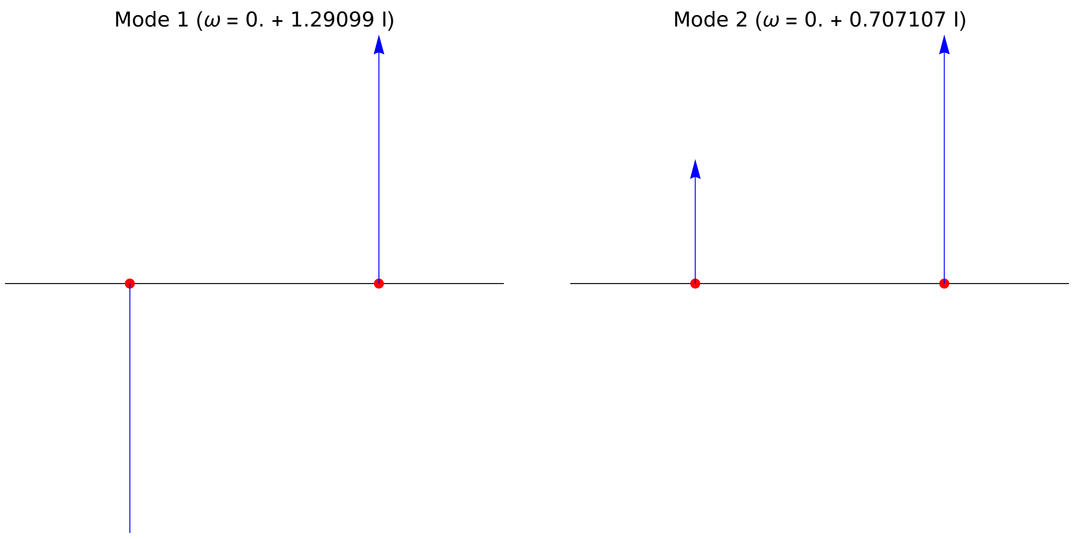
$$\begin{pmatrix} -3 & \frac{1}{2} \\ 1 & 1 \end{pmatrix}$$

Diagonalized system matrix P·A·P⁻¹:

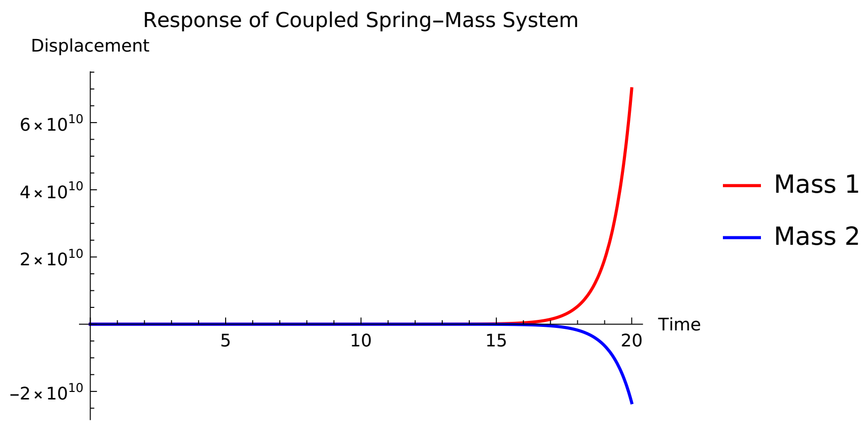
Out[∘]//MatrixForm=

$$\begin{pmatrix} -\frac{13}{7} & -\frac{19}{21} \\ \frac{2}{7} & -\frac{13}{42} \end{pmatrix}$$

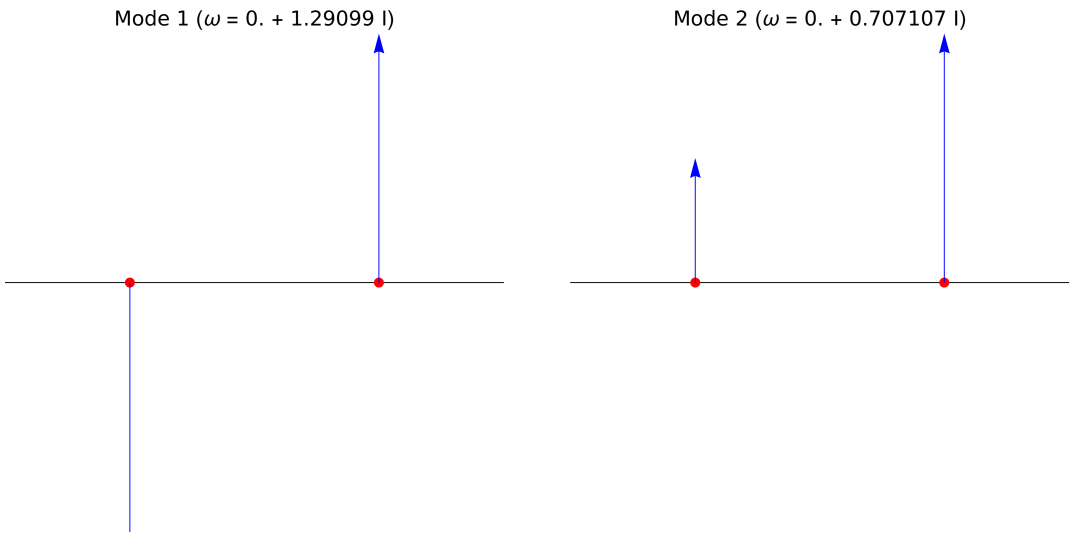
Out[∘]=

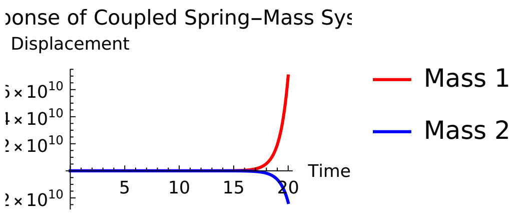


Out[∘]=



Out[∘]=





Markov Chains and the Steady State

For a stochastic matrix (transition matrix of a Markov chain), the eigenvector corresponding to eigenvalue 1 represents the steady-state probabilities.

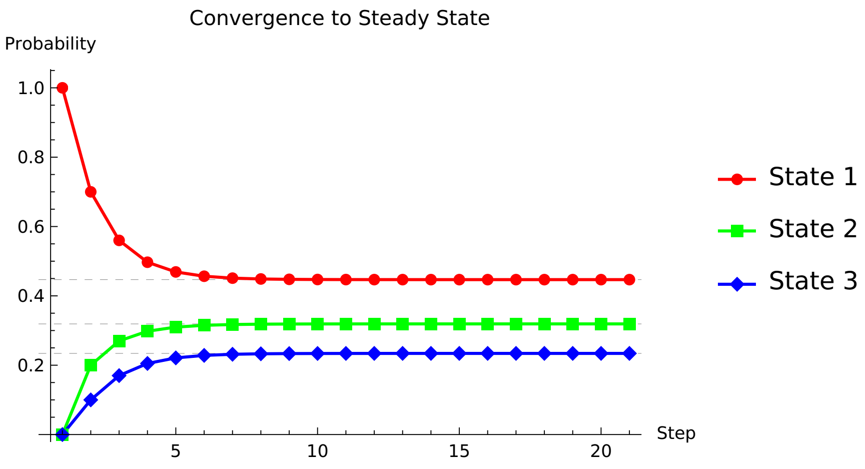
```
In[•]:= (*Define a transition matrix for a simple Markov chain*)
P = {{0.7, 0.2, 0.3}, {0.2, 0.5, 0.3}, {0.1, 0.3, 0.4}};
MatrixForm[P]

(*Verify that P is a stochastic matrix (rows sum to 1)*)
Print["Row sums: ", Total[P, {2}]];

(*Find eigenvalues and eigenvectors*)
{vals, vecs} = Eigensystem[P];
Print["Eigenvalues: ", vals];

(*Find the index of eigenvalue closest to 1*)
idx = Ordering[Abs[vals - 1], 1][[1]];
steadyStateEigenvector = vecs[[idx]];

(*Normalize to get probabilities (sum to 1)*)
steadyState = steadyStateEigenvector / Total[steadyStateEigenvector];
Print["Steady-state distribution: ", steadyState];

(*Verify P·π=π,where π is the steady-state vector*)
Print["P·π = ", P.steadyState];
Print["π = ", steadyState];

(*Alternative method:iterate the transition matrix*)
initialState = {1, 0, 0}; (*Start in state 1*)
states = NestList[P.# &, initialState, 20];

(*Plot the convergence to steady state*)
steadyStatePlot = ListPlot[Transpose[states], Joined → True, PlotStyle → {Red, Green, Blue},
   PlotMarkers → Automatic, PlotLegends → {"State 1", "State 2", "State 3"},
   PlotLabel → "Convergence to Steady State", AxesLabel → {"Step", "Probability"},
   PlotRange → All, GridLines → {None, {steadyState[[1]], steadyState[[2]], steadyState[[3]]}},
   GridLinesStyle → Directive[Gray, Dashed]]

(*Visualize the Markov chain as a graph*)
markovGraph =
  Graph[{1 → 1, 1 → 2, 1 → 3, 2 → 1, 2 → 2, 2 → 3, 3 → 1, 3 → 2, 3 → 3}, EdgeWeight → Flatten[P],
   EdgeLabels → "EdgeWeight", VertexLabels → "Name", VertexSize → Large,
   VertexStyle → {1 → Red, 2 → Green, 3 → Blue}, EdgeStyle → Directive[Gray, Thickness[0.003]]]

Grid[{{markovGraph}, {steadyStatePlot}}]
```

Out[◦]//MatrixForm=

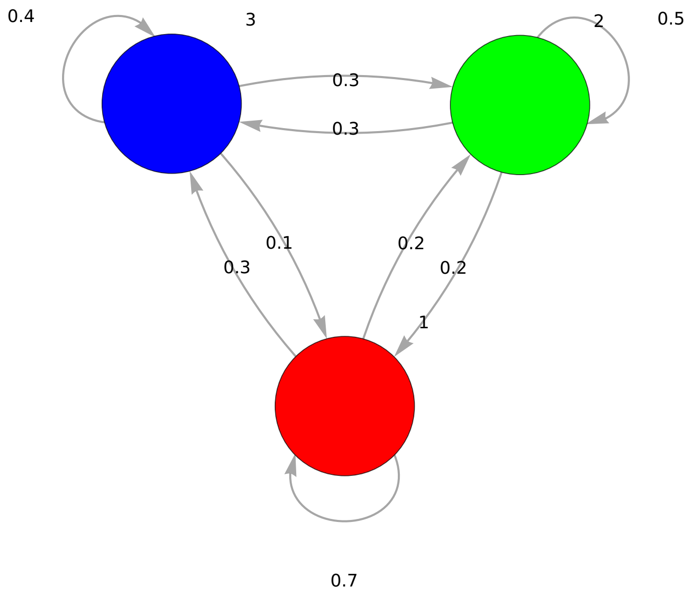
$$\begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.5 & 0.3 \\ 0.1 & 0.3 & 0.4 \end{pmatrix}$$

```
Row sums: {1.2, 1., 0.8}

Eigenvalues: {1., 0.441421, 0.158579}

Steady-state distribution: {0.446809, 0.319149, 0.234043}

P·π = {0.446809, 0.319149, 0.234043}

π = {0.446809, 0.319149, 0.234043}
```

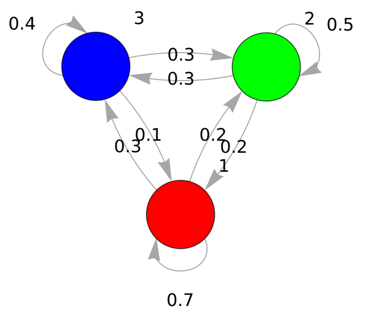
Out[◦]=

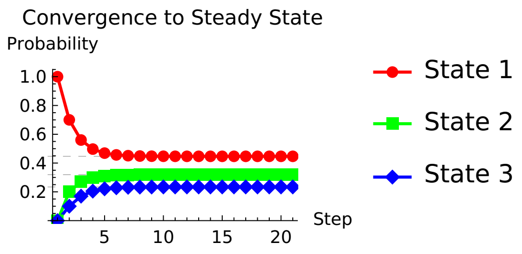


Out[◦]=



Out[∘]=





Google PageRank

In[∘]:=

```
(*Generate a random web graph with controlled density*)
n = 15; (*Number of pages*)
linkProbability = 0.2; (*Probability of a link between any two pages*)

(*Create a random adjacency matrix with no self-loops*)
SeedRandom[42]; (*Set a seed for reproducibility*)
webGraph = Table[
    If[i == j, 0, (*No self-loops*)If[RandomReal[] < linkProbability, 1, 0]], {i, 1, n}, {j, 1, n}];

(*Ensure each page has at least one outgoing link unless it's deliberately a sink*)
sinkProbability = 0.1; (*Probability of a page being a sink*)
For[i = 1, i ≤ n, i++, If[Total[webGraph[[i]]] == 0 && RandomReal[] > sinkProbability,
    (*This page has no outlinks and isn't selected to be a sink-add a random link*)
    randomTarget = RandomInteger[{1, n}];
    While[randomTarget == i, randomTarget = RandomInteger[{1, n}]];
    webGraph[[i, randomTarget]] = 1;]];

(*Display the random web graph matrix*)
Print["Random Web Graph (Adjacency Matrix):"];
MatrixForm[webGraph]
(*Create column-stochastic transition matrix*)
(*First,normalize columns to sum to 1*)
outDegrees = Total[webGraph, {1}];
stochasticMatrix =
   Table[If[outDegrees[[j]] > 0, webGraph[[i, j]] / outDegrees[[j]], 1 / Length[webGraph]],
    {i, 1, Length[webGraph]}, {j, 1, Length[webGraph]}];

(*Apply damping factor (random jump probability)*)
d = 0.85; (*Damping factor used in PageRank*)
n = Length[webGraph];
dampedMatrix = d * stochasticMatrix + (1 - d) / n * ConstantArray[1, {n, n}];
Print["PageRank transition matrix:"];
MatrixForm[dampedMatrix]

(*Compute PageRank as the dominant eigenvector*)
{vals, vecs} = Eigensystem[Transpose[dampedMatrix]];
dominantIdx = Ordering[Abs[vals], 1, Greater][[1]];
pageRank = vecs[[dominantIdx]];
pageRank = Abs[pageRank] / Total[Abs[pageRank]]; (*Ensure real values*)
Print["PageRank values: ", pageRank];

initialRank = ConstantArray[1 / n, n];
powerIter[v_, iterations_] := Nest[dampedMatrix.# &, v, iterations];
powerRank = powerIter[initialRank, 50];
powerRank = powerRank / Total[powerRank]; (*Normalize*)
Print["PageRank via power iteration: ", powerRank];

(*Visualize the web graph with PageRank values computed from power iteration*)
edgeList = Flatten[Table[If[webGraph[[i, j]] > 0, i → j, Nothing], {i, 1, n}, {j, 1, n}]];

(*Use powerRank instead of pageRank*)
pageRankGraph = Graph[edgeList, VertexSize → 0.1 + 0.5 * powerRank, VertexStyle →
     (ColorData["Rainbow"][#] & /@ Rescale[powerRank, {Min[powerRank], Max[powerRank]}, {0, 1}]),
    VertexLabels → Table[i → Style["Page " <> ToString[i] <> "\n" <>
          TextString[NumberForm[powerRank[[i]], {3, 2}]], Bold], {i, 1, n}],
    EdgeStyle → Directive[Gray, Thickness[0.0003]], GraphLayout → "SpringEmbedding"];

(*Use powerRank in bar chart too*)
pageRankBarChart = BarChart[powerRank, ChartLabels → Table["Page " <> ToString[i], {i, 1, n}],
    PlotLabel → "PageRank Values (via Power Iteration)",
    AxesLabel → {"Page", "PageRank"}, PlotTheme → "Detailed"];

Grid[{{pageRankGraph}, {pageRankBarChart}}, Alignment → {{Left, Right}, {Top, Bottom}}]

Random Web Graph (Adjacency Matrix):
```

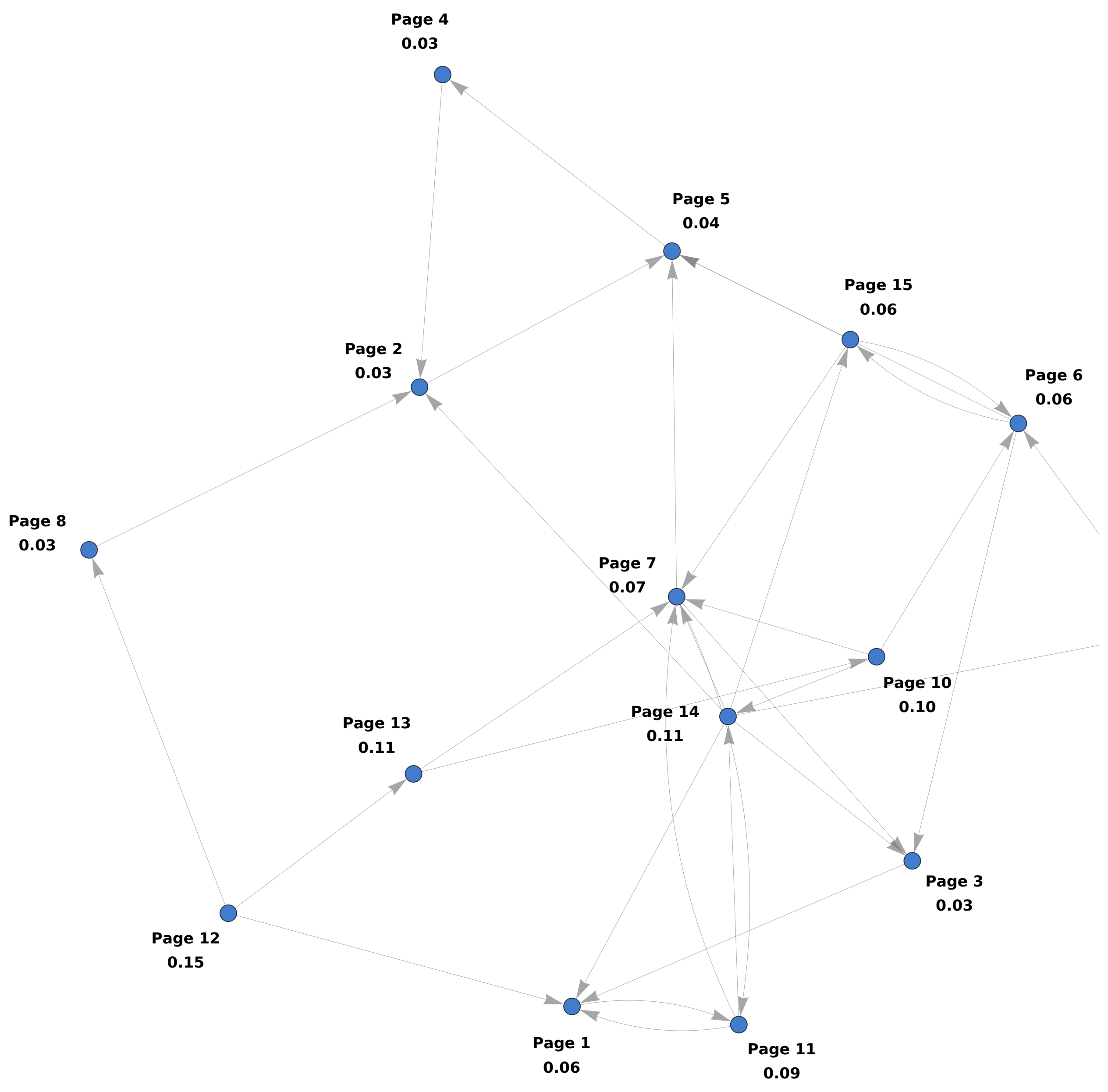
Out[ ]//MatrixForm=
```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0 0 0 0 1
0 0 1 0 1 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0
1 1 1 0 0 0 1 0 1 0 0 0 0 0 1
0 0 0 0 1 1 1 0 0 0 0 0 0 0 0
```

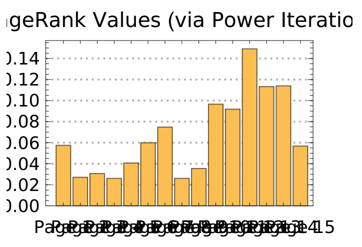
```
PageRank transition matrix:
```

Out[ ]//MatrixForm=
```
  0.01    0.01      0.01    0.01  0.01     0.01    0.01 0.01 0.01 0.01 0.435 0.066666
  0.01    0.01      0.01    0.01 0.2225    0.01    0.01 0.01 0.01 0.01  0.01 0.066666
0.2225    0.01      0.01    0.01  0.01     0.01    0.01 0.01 0.01 0.01  0.01 0.066666
  0.01  0.293333    0.01    0.01  0.01     0.01    0.01 0.01 0.01 0.01  0.01 0.066666
  0.01    0.01      0.01    0.86  0.01     0.01    0.01 0.01 0.01 0.01  0.01 0.066666
  0.01    0.01    0.293333  0.01 0.2225    0.01    0.01 0.01 0.01 0.01  0.01 0.066666
  0.01    0.01    0.293333  0.01 0.2225    0.01    0.01 0.01 0.01 0.01 0.435 0.066666
  0.01  0.293333    0.01    0.01  0.01     0.01    0.01 0.01 0.01 0.01  0.01 0.066666
  0.01    0.01      0.01    0.01  0.01   0.293333  0.01 0.01 0.01 0.01  0.01 0.066666
  0.01    0.01      0.01    0.01  0.01   0.293333  0.18 0.01 0.01 0.01  0.01 0.066666
0.2225    0.01      0.01    0.01  0.01     0.01    0.18 0.01 0.01 0.01  0.01 0.066666
0.2225    0.01      0.01    0.01  0.01     0.01    0.01 0.86 0.01 0.01  0.01 0.066666
  0.01    0.01      0.01    0.01  0.01     0.01    0.18 0.01 0.01 0.86  0.01 0.066666
0.2225  0.293333  0.293333  0.01  0.01     0.01    0.18 0.01 0.86 0.01  0.01 0.066666
  0.01    0.01      0.01    0.01 0.2225  0.293333  0.18 0.01 0.01 0.01  0.01 0.066666
```

```
PageRank values:
{0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667,
  0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667, 0.0666667}

PageRank via power iteration:
{0.0574836, 0.0270935, 0.030668, 0.0261292, 0.0406626, 0.0599179, 0.0748136,
  0.0261292, 0.0354295, 0.0965988, 0.0918373, 0.149166, 0.11328, 0.114002, 0.0567886}
```

Out[•]=





Norms, Least Squares and

Approximation Methods

Matrix Norms

Matrix norms are essential tools in numerical linear algebra that allow us to quantify the “size” of matrices. They are particularly important for analyzing errors, convergence of iterative methods, and stability of numerical algorithms.

A matrix norm $|\cdot|$ is a function that maps a matrix to a non-negative real number satisfying the following properties:

$|A| \geq 0$ for all matrices A, and $|A| = 0$ if and only if $A = 0$
$|\alpha A| = |\alpha| \cdot |A|$ for all scalars α and matrices A
$|A + B| \leq |A| + |B|$ for all matrices A and B of the same size

Let’s implement and visualize different matrix norms in Mathematica:

```
In[•]:= (*Define a sample matrix*)
A = {{3, 1, 4}, {1, 5, 9}, {2, 6, 5}};
MatrixForm[A]

(*Frobenius Norm*)
frobeniusNorm[mat_] := Sqrt[Total[Abs[mat]^2, 2]]
frobeniusNormA = frobeniusNorm[A]
Print["Frobenius norm: ", frobeniusNormA]

(*Using built-in functions*)
Print["Built-in Frobenius norm: ", Norm[A, "Frobenius"]]

(*1-Norm (maximum column sum)*)
oneNorm[mat_] := Max[Total[Abs[mat], {1}]]
Print["1-Norm: ", oneNorm[A]]
Print["Built-in 1-Norm: ", Norm[A, 1]]

(*Infinity Norm (maximum row sum)*)
infinityNorm[mat_] := Max[Total[Abs[mat], {2}]]
Print["Infinity Norm: ", infinityNorm[A]]
Print["Built-in Infinity Norm: ", Norm[A, Infinity]]

(*2-Norm (maximum singular value)*)
twoNorm[mat_] := Max[SingularValueList[N[mat]]]
Print["2-Norm: ", twoNorm[A]]
Print["Built-in 2-Norm: ", Norm[A, 2]]
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix}$$

Out[•]= $3\sqrt{22}$

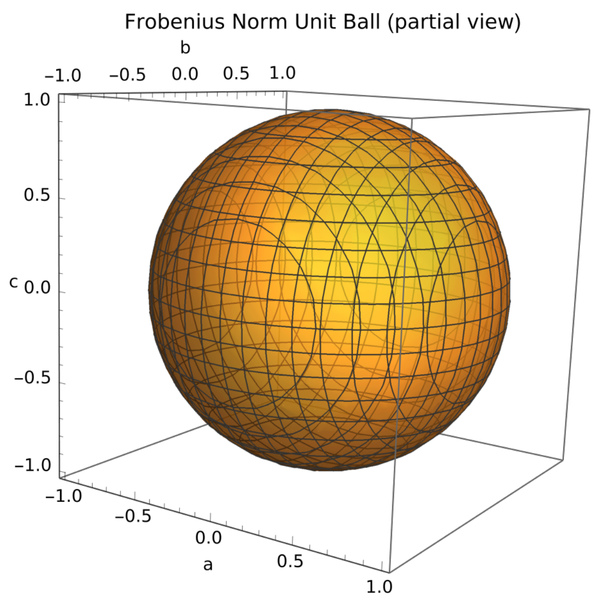
```
Frobenius norm: 3 √22

Built-in Frobenius norm: 3 √22

1-Norm: 18

Built-in 1-Norm: 18

Infinity Norm: 15

Built-in Infinity Norm: 15

2-Norm: 13.5824

Built-in 2-Norm: √(184.…)
```

Visualizing Matrix Norms

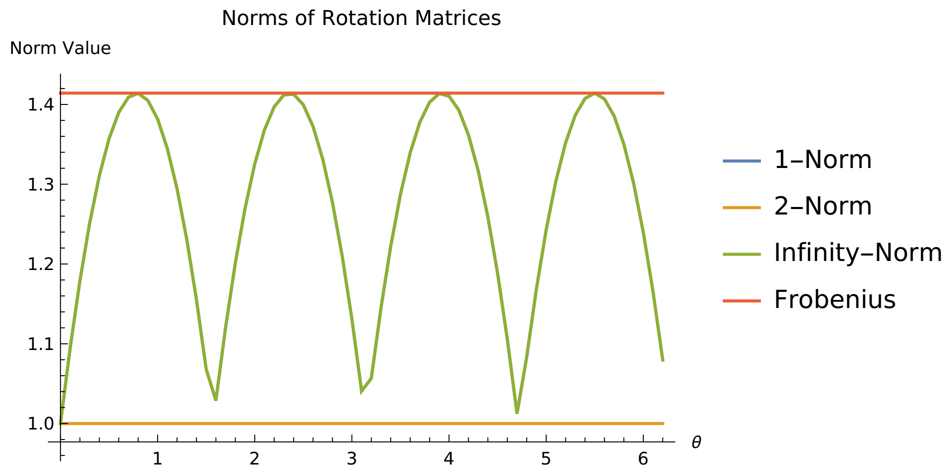
We can visualize how different norms measure matrices by examining the unit ball in each norm:

```
In[∘]:= (*Visualize unit balls for different matrix norms in 2x2 matrices*)
(*We'll parameterize 2x2 matrices using 4 parameters*)
unitBallFrobenius = RegionPlot3D[a^2 + b^2 + c^2 ≤ 1, {a, -1, 1},
  {b, -1, 1}, {c, -1, 1}, PlotLabel → "Frobenius Norm Unit Ball (partial view)",
  AxesLabel → {"a", "b", "c"}, PlotStyle → Opacity[0.7]]

(*Creating a function to compute norm values at different points*)
normComparisonPlot =
  Table[With[{mat = {{Cos[θ], Sin[θ]}, {-Sin[θ], Cos[θ]}}}, {θ, Norm[mat, 1], Norm[mat, 2],
     Norm[mat, Infinity], Norm[mat, "Frobenius"]}], {θ, 0, 2 π, 0.1}];

ListLinePlot[
 Transpose[{normComparisonPlot[[All, 1]], #}] & /@ Transpose[normComparisonPlot[[All, 2 ;;]]],
 PlotLegends → {"1-Norm", "2-Norm", "Infinity-Norm", "Frobenius"},
 AxesLabel → {"θ", "Norm Value"}, PlotLabel → "Norms of Rotation Matrices"]
```

Out[ ]=



Out[ ]=



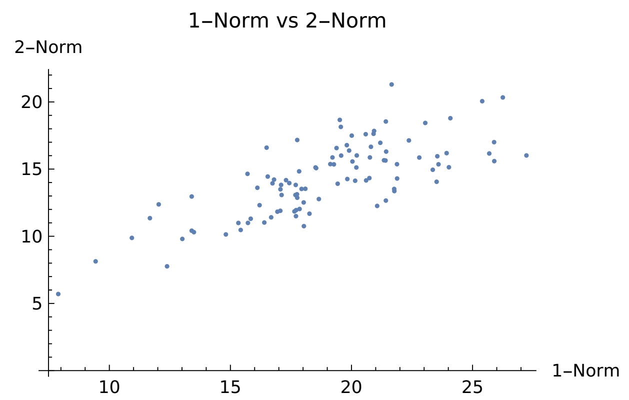
Equivalence of Matrix Norms

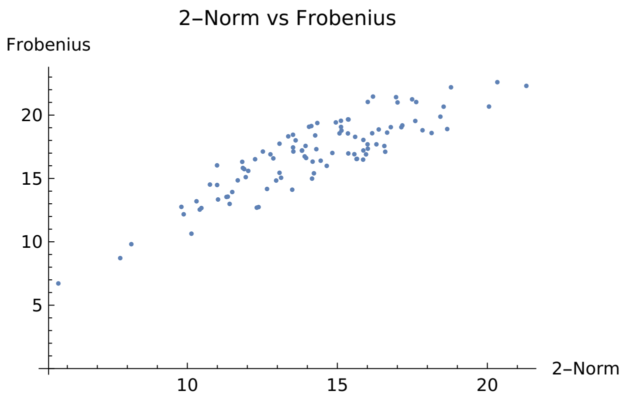
An important property of matrix norms is their equivalence . For any two matrix norms $| \cdot |_\alpha$ and $| \cdot |_\beta$, there exist positive constants c_1 and c_2 such that :

$c_1 \| A \|_\alpha \leq \| A \|_\beta \leq c_2 \| A \|_\alpha$

In[◦]:=
```
(*Generate random matrices and compare norms*)
normEquivalenceData = Table[With[{mat = RandomReal[{-10, 10}, {3, 3}]},
    {Norm[mat, 1], Norm[mat, 2], Norm[mat, Infinity], Norm[mat, "Frobenius"]}], {i, 1, 100}];
(*Plot relationships between different norms*)
GraphicsGrid[{{ListPlot[normEquivalenceData[[All, {1, 2}]],
    PlotLabel → "1-Norm vs 2-Norm", AxesLabel → {"1-Norm", "2-Norm"}]},
  {ListPlot[normEquivalenceData[[All, {2, 4}]], PlotLabel → "2-Norm vs Frobenius",
    AxesLabel → {"2-Norm", "Frobenius"}]}}]

(*Calculate bounds between norms*)
FindBounds[data_, normX_, normY_] :=
 Module[{ratios}, ratios = data[[All, normY]] / data[[All, normX]];
  {Min[ratios], Max[ratios]}]

Print["Bounds between 1-Norm and 2-Norm: ", FindBounds[normEquivalenceData, 1, 2]]
Print["Bounds between 2-Norm and Frobenius: ", FindBounds[normEquivalenceData, 2, 4]]
```

Out[◦]=





```
Bounds between 1-Norm and 2-Norm: {0.582063, 1.02794}

Bounds between 2-Norm and Frobenius: {1.01262, 1.45877}
```

The Least Square Problem

Formulation of the Least Square Problem

The least squares problem is fundamental in data analysis and numerical computing. Given a matrix $A \in \mathrm{R}^{m \times n}$ and a vector $b \in \mathrm{R}^{m}$, we seek a vector $x \in \mathrm{R}^{n}$ that minimizes:
$\| A x - b \|_2^2$,

This minimization problem arises when we have an overdetermined system (more equations than unknowns) and no exact solution exists. Instead, we seek the solution that minimizes the sum of squared errors.

The solution to the least squares problem can be derived analytically:
$x = \left(A^T A\right)^{-1} A^T b$

This formula gives the solution when $A^T A$ is invertible, which occurs when A has full column tank

In[∘]:=
```
(*Define a basic least squares solver*)
leastSquaresSolve[A_, b_] := Module[{ATA, ATb}, ATA = Transpose[A].A;
  ATb = Transpose[A].b;
  LinearSolve[ATA, ATb]]

(*Create an example overdetermined system*)
A = {{1, 1}, {2, 1}, {3, 1}, {4, 1}};
b = {6, 5, 7, 10};

(*Solve using our function*)
x = leastSquaresSolve[A, b]
Print["Our solution: ", x]

(*Using built-in function*)
builtInSolution = LeastSquares[A, b]
Print["Built-in solution: ", builtInSolution]

(*Compute the residual*)
residual = A.x - b
residualNorm = Norm[residual, 2]
Print["Residual norm: ", residualNorm]
```

Out[∘]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Our solution: $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Out[∘]= $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Built-in solution: $\left\{\frac{7}{5}, \frac{7}{2}\right\}$

Out[∘]= $\left\{-\frac{11}{10}, \frac{13}{10}, \frac{7}{10}, -\frac{9}{10}\right\}$

Out[∘]= $\sqrt{\frac{21}{5}}$

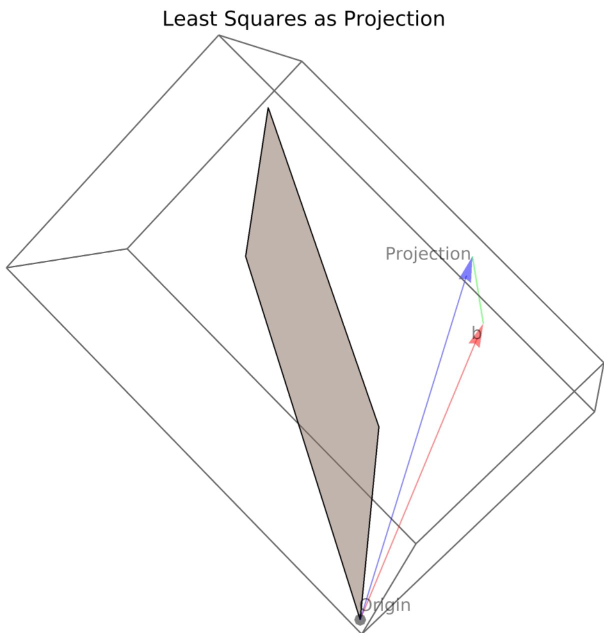
Residual norm: $\sqrt{\frac{21}{5}}$

Geometrical Interpretation

Geometrically, the least squares solution is the projection of the vector b onto the column space of A. This means we're finding the vector in the column space of A that's closest to b.

In[∘]:=
```
(*Visualize the least squares solution for a simple 2D case*)
A = {{1, 1}, {2, 1}, {3, 1}};
b = {2, 3, 1};

(*Column space of A is a plane in 3D space*)
colA1 = A[[All, 1]];
colA2 = A[[All, 2]];

(*Calculate the least squares solution*)
x = LeastSquares[A, b];
projection = A.x;

(*Visualize*)
lsPlot = Graphics3D[{Gray, Opacity[0.5], Polygon[{{0, 0, 0}, colA1, colA1 + colA2, colA2}],
    (*Column space*)
    Red, Arrow[{{0, 0, 0}, b}],
    (*Original vector b*)
    Blue, Arrow[{{0, 0, 0}, projection}],
    (*Projection*)
    Green, Line[{b, projection}],
    (*Residual vector*)
    Black, PointSize[0.02], Point[{0, 0, 0}], Text["Origin", {0, 0, 0}, {-1, -1}],
    Text["b", b, {1, 1}], Text["Projection", projection, {1, 0}]}, PlotRange → All,
   Boxed → True, AxesLabel → {"x", "y", "z"}, PlotLabel → "Least Squares as Projection"];

Show[lsPlot]
```

Out[∘]=



QR Decomposition for Least Squares

As talked about QR decomposition before here’s a simple implementation to check it’s relation to Least Squares

```
In[∘]:= (*Solve least squares using QR decomposition*)
leastSquaresQR[A_, b_] := Module[{qr, Q, R, sol}, {Q, R} = QRDecomposition[A];
  sol = LinearSolve[R, Q.b]; sol]

(*Compare with previous methods*)
A = {{1, 1}, {2, 1}, {3, 1}, {4, 1}};
b = {6, 5, 7, 10};

xQR = leastSquaresQR[A, b]
Print["QR solution: ", xQR]
Print["Normal equation solution: ", leastSquaresSolve[A, b]]
Print["Built-in solution: ", LeastSquares[A, b]]

(*Demonstrate numerical stability*)
(*Create an ill-conditioned matrix*)
illConditioned = {{1, 1}, {1, 1.000001}, {1, 0.999999}};
bIll = {1, 1, 1};

Print["Condition number: ", LinearAlgebra`MatrixConditionNumber[illConditioned]]
Print["QR solution: ", leastSquaresQR[illConditioned, bIll]]
Print["Normal equation solution: ", leastSquaresSolve[illConditioned, bIll]]
```

Singular Value Decomposition for Least Squares

```
In[∘]:= leastSquaresSVD[A_, b_] := Module[{svd, U, S, V, Sinv, solution},
  {U, S, V} = SingularValueDecomposition[A];
  (*Create pseudoinverse of S*)
  Sinv = Map[1/# &, svals]; (*Invert nonzero values*);
  solution = V.Sinv.Transpose[U].b;
  solution]
(*Solve our previous example*)
xSVD = leastSquaresSVD[A, b]
Print["SVD solution: ", xSVD]

(*For ill-conditioned matrix*)
svdSolution = leastSquaresSVD[illConditioned, bIll]
Print["SVD solution for ill-conditioned problem: ", svdSolution]
```

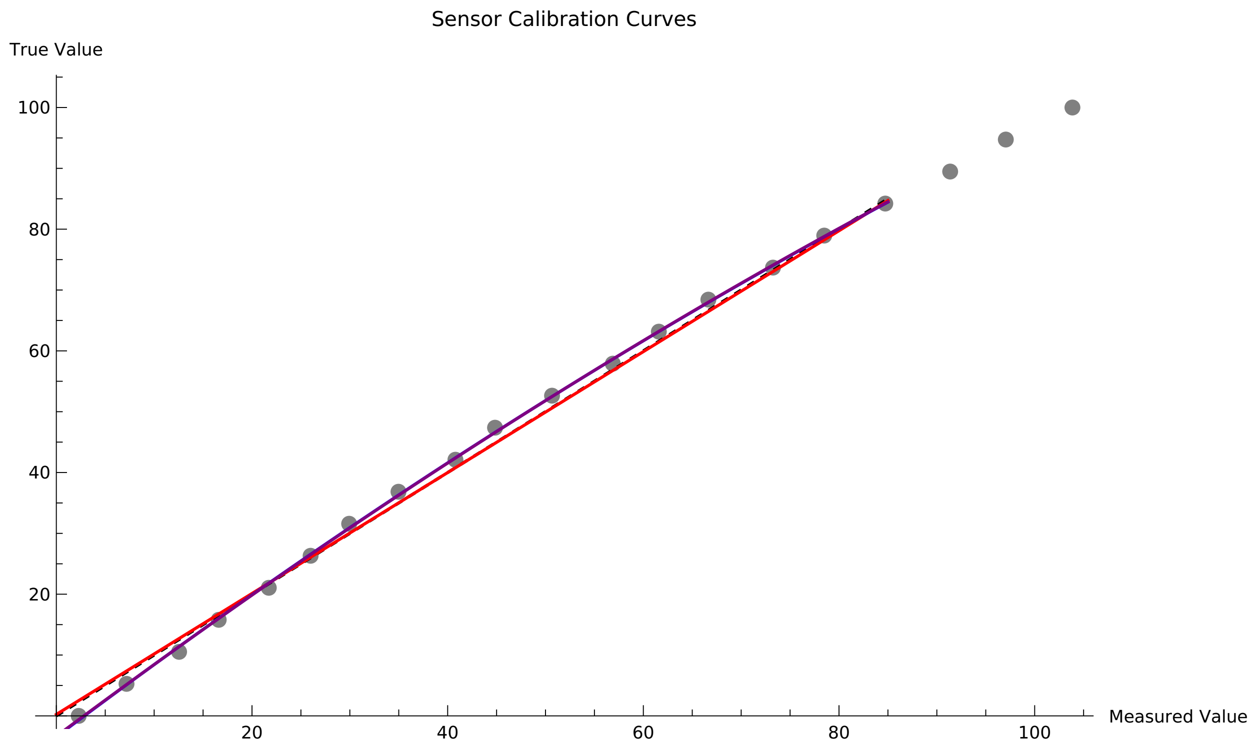
Applications in Data Fitting and Signal Processing

Calibration in Scientific Measurement

Least squares methods are widely used in scientific measurement and instrument calibration. Let’s implement a calibration procedure for a hypothetical sensor:

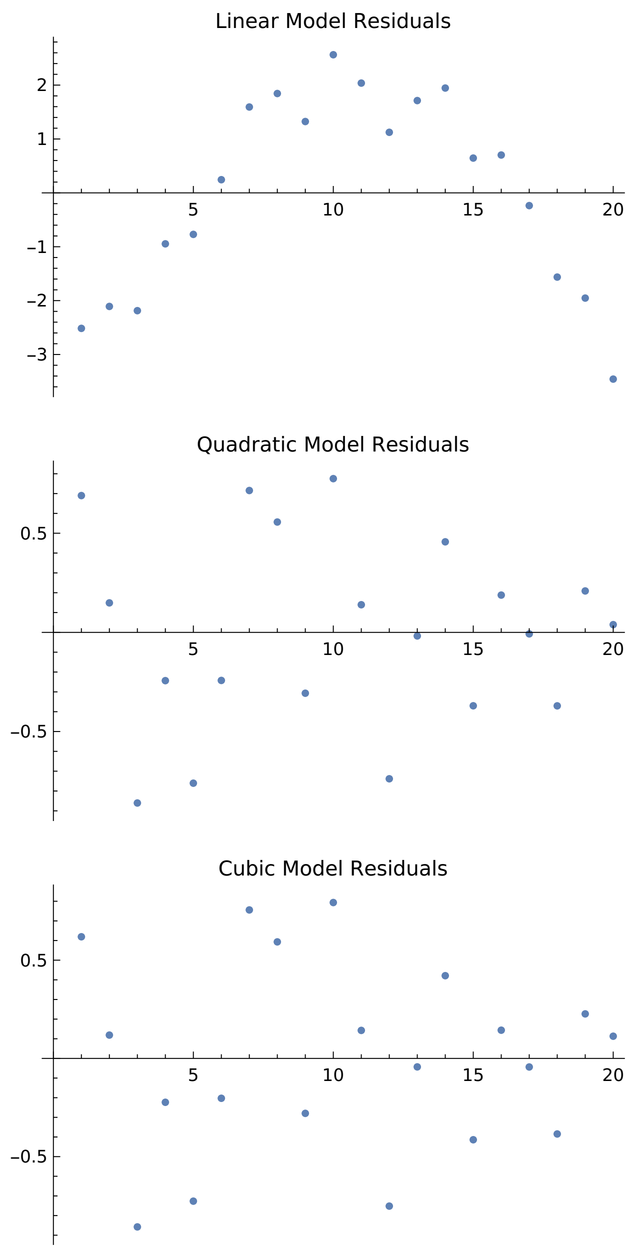
```
In[]:= (*Generate calibration data with nonlinearity for n points*)
generateCalibrationData[n_Integer, range_ : {0, 100}] :=
 Module[{trueValues, measuredValues, min, max}, {min, max} = range;
  (*Generate n evenly spaced true values*)trueValues = Subdivide[min, max, n - 1];
  (*Simulate a nonlinear sensor with noise*)
  measuredValues = 0.8 * trueValues + 0.002 * trueValues ^ 2 + 3 + RandomReal[{-1, 1}, n];
  Transpose[{measuredValues, trueValues}]]

(*Example usage with 30 data points*)
calibrationData = generateCalibrationData[20];

(*Fit different calibration models*)
(*Linear model*)
linearModel = LinearModelFit[calibrationData, x, x];

(*Quadratic model*)
quadraticModel = LinearModelFit[calibrationData, {1, x, x ^ 2}, x];
(*Cubic model*)
cubicModel = LinearModelFit[calibrationData, {1, x, x ^ 2, x ^ 3}, x];

(*Plot calibration curves*)
Show[ListPlot[calibrationData, PlotStyle → {PointSize[0.015], Gray}],
 Plot[{x, linearModel[x], quadraticModel[x], cubicModel[x]},
  {x, 0, 85}, PlotStyle → {{Black, Dashed}, Red, Blue, Purple},
  PlotLegends → {"Ideal", "Linear Fit", "Quadratic Fit", "Cubic Fit"}],
 PlotLabel → "Sensor Calibration Curves", AxesLabel → {"Measured Value", "True Value"}]

(*Evaluate the calibration models*)
Print["Linear model: ", linearModel["BestFitParameters"]]
Print["Linear model R²: ", linearModel["RSquared"]]
Print["Quadratic model: ", quadraticModel["BestFitParameters"]]
Print["Quadratic model R²: ", quadraticModel["RSquared"]]
Print["Cubic model: ", cubicModel["BestFitParameters"]]
Print["Cubic model R²: ", cubicModel["RSquared"]]

(*Show residuals plot*)
residualsPlot =
 GraphicsGrid[{{ListPlot[linearModel["FitResiduals"], PlotLabel → "Linear Model Residuals"]},
   {ListPlot[quadraticModel["FitResiduals"], PlotLabel → "Quadratic Model Residuals"]},
   {ListPlot[cubicModel["FitResiduals"], PlotLabel → "Cubic Model Residuals"]}}]
```

Out[•]=



```
Linear model: {0.241213, 0.993807}

Linear model R²: 0.996598

Quadratic model: {-3.44141, 1.20715, -0.00203684}

Quadratic model R²: 0.999751

Cubic model: {-3.34771, 1.19654, -0.00178423, -1.59859 × 10^-6}

Cubic model R²: 0.999753
```

Out[ ]=



Further Contents

- Andreescu, Titu, D. Andrica, and D. Andrica. Complex Numbers from A To--Z. Boston: Birkhäuser, 2006.
- Axler, Sheldon. Linear Algebra Done Right. Undergraduate Texts in Mathematics. Cham: Springer International Publishing, 2024. https://doi.org/10.1007/978-3-031-41026-0.

- Jackson, John David. Mathematics for Quantum Mechanics: An Introductory Survey of Operators, Eigenvalues, and Linear Vector Spaces. Dover Books on Mathematics. Mineola, N.Y: Dover Publications, 2006.
- Lang, Serge. Linear Algebra. Third Edition. Undergraduate Texts in Mathematics. New York, NY: Springer, 1987. https://doi.org/10.1007/978-1-4757-1949-9.