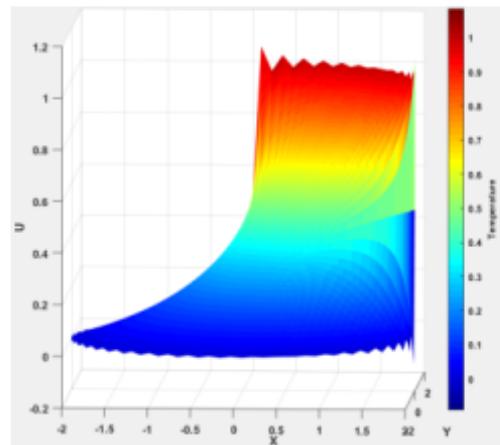


Mathematical Methods

for Engineers



Stu Blair
Mighty Goat Press

WHEN IN DOUBT, MULTIPLY BOTH SIDES BY AN ORTHOGONAL FUNCTION
AND INTEGRATE.

P.L. CHEBYSHEV

THE PURPOSE OF COMPUTING IS INSIGHT, NOT PICTURES.

L.N. TREFETHEN

NEVER DO A CALCULATION UNTIL YOU ALREADY KNOW THE ANSWER.

J.A. WHEELER

UNITED STATES NAVAL ACADEMY

MATHEMATICAL METHODS FOR ENGINEERS

MIGHTY GOAT PRESS

Copyright © 2024 United States Naval Academy

PUBLISHED BY MIGHTY GOAT PRESS

First printing, June 2024

Contents

I Introduction and Review

27

Lecture 1 - Introduction, Definitions and Terminology

29

Lecture 2 - Separable and Linear 1st order Equations

33

Assignment #1

39

Lecture 3 - Theory of Linear Equations

41

Lecture 4 - Homogeneous Linear Equations with Constant Coefficients

47

Lecture 5 - Non-homogeneous Linear Equations with Constant Coefficients

53

Assignment #2

59

Lecture 6 - Cauchy-Euler Equations

63

II Power Series Methods

69

Lecture 7 - Review of Power Series 71

Lecture 8 - Power Series Solutions at Ordinary Points 77

Assignment #3 83

Lecture 9 - Power Series Solutions with MATLAB 85

Lecture 10 - Legendre's Equation 91

Lecture 11 - Solutions about Singular Points 95

Assignment #4 101

Lecture 12 - Bessel's Equation and Bessel Functions 103

Lecture 13 - Solving ODEs Reducible to Bessel's Equation 109

Lecture 14 - Modified Bessel Function and Parametric Modified Bessel Function
113

Assignment #5 117

Review Problems #1 119

III Orthogonal Functions and Fourier Series 121

Lecture 15 - Introduction to Orthogonal Functions 123

Lecture 16 - Fourier Series 127

Lecture 17 - Generating and Plotting Fourier Series in MATLAB 133

Assignment #6 139

Lecture 18 - Sturm-Liouville Problems 141

Lecture 19 - Fourier-Bessel Series Expansions 147

Lecture 20 - Fourier-Legendre Series Expansion 155

Assignment #7 159

IV Boundary Value Problems in Rectangular Coordinates 161

Lecture 21 - Introduction to Separable Partial Differential Equations 163

Lecture 22 - Classical PDEs and BVPs 167

Assignment #8 173

Lecture 23 - The Heat Equation 175

Lecture 24 - The Wave Equation 183

Lecture 25 - Heat and Wave Equation with MATLAB 189

Assignment #9 197

Review Problems #2 199

Lecture 26 - Laplace's Equation 201

Lecture 27 - Non-homogeneous Problems 209

Lecture 28 - Orthogonal Series Expansion 213

Assignment #10 221

Lecture 29 - Fourier Series in Two Variables 223

V Boundary Value Problems in Polar, Cylindrical and Spherical Coordinates
229

Lecture 30 - Laplacian in Polar Coordinates 231

Lecture 31 - Polar Coordinates with Angular Symmetry 239

Assignment #11 247

Lecture 32 - Laplace Equation in Cylindrical Coordinates 249

Lecture 33 - Introduction to the Neutron Diffusion Equation in Cylindrical Coordinates 257

Assignment #12 263

Lecture 34 - Laplace's Equation in Spherical Coordinates 265

<i>Lecture 35 - Non-homogeneous Problem in Spherical Coordinates</i>	271
<i>Assignment #13</i>	277
<i>Review Problems #3</i>	279
<i>VI Numerical Methods Introduction</i>	281
<i>Lecture 1 - Course Introduction and Numeric Preliminaries</i>	283
<i>Lecture 2 - Linear Algebra Background</i>	291
<i>VII Solving Non-Linear Equations</i>	297
<i>Lecture 3 - The Bisection Method</i>	299
<i>Assignment #1</i>	305
<i>Lecture 4 - Newton's Method and Secant Method</i>	307
<i>Lecture 5 - MATLAB Functions for Root Finding</i>	315
<i>Assignment #2</i>	319
<i>Lecture 6 - Newton's Method for Systems of Non-Linear Equations</i>	321
<i>VIII Solving Linear Equations</i>	327
<i>Lecture 7 - Introduction to Linear Algebra and Gauss Elimination</i>	329

Lecture 8 - Gauss Elimination with Pivoting 337

Assignment #3 343

Lecture 9 - LU Factorization 345

Lecture 10 - MATLAB Built-in Methods and Error Estimates 351

Lecture 11 Sparse Matrices and Iterative Methods 355

Lecture 12 Preconditioning and MATLAB Built-in Methods 365

Assignment #4 371

Review #1 375

IX Curve Fitting and Interpolation 379

Lecture 13 Least Squares Curve Fitting 381

Lecture 14 Curve Fitting with Non-linear Functions 391

Assignment #5 395

Lecture 15 Interpolation with Lagrange Polynomials 397

Lecture 16 - Curve Fitting and Interpolation with Built-in MATLAB Tools
403

X *Numeric Differentiation and Integration* 407

Lecture 17 - Numeric Differentiation with Finite Difference Formulas 409

Lecture 18 - Numeric Differentiation with Lagrange Polynomials 417

Assignment #6 421

Lecture 19 - Numeric Integration with Newton-Cotes Formulas 423

Lecture 20 - Gauss Quadrature 429

Lecture 21 - Monte Carlo Methods for Numeric Integration 435

Lecture 22 - Numerical Quadrature with MATLAB Built-in Functions 441

Assignment #7 445

XI *Solving Initial Value Problems* 449

Lecture 23 - ODE Review and Euler's Method for IVPs 451

Lecture 24 - Solving Systems of 1st-Order IVPs 459

Lecture 25 - Solving IVPs with Runge-Kutta Methods 465

Assignment #8 475

Lecture 26 - High Order & Implicit Runge-Kutta Methods 479

Lecture 27 - Embedded RK and MATLAB Built-in RK Methods 483

XII *Solving Boundary Value Problems* 489

Lecture 28 - Solving Boundary value Problems Using the Shooting Method
491

Lecture 29 - Solving Boundary Value Problems, MATLAB Built-in Methods
499

Assignment #9 505

Lecture 30 - More Boundary Value Problem Examples 509

Lecture 31 - Solving BVPs with the Finite Difference Method 513

Lecture 32 - Solving Non-linear BVPs with Finite Difference Methods 521

Lecture 33 - The Finite Element Method, Galerkin Method and Weak Form
525

Lecture 34 - Finite Element Method, Galerkin Method FEM in One Dimension
531

XIII *Back Matter* 545

Bibliography 547

Appendices

Matlab Style Rules 553

Index 555

List of Figures

- 1 Plot of $\cosh x$ and $\sinh x$ 48
- 2 Power series solution to $u'' - (1 + x)u = 0$, $u(0) = 5$, $u'(0) = 1$. 89
- 3 Power series solution with different values of n . 89
- 4 Convergence of the power series solution to the numeric solution. 89
- 5 Legendre Polynomials of order 0 through 5. 94
- 6 An example even function. 130
- 7 An example odd function. 130
- 8 Example #1 $f(x)$. 133
- 9 Fourier series expansion with $N=5$. 135
- 10 Fourier series expansion with $N=15$. 135
- 11 Fourier series expansion with $N=150$. 135
- 12 Example #2 $f(x)$. 136
- 13 Fourier series expansion with $N=5$. 136
- 14 Fourier series expansion with $N=15$. 136
- 15 Fourier series expansion with $N=150$. 136
- 16 Even-, odd- and identity-reflection for $f(x) = x^2$. 137
- 17 Plot of $\sin \alpha + \alpha \cos \alpha$. 143
- 18 Bessel functions of order 1. 148
- 19 Fourier-Bessel expansion of $f(x) = x$. 152
- 20 Fourier-Bessel expansion of $f(x) = x$. 152
- 21 Convergence of the Fourier-Bessel expansion of $f(x) = x$. 152
- 22 Fourier-Legendre expansion with $N = 15$. 157
- 23 Convergence of Fourier-Legendre expansion 157
- 24 Smooth initial condition. 179
- 25 Solution for smooth initial condition. 179
- 26 Solution with high thermal diffusivity. 180

27	Example with discontinuous initial condition.	180
28	Solution with discontinuous initial condition, $N = 25$.	180
29	Solution with discontinuous initial condition, $N = 100$.	180
30	Solution with an insulated boundary at $x = L$.	181
31	Wave equation solution from $t = 0$ to $t = 3$.	188
32	Plot of heat equation example at $t=0, 1$, and 10 seconds	191
33	Surface plot of heat equation example.	193
34	Plot of wave equation example problem at $t=0, 1.0, 2.0$, and 3.0 sec.	195
35	Schematic of example Laplace's equation problem.	201
36	Pairs of boundary conditions for Laplace's equation.	202
37	Surface plot of solution to example problem.	206
38	Neither spatial dimension has all homogeneous boundary conditions.	206
39	Superposition of two BVPs each with homogeneous boundary conditions in one dimension.	207
40	The solution, $u(x, t)$ at various times and the steady-state solution $\psi(x)$.	212
41	Schematic of Example #1.	213
42	Plot of $\tanh(\nu)$.	214
43	Plot of $\tan \nu$ and $-\nu/h$ vs ν for $h = 1$.	215
44	Solution for Example #1.	217
45	Variations in α and h for example problem.	217
46	Schematic of Example #2.	218
47	Plots of example functions and their double Fourier series expansions.	225
48	Cylindrical coordinate system.	231
49	Solution for the case where $f(x) = ex_1(x)$.	237
50	Plots solution with boundary conditions selected for example 2.	245
51	Schematic of Case I	249
52	Plots of $I_0(\alpha r)$ and $K_0(\alpha r)$ for $\alpha r > 0$.	251
53	Cross section of Case I solution for $N=25$.	253
54	Cross section of Case I solution for $N=100$.	253
55	Schematic of domain and boundary conditions for Case I.	253
56	Solution for Case II with $N=150$.	255
57	Spherical coordinate system.	265
58	Plot of solution using ParaView.	270
59	Plots solution at various times.	276

60	The number 23 in unsigned integer format.	285
61	Demonstration of twos-complement: adding 23 and -23 in 5-bit representation.	286
62	IEEE-754 double precision number format.	287
63	Exponent: $3+1023 = 1026$ in 11-bit binary.	288
64	Mantissa: $0.25 + 0.0625 = 0.3125$.	288
65	Schematic of example.	299
66	Solution of $f(X) = 0$ must lie in $[a, b]$ if $f(a)f(b) < 0$.	300
67	First three iterations of the bisection method.	300
68	The first four iterations of Newton's method.	308
69	Newton's method convergence issue due to a local minimum.	310
70	Newton's method failure to converge due to inflection points in vicinity of solution.	310
71	Schematic of the secant method.	311
72	Secant method in action.	311
73	The first two iterations of the regula falsi method.	315
74	Example system of non-linear equations.	323
75	Two-dimensional truss structure.	329
76	Computing time vs. n for Gauss elimination.	347
77	Logic tree for <code>mldivide()</code> , part 1.	351
78	Logic tree for <code>mldivide()</code> , part 2.	352
79	A triangular mesh for a finite element method analysis of the transient heat equation.	356
80	Non-zeros in linear system for transient heat conduction.	356
81	Example sparse matrix in Compressed Sparse Column format.	356
82	Sparsity pattern of L and U matrices from decomposition of A.	357
83	Jacobi method performance comparison for 900×900 test matrix.	359
84	Sparsity pattern for <code>BCSSTK15</code> .	367
85	Output for preconditioning with <i>complete</i> LU factorization.	367
86	Output for Jacobi iteration with incomplete LU preconditioning.	368
87	Experimental data from wind tunnel testing. The y -axis is the ratio of the tangential velocity of a vortex to the free stream flow velocity ($y = V_\theta/V_\infty$). The x -axis is the ratio of the distance from the vortex core to the chord of an aircraft wing. ($x = R/C$).	381
88	<i>Guessed-fit</i> linear estimation of the experimental data. $M = -0.033$ is the measured slope of the estimated line and $b = 0.1$ is the y -intercept.	382

- 89 Best fit linear estimation of the experimental data. $M = -0.0288$ is the measured slope of the estimated line and $b = 0.0940$ is the y -intercept. 385
- 90 Best fit linear estimation of the experimental data using 2nd order, 3rd order and 6th order estimators. 386
- 91 Best fit curve with theoretical model parameters. The linear Least Squares estimator is shown for reference. 387
- 92 Linear fit, $m = 0.416$, $b = 12.913$. 388
- 93 Second-order polynomial fit. $a = 0.436$, $b = 0.0979$, and $c = -5.89e - 5$. 389
- 94 Model fit of data. $a = -6.20$, $b = 1.80$. 389
- 95 Plot of least squares estimator for Example #1. 392
- 96 Plot of least squares estimator for Example #2. 393
- 97 Plot of least squares estimator for Example #3. 394
- 98 Least squares interpolation of data points. 398
- 99 Warning issued by MATLAB due to the high condition number of $(X^T X)$. 398
- 100 Lagrange interpolation with $n = 5$, $n = 10$, $n = 15$, and $n = 20$ uniformly spaced points. 400
- 101 Lagrange interpolation with $n = 5$, $n = 10$, $n = 15$, and $n = 20$ points placed at Chebyshev nodes. 400
- 102 Monomial interpolant for stress-strain data. 402
- 103 Lagrange interpolation for stress-strain data. 402
- 104 A 3rd-order polynomial fit using `polyval()` and `polyfit()`. 404
- 105 Interpolated curve of engine power versus engine speed. 406
- 106 A discretized domain for a finite element method. 406
- 107 Interpolated temperature data. 406
- 108 A discrete grid on which a function may be defined. 409
- 109 A plot of $\cos x$ and its first derivative calculated numerically. 411
- 110 Convergence behavior using 2-point finite difference formulas. 412
- 111 Convergence behavior using 2nd-order finite difference equations throughout the domain. 412
- 112 A plot of the first four eigenvectors of A . 416
- 113 The matrix D with the first 4 eigenvalues of A . 416
- 114 Numeric differentiation with $n = 11$ uniformly spaced points. 419
- 115 Numeric differentiation with $n = 21$ uniformly spaced points. 419
- 116 Numeric differentiation with $n = 21$ non-uniformly spaced points. 419
- 117 Numeric differentiation with $n = 51$ non-uniformly spaced points. 419
- 118 Schematic of midpoint rule. 424
- 119 Convergence of the midpoint rule. 424

120 Schematic of the trapezoidal rule with equal sub-interval sizes.	424
121 Convergence behavior of the trapezoidal rule.	425
122 Schematic of Simpson's rule.	425
123 Convergence behavior of Simpson's rule.	426
124 MATLAB output to find middle sample point and all weights.	428
125 Convergence behavior of a 3-point, 6th order quadrature formula.	428
126 Published Gauss points and weights up to $n = 6$.	430
127 Convergence behavior of Gauss quadrature.	430
128 Sample of 1000 uniformly distributed random points from within the box $[-3,3] \times [0,1]$.	436
129 Convergence behavior of the Monte Carlo numeric integration algorithm.	437
130 An example of a function that Newton-Cotes and Gauss quadrature do not integrate well.	438
131 Example water channel dimensions.	441
132 Schematic of wine barrel.	446
133 Approximate solution of example problem using Euler's explicit method with $N = 30$.	455
134 Convergence behavior of Euler's explicit method for example problem.	455
135 Result of attempting to solve a "stiff" differential equation with Euler's explicit method, $N = 95$.	456
136 Solution of a stiff IVP with Euler's implicit method with $N = 10$.	457
137 Quadratic convergence of the modified Euler method.	458
138 Schematic of representative fission product decay process.	459
139 Fission product decay chain for generating xenon-135.	460
140 Xenon-135 concentration during a transient.	462
141 A rocket test facility idealized as a spring-mass system.	462
142 Displacement of the rocket during a simulated powered test.	464
143 Schematic of a Butcher Tableau.	466
144 Comparison between numeric and exact solution for Example #1.	469
145 Convergence behavior of 2 nd -order RK method for Example #1.	469
146 Solution of Example #2.	473
147 Convergence behavior of 2 nd -order RK solver for systems of equations.	473
148 Elliptical shape for breast cancer.	475
149 Water supply tank.	476

150	Solution of example problem with an embedded Runge-Kutta method.	486
151	Pin Fin Boundary Value Problem Schematic.	493
152	Notional pin temperature profile.	494
153	Shooting Method Example Solution.	496
154	Solution of example problem with <code>bvp5c</code> .	502
155	Current-carrying wire schematic.	502
156	Solution for Example #2.	502
157	Schematic of cylindrical pipe.	506
158	A typical nuclear reactor fuel pin.	509
159	Cladding temperature profile.	510
160	Temperature profile with constrained Q .	512
161	Function discretized into a finite number of intervals for the FDM.	513
162	Linear system after applying Dirichlet boundary conditions.	516
163	Finite difference method solution to Example #1 and point-wise error. $N = 200$	519
164	Finite difference method solution to Example #1 and point-wise error. $N = 400$	519
165	Finite difference method solution to Example #2 and point-wise error.	520
166	Pin Fin Boundary Value Problem Schematic.	521
167	Solution of example problem with Finite Difference Methods	524
168	Exact solution of the example BVP.	525
169	Approximate solution using the Galerkin method.	527
170	Discretization of problem domain with piece-wise linear elements.	532
171	Schematic of system assembly.	534
172	Approximate solution with three linear elements.	536
173	FEM solution with 30 elements.	537
174	FEM Solution for 3 linear elements, revisited.	539
175	FEM Solution for 3 elements with 9 th -order shape functions.	539

List of Tables

1	Forms of $u_p(x)$ for given terms in $g(x)$	54
2	The first four Legendre Polynomials	93
3	First few coefficients in solution to Bessel's Equation.	105
4	Comparison for random 50×50 matrix.	341
5	Comparison for random 150×150 matrix.	341
6	Comparison for LNS 131 from the Matrix Market.	341
7	Comparison for LNS 511 from the Matrix Market.	342
8	Common vector and matrix norms.	353
9	Numerical data from wind-tunnel experiment.	382
10	Process data from electrophoretic fiber-making process.	388
11	Transforming nonlinear equations to linear form.	391
12	Table of data for Example #1.	392
13	Table of data for Example #2.	392
14	Table of data for Example #3.	393
15	Percentage of U.S. households that own at least one computer.	403
16	Sample points and weights for a 3-point, 6th order convergent quadrature formula.	428
17	Water speed data taken at various channel depths.	441
18	Butcher Tableau for the modified Euler's method.	467
19	Butcher tableau for classical 3 rd -order, 3-stage explicit RK method.	479
20	Butcher tableau for classical 4 th -order, 4-stage explicit RK method.	479
21	Butcher tableau for a 2-stage, 3 rd -order implicit RK method.	480
22	Butcher tableau for a 2-stage, 4 th -order implicit RK method.	480
23	Butcher tableau for the Bogacki-Shampine embedded Runge-Kutta method.	484

24	A partial list of built-in IVP solvers for MATLAB.	487
25	Example problem parameters.	493
26	Finite difference formulas for y' with $\mathcal{O}(h^2)$ convergence.	514
27	Finite difference formulas for y'' with $\mathcal{O}(h^2)$ convergence.	514
28	Example problem parameters.	522

Preface

The purpose of this text is to provide a concise reference for engineering students who would like to strengthen their conceptual understanding and practical proficiency in analytical and numerical methods in engineering. The material is based on a sequence of two courses taught at the United States Naval Academy.

Analytical Methods

The first course is focused on analytical methods for linear ordinary and partial differential equations. All students come into the course having taken a three-semester sequence of calculus along with a course in ordinary differential equations. The analytical methods portion briefly reviews methods for constant coefficient linear equations and proceeds to methods for non-constant coefficient ODEs including Cauchy-Euler equations, power series methods, and the method of Frobenius. After a review of Fourier Series methods and an introduction to Fourier-Legendre and Fourier-Bessel expansions we thoroughly explore solutions to second-order, linear, partial differential equations. Since many students are also studying nuclear engineering, there is a heavy focus on addressing boundary value problems in cylindrical and spherical coordinate systems that are applicable to other topics of interest such as reactor physics. There is also heavy emphasis on heat transfer applications that students will see later on in their undergraduate curriculum.

The materials presented are based heavily on Professor Dennis Zill's excellent book.¹ We lightly select from chapters 1-3 for review; chapter 5 for series solution methods; and chapters 12-14 for Fourier Series and solutions to linear boundary value problems.

What distinguishes this course from Prof Zill's work is the explicit incorporation of computational tools in the solution process. These "semi-analytical methods" are presented here in MATLAB² owing to the students preparation with that tool. Other open-source tools like Octave³ and Python,⁴ of course, could be used.

¹ Dennis G Zill. *Advanced Engineering Mathematics*. Jones & Bartlett Learning, 2020

² Inc. The Math Works. Matlab, v2022a, 2022. URL <https://www.mathworks.com/>

³ John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*, 2020. URL <https://www.gnu.org/software/octave/doc/v5.2.0/>

⁴ Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697

Numerical Methods

The second course is focused on numerical methods for a variety of applications relevant to the physical sciences. Prerequisites for this course include a three-semester sequence of calculus followed by a course in ordinary differential equations. Students also are required to have taken an introductory programming course where they have gained experience using the MATLAB programming environment. A majority of the students who have taken this course have also taken the Analytical Methods course previously, although it is not a requirement.

The course starts with an introduction to number representation on a computer and reviews the basics of linear algebra. This is followed by treatment of algorithms for solving nonlinear equations and systems of equations. For linear systems of equations, both direct and iterative methods are described; this includes treatment of sparse linear systems and preconditioning for iterative algorithms. Least squares curve fitting is described in some detail as is interpolation with Lagrange polynomials. Numeric differentiation is treated with both finite difference formulas as well as differentiation with Lagrange polynomials. Several algorithms for numeric integration are described along with a fairly complete derivation of Gauss quadrature. Several Runge-Kutta methods are illustrated for solving initial value problems, this includes a description of embedded RK methods with adaptive time-stepping. Boundary value problems are solved using the shooting method, finite difference methods, and finite element methods. A moderately detailed derivation of the Galerkin formulation of the method of weighted residuals is used to introduce Finite element methods. The course is finished with several computational workshops using COMSOL, the details of which are not treated in this manuscript.

The materials presented are based heavily on the excellent textbook by Prof Amos Gilat and Prof Vish Subramaniam.⁵ The review is taken from selected portions of chapter 1 and 2; methods for solving non-linear and linear equations from chapter 3 and 4; curve fitting, interpolation, differentiation and integration are derived, in part, from chapters 6, 8, and 9; and most of the initial and boundary value problem examples are taken from chapters 10 and 11.

Several topics have been added to the treatment of Gilat and Subramaniam in an effort to selectively add depth and, in a few cases, provide updates to highlight tools available with more recent releases of MATLAB. Depth is added with the treatment of sparse matrices and preconditioned iterative solution schemes for linear systems of equations; a more detailed derivation of Gauss quadrature is pro-

⁵ Gilat, Amos and Subramaniam, Vish. *Numerical Methods for Engineers and Scientists: an Introduction with Applications Using MATLAB*. Wiley, Hoboken,NJ, third edition, 2014

vided; several more Runge-Kutta methods are shown for solving initial value problems including implicit and embedded RK methods. The treatment of finite element methods is based, in part, on the exposition of and lessons learned from Prof Young Kwon from the Naval Postgraduate School.⁶

This course is intended to serve scientists and engineers who will use numerical methods *as a tool* in the context of computational problems that they may face. As a consequence, a fairly pragmatic approach is taken with many of the derivation details summarized or, in some cases, skipped altogether. Many methods are described not derived and theorems are used and applied, not proven. I hope that any mathematician with the patience to read through these pages will graciously forgive me and, if needed, look for a different reference.

⁶ Kwon, Young W and Bang, Hyochoong. *The Finite Element Method using MATLAB*. CRC press, 2018

Part I

Introduction and Review

Lecture 1 - Introduction, Definitions and Terminology

Objectives

The objectives of this lecture are:

- Provide an overview of course content.
- Define basic terms related to differential equations.
- Provide examples of classification schemes for differential equations.

Course Introduction

This course is intended as a one-semester introduction to partial differential equations. It is assumed that all students have a thorough background in single- and multi-variable calculus as well as differential equations. The first few lectures comprise a review of the portions of differential equations on which this course most heavily relies. This is followed by a treatment of power series methods and the method of Frobenius. These are needed so that students will understand the origins of Legendre polynomials and Bessel functions that will be used in the solution of boundary value problems in spherical and cylindrical coordinates respectively.

THE MAIN BODY of material deals with the solution of (mostly homogeneous) boundary value problems—wave equation, heat equation, and Laplace equation—in rectangular, polar/cylindrical, and spherical coordinate systems. For this, a preparatory review of Fourier series expansions along with Fourier-Legendre and Fourier-Bessel expansions is included along with a leavening of Sturm-Liouville theory in boundary value problems. The rest is a problem-by-problem tour of methods and analysis with heavy emphasis on heat transfer and nuclear engineering applications.

Classification of Differential Equations

It is important to be able to classify differential equations. In this class we will learn a variety of techniques to find a function that satisfies a differential equation along with its boundary or initial conditions.¹ The techniques we learn in this class are tailored for specific classes of problems; you classify the problem and that tells you what method to use. If you improperly classify the equation, you will likely use an inappropriate method and may have trouble figuring out why you cannot solve the problem.

Classification by Type and Order

We shall start with the easiest classification categories: type and order. There are two *types* of differential equations that we will consider: *ordinary differential equations* and *partial differential equations*.

IN AN ORDINARY differential equation, there is only one independent variable. In a *partial* differential equation, there are multiple independent variables and consequently derivatives of the dependent variable will require partial derivatives.

THE ORDER OF a differential equation is the order of the highest derivative in the equation. This is typically not confusing for students. If anything needs to be added here it is to be mindful of the difference between a higher order derivative and an exponent. For example, in the second order, non-linear, ordinary differential equation shown below,

$$\frac{d^2u}{dx^2} + 5 \left(\frac{du}{dx} \right)^3 - 4u = e^x$$

it isn't *too* hard to realize that the "3" is an exponent and the "2" denotes a second derivative. Still, be mindful.

Classification by Linearity

An n^{th} order ordinary differential equation is said to be *linear* when it can be written in the form shown in Equation 1:

$$a_n(x)u^{(n)} + a_{n-1}(x)u^{(n-1)} + \cdots + a_1(x)' + a_0(x)u = g(x) \quad (1)$$

The key features that you should note in the form of Equation 1 are:

1. The *dependent variable* and *all of its derivatives* are of the first degree; that is, the power of each term involving u is 1.

¹ Consider the differential equation: $\frac{du}{dx} = ux$. The variable u stands for the function, $u(x)$, that satisfies the equation; u is also referred to as the **dependent variable**. The variable x is the **independent variable**. By convention we will use the variables x, y, z and r, θ, ϕ as spatial independent variables and t as an independent variable for time-dependent problems. We will use many other letters to denote dependent variables but most commonly u, f, g, h, v and ψ . When working through the separation of variables process we will also use F, G , and sometimes H .

Example ODE:

$$\frac{d^2u}{dt^2} + t \frac{du}{dt} = 3e^{-t}$$

There is one independent variable, t

Example PDE:

$$\frac{\partial^2u}{\partial x^2} + \frac{\partial^2u}{\partial y^2} = 0$$

There are two independent variables, x , and y .

Note: The notation $u^{(n)}$ and $u^{(n-1)}$ refer to the n^{th} and $(n-1)^{\text{th}}$ derivative of u respectively.

2. The coefficients of each term, $a_n(x)$, depend at most on the *independent* variable(s).

A lot of students struggle with discriminating between linear and nonlinear ODEs but it really is as simple as checking these two features. If both conditions are satisfied: the equation is linear. If not, the equation is nonlinear. As examples, Equation 2 violates the first criterion; Equation 3 violates the second.

$$\frac{d^2u}{dx^2} + u^2 = 0 \quad (2)$$

$$\frac{d^3u}{dx^3} - 5u \frac{du}{dx} = x \quad (3)$$

Verification of an Explicit Solution

A solution in which the dependent variable is expressed solely in terms of the independent variable and constants is said to be an *explicit* solution. Otherwise, the solution is *implicit*.

IN THIS CLASS we will mainly be interested in finding explicit solutions to differential equations that we are given or have derived. There are some cases, however, where we are given a function and we wish to verify that it is a solution to a given differential equation. To do this, we simply “plug” the equation into the differential operator and verify that an identity is derived.

Example: Verify that $u = \frac{6}{5} - \frac{6}{5}e^{-20t}$ is a solution to:

$$\frac{du}{dt} + 20u = 24$$

Solution: Since $\frac{du}{dt} = \frac{d}{dt}\left(\frac{6}{5} - \frac{6}{5}e^{-20t}\right) = 24e^{-20t}$, we can see that:

$$\begin{aligned} \frac{du}{dt} + 20u &= 24e^{-20t} + 20\left(\frac{6}{5} - \frac{6}{5}e^{-20t}\right) \\ &= 24e^{-20t} + 24 - 24e^{-20t} \\ &= 24 \end{aligned}$$

which is the expected identity.

Why is this important? Most of the techniques we will learn in this course *depend* upon the fact that the equation we are trying to solve is *linear*. In “*the wild*” you may be presented with (or, more likely *derive*) an equation and may not be explicitly told whether or not the equation is linear. If the equation is **not** linear you will find that most of the tools you learn in this course will not be applicable; you will most likely need to use a numerical method. You need to be able to tell the difference so you know what tools to use.

Example explicit solution: $u(x) = f(x)$.

Example implicit solution: $G(x, u) = 0$

Lecture 2 - Separable and Linear 1st order Equations

Objectives

The objectives of this lecture are:

- Define and describe the solution procedure for *separable* first order equations.
- Define and demonstrate the solution procedure for *linear* first order equations.

Separable Equations

A first order differential equation of the form shown below:

$$\frac{du}{dx} = g(u)h(x) \quad (4)$$

is said to be *separable* or have *separable variables*.

Note: There is **no** requirement that the 1st order equation be *linear*. This is the only technique that we will study in this course that can be applied to nonlinear equations.

THE SOLUTION METHOD for separable equations is, in principle, simple. For the separable differential equation given in Equation 4 we would separate and integrate:

$$\begin{aligned} \frac{du}{dx} &= g(u)h(x) \\ \frac{du}{g(u)} &= h(x)dx \\ \int \frac{1}{g(u)} du &= \int h(x) dx \end{aligned}$$

Generally speaking, one of your first checks for a first order equation should be: is it separable? If so, you should separate the variables and solve. The examples below are intended to illustrate the method. Note that in the final example, the integral cannot be done analytically.

Note: There are at least two complications here:

1. The solution you thus derive may be either implicit or explicit. An implicit solution is, as a practical matter, fairly inconvenient to deal with; and
2. It may not be possible to carry out the integrals analytically.

Nonetheless, we shall carry on and give it a try anyway.

Solve the following separable, first order differential equations .

Example 1:

$$\begin{aligned}\frac{du}{dx} &= \frac{u}{1+x} \\ \frac{du}{u} &= \frac{dx}{1+x} \\ \int \frac{d}{u} &= \int \frac{dx}{1+x} \\ \ln |u| + c_1 &= \ln |1+x| + c_2 \\ |u| &= e^{[\ln |1+x| + c_3]} \\ u(x) &= c|1+x|\end{aligned}$$

Example 2:

$$\begin{aligned}\frac{du}{dx} &= -\frac{x}{u} \\ \int u \, du &= - \int x \, dx \\ \frac{u^2}{2} &= -\frac{x^2}{2} + c \\ u(x) &= \sqrt{c - x^2}\end{aligned}$$

Example 3: Solve the first order initial value problem shown below:

$$\frac{du}{dx} = e^{-x^2}, \quad u(2) = 6, \quad 2 \leq x < \infty$$

$$\begin{aligned}du &= e^{-x^2} \, dx \\ \int_2^x \frac{du}{dt} \, dt &= \int_2^x e^{-t^2} \, dt \\ u(x) - u(2) &= \int_2^x e^{-t^2} \, dt \\ u(x) &= 6 + \int_2^x e^{-t^2} \, dt\end{aligned}$$

where we have used the dummy variable t in the integrals; the last integral will need to be evaluated numerically.

Linear Equations

A first-order differential equation of the form:

$$a_1(x) \frac{du}{dx} + a_0(x)u = g(x) \quad (5)$$

is said to be a first order *linear equation* in the dependent variable u .

When $g(x) = 0$, the first-order linear equation is said to be *homogeneous*; otherwise it is *non-homogeneous*.

WHEN SOLVING equations of this type it is customary to express it in the **standard form**:

$$\frac{du}{dx} + P(x)u = f(x) \quad (6)$$

The method for solving this equation makes use of the linearity property and we express the solution in the following way: $u(x) = u_c(x) + u_p(x)$. Inserting this into Equation 6 gives us:

$$\begin{aligned} \frac{d}{dx}[u_c + u_p] + P(x)[u_c + u_p] &= \\ \left[\frac{du_c}{dx} + P(x)u_c \right] + \left[\frac{du_p}{dx} + P(x)u_p \right] &= f(x) \end{aligned} \quad (7)$$

where $u_c(x)$ is the solution to the *associated homogeneous problem*:

$$\frac{du_c}{dx} + P(x)u_c = 0 \quad (8)$$

and $u_p(x)$ is a solution to:

$$\frac{du_p}{dx} + P(x)u_p = f(x) \quad (9)$$

We can see that Equation 8 is separable:

$$\begin{aligned} \frac{du_c}{dx} + P(x)u_c &= 0 \\ \frac{du_c}{u_c} &= -P(x) dx \\ \ln u_c + C &= - \int P(x) dx \\ u_c(x) &= e^{- \int P(x) dx + C_1} \\ u_c(x) &= e^{- \int P(x) dx} e^{C_1} \\ u_c(x) &= C e^{- \int P(x) dx} \end{aligned}$$

where $C = e^{C_1}$.

Note: it is sometimes customary to write the differential equation in *operator form* where the differential operator, $\mathcal{L} = a_1(x) \frac{d}{dx} + a_0(x)$, is applied to the function $u(x)$ to get $g(x)$; $\mathcal{L}u(x) = g(x)$

Notice that $g(x)$ is the only term in Equation 5 that does not include u or any of its derivatives.

When we say an operator is **linear**, what we mean is that the following relationships must hold:

1. $\mathcal{L}(\alpha u) = \alpha \mathcal{L}(u)$
 2. $\mathcal{L}(u + v) = \mathcal{L}(u) + \mathcal{L}(v)$
- for functions u, v and scalar constant α . Think of this as a *definition* of linearity.

The linear operator here is: $\mathcal{L} = \frac{d}{dx} + P(x)$. Equation 8 says $\mathcal{L}u_c = 0$; Equation 9 says $\mathcal{L}u_p = f(x)$; Equation 7 says that $\mathcal{L}(u_c + u_p) = 0 + f(x) = f(x)$.

What might trouble you now is: if we have u_p , is this not a solution to Equation 6? Why do we need u_c ? The next thing that should trouble you is that if u_p is a solution, by the linearity property of \mathcal{L} , so is u_p plus *any* constant multiple of u_c . The solution is not *unique*.

This will all be resolved when we recall that u_c will have an arbitrary constant through which we will be able to say that $u = u_c + u_p$ is a function describing *all* possible solutions of Equation 6 and the arbitrary constant(s) in u_c will be set so as to uniquely satisfy a given initial/boundary condition.

Note: Going forward, we will desist in making such peddling distinctions between constants. C_1 is an arbitrary constant, e^{C_1} is still an arbitrary constant. There is no real difference between C_1 and C and, in this author's humble opinion, they do not rate different symbols.

We NEED TO FIND a solution $u_p(x)$ to Equation 9. The technique we will use is called *variation of parameters*. It consists of looking for a solution in the form $y_p(x) = v(x)u_1(x)$, where $u_1(x) = e^{-\int P(x) dx}$ which is $u_c(x)$ with the arbitrary constant set to 1 and $v(x)$ might be thought of as some kind of weighting or *variational* function.

We will insert this proposed form of $y_p(x)$ into Equation 9:

$$\frac{d(vu_1)}{dx} + P(x)[v(x)u_1(x)] = f(x)$$

We apply the product rule to the first term and re-arrange to get:

$$\begin{aligned} u_1(x)\frac{dv}{dx} + v(x)\frac{du_1}{dx} + P(x)[v(x)u_1(x)] &= f(x) \\ v(x)\underbrace{\left[\frac{du_1}{dx} + P(x)u_1(x)\right]}_{=0} + u_1(x)\frac{dv}{dx} &= f(x) \\ u_1(x)\frac{dv}{dx} &= f(x) \end{aligned}$$

In the last line we can observe that the equation is *separable* and thus solve accordingly:

$$\begin{aligned} v(x) &= \int \frac{f(x)}{u_1(x)} dx \\ &= \int e^{\int P(x) dx} f(x) dx \end{aligned}$$

Now that we know what $v(x)$ must be, we can combine this with $u_1(x)$ to get $u_p(x)$:

$$u_p(x) = e^{-\int P(x) dx} \left[\int e^{\int P(x) dx} f(x) dx \right] \quad (10)$$

Equation 10 is messy and perhaps a bit scary but given definitions of $P(x)$ and $f(x)$ we might hope we can solve it anyway. We now have expressions for both u_c and u_p and we combine them to form the solution for the first-order linear equation:

$$u(x) = Ce^{-\int P(x) dx} + e^{-\int P(x) dx} \left[e^{\int P(x) dx} f(x) dx \right] \quad (11)$$

Method of Solution

In summary, once we have identified a problem to be first-order and linear, we will solve the problem using the following steps:

1. Write the equation in standard form (Equation 6)
2. Determine the integrating factor $\mu = e^{-\int P(x) dx}$.

3. Solve for the general solution $u(x)$ using Equation 11.
4. Apply initial/boundary condition if given.

Example: Solve the problem:

$$\frac{du}{dx} + u = x, \quad u(0) = 4$$

Step #1: The equation is already in standard form.

Step #2: Find the integrating factor μ .

$$mu = e^{-\int P(x) dx} = e^{-\int 1 dx} = e^{-x}$$

Step #3: Solve for the general solution $u(x)$ using Equation 11

$$\begin{aligned} u(x) &= Ce^{-x} + e^{-x} \int e^x x dx \\ &= Ce^{-x} + e^{-x} [xe^x - e^x] \\ &= Ce^{-x} + x - 1 \end{aligned}$$

← For the integral $\int e^x x dx$ we need to use integration by parts.

Step #4: Apply initial/boundary conditions if given

$$\begin{aligned} u(0) &= Ce^0 + 0 - 1 \\ &= C - 1 = 4 \\ \Rightarrow C &= 5 \\ u(x) &= 5e^x + x - 1 \end{aligned}$$

Assignment #1

State the order of the given ordinary differential equation and indicate if it is linear or non-linear.

1. $(1-x)u'' - 4xu' + 5u = \cos x$

2. $t^5u^{(4)} - t^3u'' + 6u = 0$

Verify the indicated function is an explicit solution of the given differential equation.

3. $2u' + u = 0, \quad u = e^{-x/2}$

4. $u'' - 6u' + 13u = 0, \quad u = e^{3x} \cos 2x$

Solve the given differential equation by separation of variables.

5. $\frac{du}{dx} = \sin 5x$

6. $dx + e^{3x}du = 0$

7. $\frac{dS}{dr} = kS$

8. $\frac{du}{dx} = x\sqrt{1-u^2}$

Find an explicit solution of the given initial-value problem.

9. $x^2 \frac{du}{dx} = u - xu, \quad u(-1) = -1$

Find the general solution of the given differential equation.

$$10. \frac{du}{dx} + u = e^{3x}$$

$$11. u' + 3x^2u = x^2$$

$$12. x\frac{du}{dx} - u = x^2 \sin x$$

Lecture 3 - Theory of Linear Equations

Objectives

The objectives of this lecture are:

- Introduce several theoretical concepts relevant to initial value problems and boundary value problems.
- Demonstrate use of the Wronskian to determine linear independence of solutions.
- Present some important theorems and definitions relevant to the theory of linear ordinary differential equations.

Initial Value Problems

For a linear differential equation, an n^{th} -order initial value problem (IVP) is given by the following governing equation and initial conditions:

$$\begin{aligned} \text{Governing Equation: } & a_n(x) \frac{d^n u}{dx^n} + a_{n-1}(x) \frac{d^{n-1} u}{dx^{n-1}} + \dots \\ & + a_1(x) \frac{du}{dx} + a_0(x)u = g(x) \quad (12) \end{aligned}$$

$$\text{Initial Conditions: } u(x_0) = u_0, u'(x_0) = u_1, \dots, u^{(n-1)}(x_0) = u_{n-1} \quad (13)$$

We seek a function defined on some interval containing x_0 that satisfies the differential equation with n conditions applied. The theorem below, which we will use by *citing* rather than *proving*, gives us assurance that, subject some fairly reasonable assumptions, such a solution will exist.

Theorem 1 (Existence and Uniqueness for IVPs)

If $a_n(x), a_{n-1}(x), \dots, a_1(x), a_0(x)$ and $g(x)$ are continuous on an interval \mathcal{I} , and if $a_n(x) \neq 0$ for every $x \in \mathcal{I}$, and if x_0 is any point in this interval, then a solution $u(x)$ of the IVP exists on the interval and it is unique.

Note: For an initial value problem, all of the initial conditions are provided at the same value of x ; in accordance with custom we call this x_0 . The name *initial* condition gives the implication that these conditions are at some “end” of the interval (beginning, left side, whatever) and in all examples and exercises in this text this is indeed the case. It is not, however, a requirement. Generally for an n^{th} -order IVP you will need n conditions.

FOR THIS CLASS we will adopt a fairly operational definition of continuity: if you can draw the function throughout the specified interval without picking up your pencil or without diverging to infinity, then the function is continuous.

Consider, as an example, the following initial value problem:

$$u'' - 4u = 12x, \quad u(0) = 4, \quad u'(0) = 1 \quad (14)$$

This IVP satisfies the conditions of Theorem 1 since all of the coefficients and $g(x)$ are continuous and a_2 is constant and nonzero; hence a unique solution exists on any interval and that solution is unique.

Here is an IVP that does *not* satisfy the criteria of Theorem 1:

$$x^2u'' - 2xu' + 2u = 6, \quad u(0) = 3, \quad u'(0) = 1 \quad (15)$$

In this case, the coefficients and $g(x)$ are all continuous but $a_2(x)$ is equal to zero at $x = 0$. This might not be a problem—i.e. if $x = 0$ is not in the interval of interest for the IVP then we are okay—but since $x_0 = 0$, $x = 0$ *must* be in the domain for the theorem to apply. So we have no assurances that a solution exists or, if a solution does exist, it may not be unique.

Take a moment to verify that $u(x) = 3e^{2x} + e^{-2x} - 3x$ satisfies both the governing equation and initial conditions given in Equation 14 and thus is *the* unique solution to this IVP.

You should take a moment to verify that $u(x) = cx^2 + x + 3$ is a solution for the IVP given in Equation 15 for *any* choice of parameter c .

Boundary Value Problems

For this section let us, without undue loss of generality, consider a 2nd-order boundary value problem (BVP):

$$\text{Governing Equation: } a_2(x) \frac{d^2u}{dx^2} + a_1(x) \frac{du}{dx} + a_0(x)u = g(x) \quad (16)$$

$$\text{Boundary Conditions: } y(a) = y_0, \quad y(b) = y_1, \quad a \neq b \quad (17)$$

DEPENDING ON THE boundary conditions, BVPs may have no solutions, one unique solution, or infinitely many solutions.

Example: The equation $u'' + 16u = 0$ has the general solution $u(t) = c_1 \cos(4t) + c_2 \sin(4t)$. Consider the three different sets of boundary conditions provided below:

- a) $u(0) = 0, \quad u(\pi/2) = 0$. Application of the first boundary condition gives us $c_1(1) + c_2(0) = 0 \Rightarrow c_1 = 0$. The second boundary condition is $c_2 \sin(2\pi) = 0$, which is true for *any* value of c_2 . Therefore there problem has infinitely many solutions.
- b) $u(0) = 0, \quad u(\pi/8) = 0$. The first boundary condition again gives us $c_1 = 0$; the second condition $c_2 \sin(4\frac{\pi}{8}) = 0$ is only satisfied if $c_2 = 0$. Thus $c_1 = c_2 = 0$; only the trivial solution, $u = 0$, satisfies

Almost all of the applications we will consider for this class will involve 2nd-order operators. The way we derive important boundary-value problems from underlying physical laws like conservation of mass and conservation of energy lead to them being 2nd-order. You should think about this while you are sitting in your fluid dynamics class and equations are being derived for conservation of mass and momentum for viscous incompressible fluid flow or when you are sitting in heat transfer class and the heat equation is being derived from conservation of energy principles. Probably the most obvious counterexample is beam theory which involves a 4th-order operator.

both the differential equation and boundary conditions. This is not a very interesting solution but at least it is a *solution* so we will take this as an example of a BVP having a unique solution.

- c) $u(0) = 0, u(\pi/2) = 1$. In this case, again $c_1 = 0$ from the first boundary condition. This leaves the second boundary condition: $c_2 \sin(4\frac{\pi}{2}) = c_2(0) = 1$ which cannot be satisfied for any value of c_2 . In this case no solution exists.

For applications, we will generally be only interested in *non-trivial* solutions; that is, solutions that are not identically equal to zero.

Superposition and Linear Dependence

In this section some important theorems regarding IVPs and BVPs will be presented. No attempt will be made to prove these theorems; we will simply take these theorems as facts that are relevant for this course that you should try to understand as best you can.

Theorem 2 (Superposition Principle for Homogeneous Equations)

Let u_1, u_2, \dots, u_k be solutions of a homogeneous n^{th} -order linear differential equation. Then any linear combination of those solutions

$$u = c_1 u_1 + c_2 u_2 + \cdots + c_k u_k$$

where c_1, c_2, \dots, c_k are arbitrary constants, is also a solution.

As an example, If we denote the linear homogeneous differential equation as \mathcal{L} , then $\mathcal{L}(u_i) = 0$ for any $i \in [1, 2, \dots, k]$. By the linearity property of \mathcal{L} , for any constants α and β :

$$\begin{aligned} \mathcal{L}(\alpha u_i + \beta u_j) &= \alpha \mathcal{L}(u_i) + \beta \mathcal{L}(u_j) \\ &= \alpha(0) + \beta(0) \\ &= 0 \end{aligned}$$

thus $\alpha u_i + \beta u_j$ is a solution.

Theorem 3 (Linear Dependence / Independence of Functions)

A set of functions $f_1(x), f_2(x), \dots, f_k(x)$ is said to be linearly dependent on an interval \mathcal{I} if there exist constants c_1, c_2, \dots, c_k , not all of which are zero, such that

$$c_1 f_1(x) + c_2 f_2(x) + \cdots + c_k f_k(x) = 0$$

for every $x \in \mathcal{I}$. If the set of functions is not linearly dependent, it is linearly independent.

Repeatedly throughout this course we will want to clarify whether or not two or more functions are linearly independent of each other. I think most engineers have a general idea of what it is we mean when we say two functions are linearly independent or dependent but Theorem 3 specifies what these things mean *mathematically*.

Note: It is essential that both the governing equation and conditions (boundary or initial) for the linear differential equation are homogeneous. As a reminder, this means that all terms in the governing equation and boundary conditions must either a) involve the dependent variable or one of its derivatives; or b) be equal to zero.

Question: What if a member of the set of functions is $f(x) = 0$?

Answer: The set will no longer be linearly independent. The trivial function $f(x) = 0$ is not linearly independent from anything.

WE NEED A TEST to help us determine whether or not the members of a set are linearly independent. This will be especially important as we evaluate solutions to a linear homogeneous differential equation. Even if you are the sort of savant who can, by inspection, always detect linear dependence, you might have a hard time convincing your friends that your assessment is always correct. Luckily, there is a theorem that provides a suitable test that can serve as irrefutable evidence of the state of linear dependence/independence of functions.

Theorem 4 (Criterion for Linearly Independent Solutions)

Let u_1, u_2, \dots, u_n be solutions of a homogeneous linear n^{th} -order differential equation defined on an interval \mathcal{I} . Then the set of solutions is linearly independent on the interval if and only if the Wronskian of the solution is non-zero for every $x \in \mathcal{I}$.

The Wronskian is a function that takes functions as arguments and returns a scalar numeric quantity and is defined in Equation 18:

$$W(u_1, u_2, \dots, u_n) = \begin{vmatrix} u_1 & u_2 & \cdots & u_n \\ u'_1 & u'_2 & \cdots & u'_n \\ \vdots & \vdots & \ddots & \vdots \\ u_1^{(n-1)} & u_2^{(n-1)} & \cdots & u_n^{(n-1)} \end{vmatrix} \quad (18)$$

where $|\cdot|$ denotes the matrix determinant. For large values of n this is difficult to calculate but, for the case $n = 2$, engineering students should be familiar with the formula:

$$W(u_1, u_2) = \begin{vmatrix} u_1 & u_2 \\ u'_1 & u'_2 \end{vmatrix} = u_1 u'_2 - u'_1 u_2 \quad (19)$$

Students of this class should also be able to find determinants for 3×3 matrices. If you have forgotten that formula, consult your favorite textbook or online resource for a reminder.

Example: Show that the functions $u_1 = e^{3x}$ and $u_2 = e^{-3x}$ are linearly independent solutions to the homogeneous linear equation $u'' - 9u = 0$ for every $x \in (-\infty, \infty)$.

Solution: The Wronskian is given by:

The reader should verify that both $u_1 = e^{3x}$ and $u_2 = e^{-3x}$ satisfy the given differential equation.

$$\begin{aligned}
W &= \begin{vmatrix} e^{3x} & e^{-3x} \\ 3e^{3x} & -3e^{-3x} \end{vmatrix} \\
&= e^{3x}(-3e^{-3x}) - 3e^{3x}(e^{-3x}) \\
&= 3e^{3x-3x} - 3e^{3x-3x} \\
&= 3 - 3 \\
&= 6
\end{aligned}$$

Since $6 \neq 0$ for all $x \in (-\infty, \infty)$ the solutions are linearly independent.

Definition 1 (Fundamental Set of Solutions)

Any set $\{u_1, u_2, \dots, u_n\}$ of n linearly independent solutions of the homogeneous linear n^{th} -order differential equation on an interval is said to be a fundamental set of solutions on an interval \mathcal{I} .

Theorem 5 (Existence of a Fundamental Set)

There exists a fundamental set of solutions for the homogeneous linear n^{th} -order differential equation on an interval \mathcal{I} .

Definition 2 (General Solution—Homogeneous Equation)

Let $\{u_1, u_2, \dots, u_n\}$ be a fundamental set of solutions to the homogeneous linear n^{th} -order differential equation defined on an interval \mathcal{I} , then the general solution is:

$$u(x) = c_1 u_1(x) + c_2 u_2(x) + \cdots + c_n u_n(x)$$

IT IS IMPORTANT to understand from the above that:

- Any possible solution to the homogeneous, linear, n^{th} -order differential equation can be constructed by setting the coefficients of the general solution; and
- there is **no** solution that can be constructed from functions that are linearly independent from the general solution.

General Solution for a Non-homogeneous Problem

Recall: “non-homogeneous” for a linear n^{th} -order differential equation means that $g(x) \neq 0$. If u_p is any particular solution to the non-homogeneous, linear, n^{th} -order ODE on an interval \mathcal{I} and $u_c = c_1 u_1(x) + c_2 u_2(x) + \cdots + c_n u_n(x)$ is the general solution to the associated homogeneous ODE (called the *complementary* solution) then the general solution to the non-homogeneous ODE is:

$$u = u_c + u_p$$

Note: This is different than saying that a BVP or IVP has a solution. This theorem is only referring to the differential equation; not the boundary or initial conditions.

Example: By substitution it can be seen that $u_p = -\frac{11}{12} - \frac{1}{2}x$ is a particular solution to $u''' - 6u'' + 11u' - 6u = 3x$. The general solution to the associated homogeneous problem is $u_c = c_1e^x + c_2e^{2x} + c_3e^{3x}$. Consequently, the general solution to the linear non-homogeneous problem is:

$$\begin{aligned} u(x) &= u_c + u_p \\ &= c_1e^x + c_2e^{2x} + c_3e^{3x} - \frac{11}{12} - \frac{1}{2}x \end{aligned}$$

You are, again, strongly encouraged to verify that u_p satisfies the given equation and that u_c satisfies the associated homogeneous equation.

Lecture 4 - Homogeneous Linear Equations with Constant Coefficients

Objectives

The objectives of this lecture are:

- Review the solution methodology for homogeneous linear equations with constant coefficients.
- Illustrate this method with several examples.

Introduction

In this lecture we will review the well-trod ground of your differential equations class and remind ourselves how to solve linear, constant coefficient, homogeneous, n^{th} -order differential equations.

These equations have the general form shown in Equation 20:

$$a_n u^{(n)} + a_{n-1} u^{(n-1)} + \cdots + a_1 u' + a_0 u = 0 \quad (20)$$

where the coefficients are real and constant and $a_n \neq 0$.

THE BASIC STRATEGY is to assume the solution is of the form: $u(x) = e^{mx}$. For the case of 2nd-order equations, we get:

$$\begin{aligned} a_2 m^2 e^{mx} + a_1 m e^{mx} + a_0 e^{mx} &= 0 \\ e^{mx} (a_2 m^2 + a_1 m + a_0) &= 0 \end{aligned}$$

where the last line above is called the auxiliary equation:

$$am^2 + bm + c = 0 \quad (21)$$

From the well-known quadratic equation, solutions are: $m = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Solution of this equation gives the following three cases:

1. **Distinct Real Roots** In this case $m_1 \neq m_2$ and the general solution is of the form:

$$u(x) = c_1 \underbrace{e^{m_1 x}}_{u_1(x)} + c_2 \underbrace{e^{m_2 x}}_{u_2(x)} \quad (22)$$

If $u(x) = e^{mx}$ then, of course, $u' = me^{mx}$ and $u'' = m^2 e^{mx}$.

Here we re-name the constants so Equation 21 takes a familiar form.

Using tools from the last lecture you should recognize that $u_1(x)$ and $u_2(x)$ are linearly independent for all $x \in (-\infty, \infty)$, and thus form a fundamental set of solutions.

AN IMPORTANT SPECIAL CASE is when m_1 and m_2 are roots of a positive real number and thus $m_1 = -m_2$. This happens when the governing equation is of the form:

$$u'' - k^2 u = 0 \quad (23)$$

The general solution is:

$$u(x) = c_1 e^{-kx} + c_2 e^{kx} \quad (24)$$

For reasons that will become clear later in the course, it is sometimes useful to re-express the solution shown in Equation 24 in terms of the functions $\cosh()$ and $\sinh()$. These functions are defined as linear combinations of exponentials as shown below and plotted in Figure 1.

$$\begin{aligned}\cosh x &= \frac{e^x + e^{-x}}{2} \\ \sinh x &= \frac{e^x - e^{-x}}{2}\end{aligned}$$

2. **Real Repeated Roots** In this case $m_1 = m_2$. One solution is:

$$u_1(x) = e^{m_1 x} \quad (25)$$

The other solution so derived is, of course, the same and thus we do not have two linearly independent solutions as required to form a fundamental set of solutions for a 2nd-order linear homogeneous equation.

IT CAN BE SHOWN that a second linearly independent solution can be formed by multiplying $u_1(x)$ by the independent variable, x :

$$u_2(x) = xu_1(x) = xe^{mx}$$

and thus the general solution for this case is:

$$u(x) = c_1 e^{mx} + c_2 x e^{mx} \quad (26)$$

3. **Conjugate Complex Roots** In this case the discriminant, $b^2 - 4ac$, is negative so its square root is imaginary. This results in m_1 and m_2 being complex conjugates which we will express as: $m_1 = \alpha + i\beta$ and $m_2 = \alpha - i\beta$.

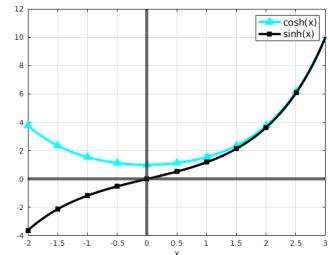


Figure 1: Plot of $\cosh x$ and $\sinh x$

When applying boundary conditions and resolving unknown constants in a general solution to a differential equation, it is helpful that $\sinh 0 = 0$. In contrast, neither e^x nor e^{-x} is equal to zero for finite values of x .

i.e. from the quadratic equation, $b^2 - 4ac = 0$

This result is derived using a technique referred to as *reduction of order*. We will not take the time to cover it in this class (or in this book) but is concisely described in section 3.2 of Zill. At a minimum you might at least confirm for yourself that a) $xu_1(x)$ is a solution to the equation; and b) use the Wronskian to confirm that it is linearly independent from $u_1(x)$.

The general solution is:

$$\begin{aligned} u(x) &= c_1 e^{(\alpha+i\beta)x} + c_2 e^{(\alpha-i\beta)x} \\ &= e^{\alpha x} \left(c_1 e^{i\beta x} + c_2 e^{-i\beta x} \right) \end{aligned}$$

The complex exponentials in the last equation can be re-expressed using the Euler Formula:

$$\begin{aligned} e^{i\beta x} &= \cos \beta x + i \sin \beta x \\ e^{-i\beta x} &= \cos \beta x - i \sin \beta x \end{aligned}$$

which is slightly more convenient insofar as the solutions are no longer expressed as complex exponentials; this expression also breaks the solutions down into their real and complex parts. It can be shown that both the real and imaginary parts of the solution must satisfy the differential equation *independently*. This fact allows us yet again to re-express the solution in a still more simple form that does not involve complex numbers at all:

$$u(x) = e^{\alpha x} (c_1 \cos \beta x + c_2 \sin \beta x) \quad (27)$$

ANOTHER IMPORTANT special case is when the solution is *pure imaginary* (i.e. $\alpha = 0$) so the solution is:

$$u(x) = c_1 \cos \beta x + c_2 \sin \beta x \quad (28)$$

These solutions arise when the governing equation is as shown in Equation 29:

$$u'' + k^2 u = 0 \quad (29)$$

The roots are $m_{1,2} = \pm ik$ and the general solution is:

$$u(x) = c_1 \cos kx + c_2 \sin kx \quad (30)$$

This equation will be revisited throughout the course as it repeatedly comes up in applications.

Three Examples

The cases described above will be illustrated with three examples:

Example #1: Find the general solution to $2u'' - 5u' - 3u = 0$. Inserting $u = e^{mx}$ into the equation gives us the auxiliary equation:

$$2m^2 - 5m - 3 = (2m + 1)(m - 3)$$

with roots: $m_1 = -\frac{1}{2}$ and $m_2 = 3$. These are real, distinct roots so the general solution is:

$$u(x) = c_1 e^{-x/2} + c_2 e^{3x}$$

Example #2: Find the general solution to $u'' - 10u' + 25u = 0$. The auxiliary equation is:

$$m^2 - 10m + 25 = (m - 5)(m - 5)$$

with (repeated) roots: $m_1 = 5$ and $m_2 = 5$. These are real, repeated roots so the general solution is:

$$u(x) = c_1 e^{5x} + c_2 x e^{5x}$$

Example #3: Find the general solution to $4u'' + 4u' + 17u = 0$, $u(0) = -1$, $u'(0) = 2$.

This is an initial value problem with continuous (and constant) coefficients and with $a_2(x) \neq 0$ for all values of x . We know from Theorem 1 that a unique solution exists. We will first find the general solution, then apply the initial conditions to resolve the unknown coefficients to reveal the solution.

The auxiliary equation is:

$$4m^2 + 4m + 17 = 0$$

using the quadratic equation, gives us:

$$\begin{aligned} \frac{-4 \pm \sqrt{16 - 4(4)(17)}}{2(4)} &= -\frac{1}{2} \pm \frac{\sqrt{-256}}{8} \\ &= -\frac{1}{2} \pm \frac{-16}{8} \\ &= -\frac{1}{2} \pm 2i \end{aligned}$$

We can see that it must be an *initial* value problem because the conditions are both given at the same location, $x_0 = 0$.

This gives us complex conjugate roots and the general solution is:

$$u(x) = e^{-x/2} (c_1 \cos 2x + c_2 \sin 2x)$$

Applying the initial condition $u(0) = -1$ gives us:

$$\begin{aligned} u(0) &= e^0 (c_1 \cos 0 + c_2 \sin 0) \\ &= 1(c_1(1) + c_2(0)) \\ &= c_1 = -1 \end{aligned}$$

To apply the second initial condition we need to use the chain-rule and product rule to differentiate the general solution. This gives us:

$$\begin{aligned} u'(x) &= -\frac{1}{2}e^{-x/2}c_1 \cos 2x - 2e^{-x/2}c_1 \sin 2x + \\ &\quad -\frac{1}{2}e^{-x/2}c_2 \sin 2x + 2e^{-x/2}c_2 \cos 2x \end{aligned}$$

Evaluating $u'(0)$ and substituting $c_1 = -1$ gives us:

$$\begin{aligned} u'(0) &= -\frac{1}{2}(1)(-1)(1) + (1)(2)c_2(1) \\ &= \frac{1}{2} + 2c_2 = 2 \\ \Rightarrow 2c_2 &= \frac{3}{2} \\ c_2 &= \frac{3}{4} \end{aligned}$$

Both constants are now known and the unique solution is:

$$u(x) = e^{-x/2} \left(-\cos 2x + \frac{3}{4} \sin 2x \right)$$

Lecture 5 - Non-homogeneous Linear Equations with Constant Coefficients

Objectives

The objectives of this lecture are:

- Describe the Method of Undetermined Coefficients for solving non-homogeneous linear equations with constant coefficients.
- Carry out some examples to illustrate the methods.

Background

In this lecture we will review a method for finding solutions to non-homogeneous linear equations with constant coefficients.

CONSIDER THE EQUATION

$$a_n u^{(n)} + a_{n-1} u^{(n-1)} + \cdots + a_1 u' + a_0 u = g(x) \quad (31)$$

where:

- the coefficients a_i , $i \in [1, 2, \dots, n]$ are constants; and
- the function $g(x)$ is a constant, a polynomial function, exponential function, sine or cosine, or finite sums or products of these functions.

The general solution, $u(x)$, can be constructed as $u_c(x) + u_p(x)$ where:

- $u_c(x)$ is the complementary solution which, as you should recall, is the general solution to the associated homogeneous problem—i.e. Equation 31 with $g(x) = 0$; and
- $u_p(x)$ is (any) particular solution—that is, a not-necessarily-unique function that satisfies Equation 31.

We spent the last lecture describing how to find $u_c(x)$. The question this lecture will hope to answer is: “How do I find $u_p(x)$?”

To be perfectly honest, we spend very little time in this class dealing with non-homogeneous equations of any kind. Many of those types of equations are beyond our ability to solve analytically so we turn to numerical methods instead. Nonetheless there is value in reminding ourselves how to construct solutions for those cases where we can.

Method of Undetermined Coefficients

One method for finding $u_p(x)$ is called the Method of Undetermined Coefficients.¹

THERE ARE THREE rules that comprise this technique:

1. **Basic Rule:** Based on the terms in $g(x)$, select the appropriate form for $u_p(x)$ using Table 1.

Term in $g(x)$	Choice for $u_p(x)$
$ke^{\gamma x}$	$Ae^{\gamma x}$
$kx^n, (n = 0, 1, \dots)$	$K_n x^n + K_{n-1} x^{n-1} + \dots + K_1 x + K_0$
$k \cos \omega x$	$\} K \cos \omega x + M \sin \omega x$
$k \sin \omega x$	
$ke^{\alpha x} \cos \omega x$	$\} e^{\alpha x} (K \cos \omega x + M \sin \omega x)$
$ke^{\alpha x} \sin \omega x$	

2. **Modification rule:** If $u_p(x)$ obtained by the **Basic Rule** happens to be a solution to the associated homogeneous equation, multiply $u_p(x)$ from the table by x (or x^2 if needed).²
3. **Sum rule:** If $g(x)$ is a linear combination of terms from the left-hand column, construct $u_p(x)$ from a linear combination of the corresponding entries in the right-hand column.

For the remainder of this lecture, we will practice applying these rules to example problems.

Example: Solve $u'' + 4u' - 2u = 2x^2 - 3x + 6$.

Step #1: Find the general solution to the associated homogeneous equation.

The auxiliary equation is: $m^2 + 4m - 2 = 0$. Using the quadratic equation gives us:

$$\begin{aligned} m &= \frac{-4 \pm \sqrt{16 - (4)(1)(-2)}}{2(1)} \\ &= -2 \pm \frac{\sqrt{24}}{2} \\ &= -2 \pm \sqrt{6} \end{aligned}$$

so $u_c(x) = c_1 e^{(-2+\sqrt{6})x} + c_2 e^{(-2-\sqrt{6})x}$.

¹ Some people lovingly refer to this technique as "The Method of Guessing."

Table 1: Forms of $u_p(x)$ for given terms in $g(x)$

² It's a bad example but consider:

$$u'' = 3$$

where $u_c(x) = c_1 x + c_2$. Since $g(x)$ is a constant, the basic rule (second row) dictates that $u_p(x)$ should also be a constant, K_0 . That will not work in this case since a constant term is also part of $u_c(x)$. If we multiply by x , then we still have a part of $u_c(x)$. If we multiply by x^2 , the method will work. Of course in this case the equation is separable and we would solve it by separating and integrating but hopefully this clarifies the need for the Modification rule.

Here you are expected to examine the associated homogeneous problem as $u'' + 4u' - 2u = 0$, identify it as constant coefficient and linear, and solve by assuming $u = e^{mx}$ and thus deriving the auxiliary equation shown without further prompting.

Step #2: Apply the method of undetermined coefficients to construct a candidate $u_p(x)$.

Since $g(x)$ is a second-order polynomial, the table tells us $u_p(x)$ is in the general form of a second-order polynomial.

$$u_p(x) = K_2x^2 + K_1x + K_0$$

We insert this into the governing equation and this gives us:

$$2K_2 + 4(2K_2x + K_1) - 2(K_2x^2 + K_1x + K_0) = 2x^2 - 3x + 6$$

Now we need to equate the coefficient for each power of x :

$$\begin{array}{lll} x^2: & -2K_2 & = 2 \\ x: & 8K_2 - 2K_1 & = -3 \\ 1: & 2K_2 + 4K_1 - 2K_0 & = 6 \end{array}$$

Luckily for us, this system of equations is structured such that it can easily be solved. We see by inspection that $K_2 = 2/-2 = -1$; this can be plugged into the second equation to find $K_1 = -5/2$ and then we can solve the last equation to find that $K_0 = -9$. Thus the particular solution is:

$$u_p(x) = -x^2 - \frac{5}{2}x - 9$$

Step #3: Construct the general solution: $u(x) = u_c(x) + u_p(x)$.

We now have both the complementary solution and a particular solution. We form the general solution to the equation by adding them together.

$$\begin{aligned} u(x) &= u_c(x) + u_p(x) \\ &= c_1e^{(-2+\sqrt{6})x} + c_2e^{(-2-\sqrt{6})x} - x^2 - \frac{5}{2}x - 9 \end{aligned}$$

Example: Solve $u'' - 5u' + 4u = 8e^x$.

Step #1: Find the general solution to the associated homogeneous problem.

The auxiliary equation is $m^2 - m + 4 = 0$ the left side of which can easily be factored to give $(m - 4)(m - 1) = 0$; the roots of which are $m_1 = 4, m_2 = 1$. The complementary solution is:

$$u_c(x) = c_1e^{4x} + c_2e^x$$

In general you cannot expect this to go so nicely. What you *can* hope for is that the equations you derive will have a unique solution. We could re-write the system in the form of a matrix-vector equation:

$$\begin{bmatrix} 0 & 0 & -2 \\ 0 & -2 & 8 \\ -2 & 4 & 2 \end{bmatrix} \begin{bmatrix} K_0 \\ K_1 \\ K_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 6 \end{bmatrix}$$

If the solution of such a matrix cannot be done by inspection and simple algebra as it was in this case, we could use tools like MATLAB to solve the linear system of equations. This topic and much more is covered in the numerical methods portion of this text.

Question: Why, again, do we need the constants c_1 and c_2 ?

Answer: Because we have not yet applied initial/boundary conditions. If those conditions are provided—two conditions for a 2nd-order problem—then we can resolve the constants.

Step #2: Apply the method of undetermined coefficients to construct $u_p(x)$.

Inspecting Table 1 we see that $u_p(x)$ should be of the form Ae^x . If that function seems vaguely familiar it may be because e^x is part of the complementary solution.

Question: If you plug Ae^x into your governing equation, without doing any calculations, what value should you get?

Answer: You will get zero! Why? Because e^x is one of the two linearly independent solutions to the associated homogeneous problem.

Question: What do I do now?

Answer: Invoke the Modification Rule—this is, after all, the reason why the rule exists—and multiply u_p by x . We now have $u_p(x) = Axe^x$.

We insert this function for $u_p(x)$ into the equation and we get:

$$2Ae^x + Axe^x - 5(Ae^x + Axe^x) + 4Axe^x = 8e^x$$

Combine terms and solve for A :

$$\begin{aligned} 2Ae^x - 5Ae^x &= 8e^x \\ -3Ae^x &= 8e^x \\ A &= -\frac{8}{3} \end{aligned}$$

So the particular solution is:

$$u_p(x) = -\frac{8}{3}xe^x$$

Step #3: Construct the general solution: $u(x) = u_c(x) + u_p(x)$.

$$\begin{aligned} u(x) &= u_c(x) + u_p(x) \\ &= c_1e^{4x} + c_2e^x - \frac{8}{3}xe^x \end{aligned}$$

Note: If any of this seems at all sketchy to you, the good news is that you need not worry if your proposed $u_p(x)$ is any good; you can just plug it into the differential equation and find out!

THIS LAST EXAMPLE illustrates the use of the Sum Rule; it also includes an initial condition so the unique solution to the initial value problem can be found.

Example: Solve the initial value problem: $u'' + u = 4x + 10 \sin x$ with initial conditions $u(\pi) = 0$, $u'(\pi) = 2$.

Step #1: Find the general solution to the associated homogeneous problem.

The auxiliary equation is: $m^2 + 1 = 0$, therefore $m = \pm i$ and $u_c(x)$ can be found as:

$$u_c(x) = c_1 \cos x + c_2 \sin x$$

Step #2: Apply the method of undetermined coefficients to construct $u_p(x)$.

For this problem, $g(x) = 4x + 10 \sin x$ has two terms, so we will construct $u_p(x)$ using one term at a time; $u_{p_1}(x)$ using $4x$ and $u_{p_2}(x)$ using $10 \sin x$.

It's the linearity property of $\mathcal{L} = \frac{d^2}{dx^2} + 1$ that makes this possible. If $\mathcal{L}(u_{p_1}) = 4x$ and $\mathcal{L}(u_{p_2}) = 10 \sin x$ then $\mathcal{L}(u_{p_1} + u_{p_2}) = 4x + 10 \sin x$.

Step #2.a: Find $u_{p_1}(x)$.

According to Table 1, when $g(x) = 4x$, we should select $u_{p_1} = K_1 x + K_0$. Inserting this into the differential equation gives us: $K_1 x + K_0 = 4x$. By inspection we can see that $K_0 = 0$ and $K_1 = 4$ so $u_{p_1}(x) = 4x$.

Step #2.b: Find $u_{p_2}(x)$.

According to Table 1, when $g(x) = 10 \sin x$, we should select $u_{p_2} = K \cos x + M \sin x$. Now that we have done this a couple of times we should be on the alert for portions of the complementary solution cropping up in our guesses for $u_p(x)$. We thus immediately see that we need to multiply u_{p_2} by x . If we do this and insert $Kx \cos x + Mx \sin x$ into the differential equation we get:

$$\begin{aligned} & (-2K - Mx) \sin x + (2M - Kx) \cos x + \dots \\ & Kx \cos x + Mx \sin x = 10 \sin x \end{aligned}$$

Matching coefficients for $\sin x$ and $\cos x$ on both sides of the above equation leads us to conclude that $M = 0$ and $-2K = 10$. Therefore $K = -5$ and $u_{p_2}(x) = -5x \cos x$.

Step #3: Construct the general solution: $u(x) = u_c(x) + u_p(x)$.

Again, there is no harm in testing your proposed $u_{p_2}(x)$ to see if it does indeed produce the expected result.

$$\begin{aligned} u(x) &= u_c(x) + u_p(x) \\ &= u_c(x) + u_{p_1}(x) + u_{p_2}(x) \\ &= c_1 \cos x + c_2 \sin x + 4x - 5x \cos x \end{aligned}$$

All that remains is to apply the initial conditions.

$$\begin{aligned} u(\pi) &= c_1(-1) + c_2(0) + 4\pi - 5(\pi)(-1) \\ &= -c_1 + 9\pi = 0 \\ \Rightarrow c_1 &= 9\pi \end{aligned}$$

Applying the initial condition $u' = 2$:

$$u'(\pi) = -9\pi(0) + c_2(-1) + 4 - 5(-1) + 5\pi(0) = 2$$

Solving for c_2 gives us: $c_2 = 7$; folding this into the general solution:

$$u(x) = 9\pi \cos x + 7 \sin x + 4x - 5x \cos x$$

Assignment #2

The given family of functions is the general solution of the differential equation on the indicated interval. Find a member of the family (i.e. find the values for the constants c_1 and c_2) that is a solution of the initial-value problem.

1. $u = c_1 e^x + c_2 e^{-x}; \quad u'' - u = 0, \quad u(0) = 0, \quad u'(0) = 1$

2. $u = c_1 x + c_2 x \ln x, \quad (0, \infty), \quad x^2 u'' - xu' = 0, \quad u(1) = 3, \quad u'(1) = -1$

The given two-parameter family is a solution of the indicated differential equation on the interval $(-\infty, \infty)$. Determine if a member of the family can be found that satisfies the boundary conditions.

3. $u = c_1 e^x \cos x + c_2 e^x \sin x; \quad u'' - 2u' + 3u = 0$

- (a) $u(0) = 1, \quad u'(\pi) = 0$
- (b) $u(0) = 1, \quad u(\pi) = -1$
- (c) $u(0) = 1, \quad u(\pi/2) = 1$
- (d) $u(0) = 0, \quad u(\pi) = 0$

Determine if the given set of functions is linearly dependent or linearly independent on the interval $(-\infty, \infty)$.

4. $f_1(x) = x, \quad f_2(x) = x^2, \quad f_3(x) = 4x - 3x^2$

5. $f_1(x) = 1 + x, \quad f_2(x) = x, \quad f_3(x) = x^2$

Verify that the given two-parameter family of functions is the general solution of the non-homogeneous differential equation on the indicated interval.

6. $u'' - 7u' + 10u = 24e^x, \quad u = c_1 e^{2x} + c_2 e^{5x} + 6e^x, \quad (-\infty, \infty)$

Find the general solution to the given second-order differential equation.

7. $4u'' + u' = 0$

8. $u'' - u' - 6u = 0$

9. $u'' + 8u' + 16u = 0$

10. $u'' + 9u = 0$

Solve the given initial-value problem.

11. $u'' + 16u = 0, \quad u(0) = 2, \quad u'(0) = -2$

12. $u'' - 4u' - 5u = 0, \quad u(1) = 0, \quad u'(1) = 2$

13. $u'' + u = 0, \quad u'(0) = 0, \quad u'(\pi/2) = 0$

Solve the given differential equation using the Method of Undetermined Coefficients.

14. $u'' - 10u' + 25u = 30x + 3$

15. $u'' + 3u = -48x^2e^{3x}$

Solve the given initial-value problem.

16. $5u'' + u' - 6x, \quad u(0) = 0, \quad u'(0) = -10$

Solve the given boundary-value problem.

17. $u'' + u = x^2 + 1, \quad u(0) = 5, \quad u(1) = 0$

Solve the given initial-value problem in which the input function $g(x)$ is discontinuous. (**Hint:** Solve the problem on two intervals and then find a solution so that u and u' are continuous at the boundary of the interval.)

$$18. \quad u'' + 4u = g(x), \quad u(0) = 1, \quad u'(0) = 2$$

$$g(x) = \begin{cases} \sin x & 0 \leq x \leq \pi/2 \\ 0 & x > \pi/2 \end{cases}$$

Lecture 6 - Cauchy-Euler Equations

Objectives

The objectives of this lecture are:

- Introduce Cauchy-Euler equations and demonstrate a method of solution.
- Carry out some examples to illustrate the methods for 2nd-order, homogeneous Cauchy-Euler equations.

Cauchy-Euler Equations

A linear differential equation of the form:

$$a_n x^n \frac{d^n u}{dx^n} + a_{n-1} x^{n-1} \frac{d^{n-1} u}{dx^{n-1}} + \cdots + a_1 x \frac{du}{dx} + a_0 u = g(x) \quad (32)$$

is called a Cauchy-Euler equation.

TAKE NOTE OF the relationship between the exponent of x in the coefficients and the order of the differential operators. This correspondence between the decreasing power of x in the coefficient and the decreasing order of the differential operator is characteristic of this type of equation and is the way you should recognize it.

Also observe that this equation is *linear*; if $g(x) = 0$ it is homogeneous, otherwise it is non-homogeneous. For this lecture we will focus our attention on the homogeneous, 2nd-order Cauchy-Euler equation:

$$ax^2 \frac{d^2 u}{dx^2} + bx \frac{du}{dx} + cu = 0 \quad (33)$$

Lastly you should be alert to the fact that the coefficient for the highest order derivative is zero at $x = 0$; consequently we will restrict the interval of interest for these equations to $x \in (0, \infty)$.

It is the corresponding *change* in power/order that matters. The equation: $a \frac{d^2 u}{dx^2} + \frac{1}{x^2} u = 0$ is also a Cauchy-Euler equation since the power of x in the coefficient goes from 0 to -2 while the order of the differential operator goes from 2nd to 0.

Reminder: The notation $(0, \infty)$ means that the interval is *open* and that we do *not* include the endpoints. A *closed* interval—where we *do* include the endpoints—is denoted with square brackets such as $[a, b]$.

THE BASIC STRATEGY in solving these equations is to try a solution in the form $u(x) = x^m$. When we substitute this solution into the equation we get:

$$\begin{aligned} am(m-1)x^2x^{m-2} + bmxx^{m-1} + cx^m &= 0 \\ x^m [am(m-1) + bm + c] &= 0 \end{aligned}$$

That last part in the brackets is referred to as the “auxiliary equation”:

$$am^2 + (b-a)m + c = 0 \quad (34)$$

We will look for values of m that satisfy this quadratic equation; those values will be the exponents for our solutions.

AS IS THE CASE for any other quadratic equation, there are three possible outcomes for the auxiliary equation:

1. **Distinct Real Roots.** In this case $m_1 \neq m_2$ and the general solution is of the form:

$$u(x) = c_1x^{m_1} + c_2x^{m_2} \quad (35)$$

Example: Find the general solution for $x^2 \frac{d^2u}{dx^2} - 2x \frac{du}{dx} - 4u = 0$.

Referring to Equation 34, $a = 1$, $b = -2$, $c = -4$ so the auxiliary equation is:

$$\begin{aligned} m^2 - 3m - 4 &= 0 \\ (m-4)(m+1) &= 0 \end{aligned}$$

By inspection the roots are $m_1 = 4$ and $m_2 = -1$. The general solution is $u(x) = c_1x^4 + c_2x^{-1}$.

If $u(x) = x^m$ then, of course, $u' = mx^{(m-1)}$ and $u'' = m(m-1)x^{m-2}$.

2. **Real Repeated Roots.** In this case, $m_1 = m_2$. We have one solution, $u_1(x) = c_1x^{m_1}$. Clearly we need to take some kind of action if we hope to get another linearly independent solution. It can be shown that if we form the second solution by multiplying the first solution by $\ln x$ —i.e. $u_2(x) = \ln(x)u_1(x)$ —then $u_2(x)$ will satisfy the equation and also be linearly independent from $u_1(x)$.

Example: Find the general solution for $4x^2 \frac{d^2u}{dx^2} + 8x \frac{du}{dx} + u = 0$.

The auxiliary equation in this case is: $4m^2 + 4m + 1 = 0$. This can be factored to give $(2m+1)^2 = 0$ so we have a case of repeated roots where $m_1 = m_2 = -\frac{1}{2}$.

The solution is: $u(x) = c_1x^{-1/2} + c_2x^{-1/2} \ln x$.

Be careful with these coefficients. In contrast to the case with constant coefficient linear equations, we do not plug these coefficients directly into the quadratic equation. Instead we put them in the auxiliary equation and then solve *that* with the quadratic equation.

The first one or two times you solve these problems, you should verify both of those assertions. Namely that:

- (a) $u_2(x) = \ln(x)u_1(x)$ is a solution to the equation; and
- (b) $u_2(x)$ is linearly independent from $u_1(x)$.

3. **Complex Conjugate Roots.** This case is analogous with the previous cases vis-à-vis linear constant coefficient equations. The roots are $m_{1,2} = \alpha \pm i\beta$ and the general solution is:

$$u(x) = x^\alpha [c_1 \cos(\beta \ln x) + c_2 \sin(\beta \ln x)] \quad (36)$$

Example: Solve: $4x^2u'' + 17u = 0$, $u(1) = -1$, $u'(1) = -1/2$.

The auxiliary equation is $4m^2 - 4m + 17 = 0$. Using the quadratic formula the roots are found to be:

$$\begin{aligned} m_{1,2} &= \frac{4 \pm \sqrt{16 - 4(4)(17)}}{8} \\ &= \frac{1}{2} \pm \frac{\sqrt{-256}}{8} \\ &= \frac{1}{2} \pm \frac{16i}{8} \\ &= \frac{1}{2} \pm 2i \\ &\quad \alpha \qquad \beta \end{aligned}$$

So the general solution is:

$$u(x) = x^{1/2} [c_1 \cos(2 \ln x) + c_2 \sin(2 \ln x)]$$

We can apply the first boundary condition, $u(1) = -1$:

$$\begin{aligned} u(1) &= 1 [c_1 \cos 0 + c_2 \sin 0] \\ &= c_1(1) + c_2(0) = -1 \\ \Rightarrow c_1 &= -1 \end{aligned}$$

The calculus is a bit more tedious for the second boundary condition:

$$\begin{aligned} u'(x) &= -\frac{1}{2}x^{-1/2} \cos(2 \ln x) + 2x^{-1/2} \sin(2 \ln x) + \dots \\ &\quad c_2 \left[\frac{1}{2}x^{-1/2} \sin(2 \ln x) + 2x^{-1/2} \cos(2 \ln x) \right] \end{aligned}$$

Evaluating this at $x = 1$:

$$\begin{aligned} u'(1) &= -\frac{1}{2}(1)(1) + 2(1)(0) + c_2[0 + 2(1)(1)] \\ &= -\frac{1}{2} + 2c_2 = -\frac{1}{2} \\ \Rightarrow c_2 &= 0 \end{aligned}$$

So the solution is: $u(x) = -x^{1/2} \cos(2 \ln x)$.

Non-homogeneous Cauchy-Euler Equations

Sadly, the method of undetermined coefficients will not work with Cauchy-Euler equations, a limitation of that method being that the coefficients need to be constant. Interested students can investigate a method called *variation of parameters* that can be used to address this problem analytically. Otherwise, we will plan to use numerical methods to solve non-homogeneous problems of this type.

Derivation of the Solution to Cauchy-Euler Equations

It would be hard not to notice the similarity in the solution methods of Cauchy-Euler equations and constant coefficient linear equations. This is not a coincidence. In this section I want to briefly show you that, through a change of variables, Cauchy-Euler equations are, in some sense, equivalent to constant coefficient linear equations.

Change of Independent Variable

What we will do, is change the independent variable from x to e^t .¹ If $x = e^t$, that means that $t = \ln x$ and $\frac{dt}{dx} = \frac{1}{x} = e^{-t}$.

If we consider, again, the 2nd-order Cauchy-Euler equation,

$$ax^2 \frac{d^2u}{dx^2} + bx \frac{du}{dx} + cu = 0$$

every appearance of x needs to be converted into its equivalent in terms of t and every derivative with respect to x needs to be converted into a derivative with respect to t .

It's easy enough to replace x with e^t ; converting the derivatives takes a bit more work. We will use the chain rule as shown below:

$$\begin{aligned}\frac{du}{dx} &= \frac{du}{dt} \frac{dt}{dx} \\ &= u_t e^{-t}\end{aligned}$$

where we use the subscript notation to denote derivatives with respect to t and use the substitution $\frac{dt}{dx} = e^{-t}$ as determined above.

We do it again, to convert the second derivatives:

$$\begin{aligned}\frac{d^2u}{dx^2} &= \frac{d}{dx} \left(\frac{du}{dx} \right) \\ &= \frac{d}{dt} \left(\frac{du}{dx} \right) \frac{dt}{dx} \\ &= \frac{d}{dt} (u_t e^{-t}) e^{-t} \\ &= (u_{tt} e^{-t} - u_t e^{-t}) e^{-t} \\ &= e^{-2t} (u_{tt} - u_t)\end{aligned}$$

¹ Think of this as "stretching" the x -axis.

We are now ready to make our substitutions into the differential equation:

$$\frac{a}{x^2} e^{2t} \left[\frac{\frac{d^2 u}{dx^2}}{e^{-2t}} \right] + b \frac{e^t}{x} \left[\frac{\frac{du}{dx}}{e^{-t}} \right] + cu = 0$$

Combining terms to simplify gives us Equation 37 which is now, under this change of variables, a 2nd-order linear constant coefficient equation.

$$au_{tt} + (b - a)u_t + cy = 0 \quad (37)$$

If we solve this using our standard method, the resulting auxiliary equation is the same as what is shown in Equation 34.

IN THE CASE of constant coefficient linear equations, the solutions were of the form $u = e^{mx}$ which, according to the exponentiation rules, the same as $u = e^{x^m}$. But now, our independent variable is t , where $t = \ln x$. With this substitution:

$$\begin{aligned} u(t) &= e^{(\ln x)^m} \\ &= x^m \end{aligned}$$

which is the assumed form of solution for Cauchy-Euler equations.

Part II

Power Series Methods

Lecture 7 - Review of Power Series

Objectives

The objectives of this lecture are:

- Review definitions and basic properties of power series.
- Illustrate important basic operations on power series.

Introduction and Review

The methods that we have discussed so far have largely been a review of your differential equations class. With this handful of methods we have found that we can solve a variety of simple problems. We can solve constant coefficient linear equations, and variable coefficient linear equations if they happen to be Cauchy-Euler equations. We can solve many first-order linear equations but if an equation is nonlinear we are sunk unless it happens to be separable. This leaves out a lot of interesting equations. In this sequence of lectures we will discuss how to solve linear equations with variable coefficients (other than Cauchy-Euler equations). To do this we will need to use power series.

YOU LEARNED ABOUT power series back in calculus class, but you weren't ready to use them for this important application. Now you are ready and now this is what we shall do. We will begin this section with some definitions that will be needed as we describe the use of power series in the solution of differential equations.

Definitions

Definition 3 (Sequence)

A sequence is a list of numbers (or other mathematical objects, like functions) written in a definite order.

$$\{c_0, c_1, c_2, c_3, \dots, c_n\}$$

Definition 4 (Limit of a Sequence, convergence, divergence)

A sequence has a limit (L) if we can make the terms c_n arbitrarily close to L by taking n sufficiently large. If $\lim_{n \rightarrow \infty} c_n$ exists, we say the sequence converges; otherwise, we say the sequence diverges or is divergent.

There are mathematical tools available for determining if an infinite sequence converges or diverges without needing to examine every element. We will discuss one of them as part of our treatment of infinite series.

Definition 5 (Series, infinite series)

A series is the sum of a sequence. For example, $S_0 = c_0$; $S_1 = c_0 + c_1$; $S_n = c_0 + c_1 + \dots + c_n$. If the sequence is infinite, we call the sum an infinite series.

Definition 6 (Series Convergence)

Given a series $\sum_{n=0}^{\infty} s_i = s_1 + s_2 + \dots + s_n + \dots$, let s_n denote its n^{th} partial sum. If the sequence $\{s_n\}$ is convergent then the series is convergent to the same limit. Otherwise the series is divergent.

We will use the notation: $s_n \rightarrow \infty$ to indicate that a partial sum is unbounded.

Definition 7 (Power Series)

A series of the form $\sum_{n=0}^{\infty} c_n(x - a)^n = c_0 + c_1(x - a) + \dots$ is called a power series. The constant, a , is referred to as the center of the power series.

For almost all of the power series we will work with in this class, the series will be centered on $a = 0$ and will be denoted $\sum_{n=0}^{\infty} c_n x^n$.

Definition 8 (Interval of Convergence, Radius of Convergence)

The interval of convergence is the set of all real numbers x for which the series converges. This interval can also be expressed as a radius of convergence (R). The series converges for all $(a - R) < x < (a + R)$.

Ratio Test

We should have at least one test that we can use to decide whether or not a series, or at least a power series, converges. The test we will use is called the *ratio test*; so named because it involves the ratio of the n^{th} and $(n + 1)^{\text{th}}$ term in a power series. The ratio test is shown in Equation 38.

$$\lim_{n \rightarrow \infty} \left| \frac{c_{n+1}(x - a)^{(n+1)}}{c_n(x - a)^n} \right| = |x - a| \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}}{c_n} \right| = L \quad (38)$$

The following cases are considered:

- If $L < 1$ then the series converges absolutely.
- If $L = 1$ then the test is inconclusive; some other test must be used; and
- if $L > 1$ then the series diverges.

Note: *Absolute convergence* means that the series converges irrespective of the signs of each term—i.e. whether or not all terms are positive, negative, or a mix of both positive and negative.

Example: Find the radius of convergence and associated interval of convergence for the following power series:

$$1. \sum_{n=0}^{\infty} (-1)^n x^n$$

$$\begin{aligned} \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}(x-a)^{n+1}}{c_n(x-a)^n} \right| &= |x-a| \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}}{c_n} \right| = L < 1, \quad a = 0, \quad c_n = (-1)^n \\ \lim_{n \rightarrow \infty} \left| \frac{x^{n+1}}{x^n} \right| &< 1 \\ |x| \lim_{n \rightarrow \infty} |1| &< 1 \\ \Rightarrow |x| &< 1 \end{aligned}$$

The radius of convergence $R = 1$ and the interval of convergence is $x \in (-1, 1)$.

Here I have purposely avoided analyzing the end-points to see if we could use a closed or partially-closed interval instead. Since we specified $L < 1$, we only have the radius of absolute convergence. If we wanted to be picky, we could allow $L = 1$ and use some other test to determine if the series converges. If we did that in this case we would find that the series diverges at both endpoints.

$$2. \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n(n+1)}$$

$$\begin{aligned} \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}(x-a)^{n+1}}{c_n(x-a)^n} \right| &= |x-a| \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}}{c_n} \right| = L < 1, \quad a = 0, \quad c_n = \frac{(-1)^n}{n(n+1)} \\ \lim_{n \rightarrow \infty} \left| \frac{x^{n+1}}{(n+1)(n+1+1)} \frac{n(n+1)}{x^n} \right| &< 1 \\ |x| \lim_{n \rightarrow \infty} \left| \frac{n}{n+2} \right| &< 1 \\ |x| \lim_{n \rightarrow \infty} \left| \frac{n}{n+2} \right| &\xrightarrow{1} 1 \\ |x| &< 1 \end{aligned}$$

Once again, the radius of convergence $R = 1$ and the interval of convergence is $x \in (-1, 1)$.

$$3. \sum_{n=1}^{\infty} \frac{x^{2n}}{2^n n^2}$$

$$\begin{aligned}\lim_{n \rightarrow \infty} \left| \frac{c_{n+1}(x-a)^{n+1}}{c_n(x-a)^n} \right| &= |x-a| \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}}{c_n} \right| = L < 1, \quad a = 0, \quad c_n = \frac{1}{2^n n^2} \\ \lim_{n \rightarrow \infty} \left| \frac{x^{2n+2}}{2^{n+1}(n+1)^2} \frac{2^n n^2}{x^{2n}} \right| &< 1 \\ \lim_{n \rightarrow \infty} \left| \frac{x^2}{2} \frac{n^2}{(n+1)^2} \right| &< 1 \\ \frac{|x^2|}{2} \lim_{n \rightarrow \infty} \left| \frac{n^2}{(n+1)^2} \right| &\xrightarrow{1} 1 \\ |x^2| &< 2 \\ |x| &< \sqrt{2}\end{aligned}$$

In this case the radius of convergence $R = \sqrt{2}$ and the interval of convergence is $x \in (-\sqrt{2}, \sqrt{2})$.

In this case, more detailed analysis shows that this series converges at both endpoints so a closed interval could be used instead.

$$4. \sum_{n=1}^{\infty} \frac{(x-2)^n}{3^n}$$

$$\begin{aligned}\lim_{n \rightarrow \infty} \left| \frac{c_{n+1}(x-a)^{n+1}}{c_n(x-a)^n} \right| &= |x-a| \lim_{n \rightarrow \infty} \left| \frac{c_{n+1}}{c_n} \right| = L < 1, \quad a = 2, \quad c_n = \frac{1}{3^n} \\ \lim_{n \rightarrow \infty} \left| \frac{(x-2)^{n+1}}{3^{n+1}} \frac{3^n}{(x-2)^n} \right| &= L < 1 \\ \frac{|x-2|}{3} \lim_{n \rightarrow \infty} |1| &< 1 \\ |x-2| &< 3\end{aligned}$$

We see that for this example the radius of convergence is $R = 3$ centered at $x = 2$; the interval of convergence is $x \in (-1, 5)$.

Properties of Convergent Series

Within the radius of convergence, a power series defines a function and the function so defined is:

- continuous
- differentiable (term-by-term); and
- integrable (term-by-term).

For the interested reader, it can be shown that this series is divergent at both endpoints so it should remain an open interval.

If x is not within the interval of convergence for a series or if the series is divergent then *none* of these properties hold true. This is why it is important to be able to find the interval/radius of convergence.

Definition 9 (Identity Property for a Power Series)

If $\sum_{n=0}^{\infty} c_n(x-a)^n = 0$, $R > 0$, for all numbers x in the interval of convergence then $c_n = 0$ for all n .

Definition 10 (Analytic Function)

A function f is analytic at a point, a_0 , if it can be represented by a power series in $x - a_0$ with a positive radius of convergence.

Some Common Power Series

You have probably had some exposure to power series in your previous mathematical courses. As a reminder, the power series representations of some important/common functions are shown below:

1. $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$
2. $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$
3. $\ln x = \frac{x-1}{x} + \frac{(x-1)^2}{2x^2} + \frac{(x-1)^3}{3x^3} + \dots$
4. $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots$

Combining Power Series

This is a practical “utility skill” that you will need to master to be successful during this portion of the course. What we need to be able to do is combine multiple power series into a single power series.

FOR EXAMPLE, consider the two power series below that we want to write as a single power series:

$$\sum_{n=2}^{\infty} n(n-1)c_n x^{n-2} - \sum_{n=0}^{\infty} c_n x^{n+1}$$

If I want to combine these series, I need to overcome two issues:

1. The powers of x in each term in both summations need to be “in phase”—that is the corresponding terms need to have the same power of x . The first term in the first summation is constant (x^0) while the first term in the second summation is linear (x^1).
2. The first summation index starts at $n = 2$ while the second summation index starts at $n = 0$.

WE WILL ADDRESS these issues one at a time, starting with the first one. We will leave the summation whose first term is higher order

Hopefully this definition seems obvious to you. You will find that most of what we do when using power series to solve homogeneous linear differential equations is carry out the necessary algebra to ensure that the coefficients for some series all are equal to zero.

This is just a vocabulary term that you should know. It will come up again later when distinguishing between ordinary points and singular points of a differential equation, and between regular and irregular singular points of a singular differential equation.

It is not only the first term that is important but if you can get the summations in phase for the first term, and if the power of x increases by one with each consecutive term, then if the first term is correct, they will all be correct.

as-is; for all other summations—i.e. if there are more than two—we will “peel-off” any lower-order summation terms.

In this case that means we will leave the second series as-is, since its first term is of higher order (x^1 compared to x^0), while we will “peel-off” the constant term from the first summation:

$$(2)(1)c_2x^0 + \underbrace{\sum_{n=3}^{\infty} n(n-1)c_nx^{n-2}}_{\text{constant term now } n=3} + \underbrace{\sum_{n=0}^{\infty} c_nx^{n+1}}_{\text{Same as before}}$$

Notice that the summation index for the first summation now starts at $n = 3$; this is because we’ve separated out the first term corresponding to $n = 2$. The two remaining summations are “in phase” since all of the terms now have the same power of x .

THE SECOND PROBLEM will be fixed by establishing a new common index, k , and re-write the existing indices (n for both summations) in terms of k . In each case we will set k equal to the exponent of x appearing in the summation.

- For the first summation— $\sum_{n=3}^{\infty} n(n-1)c_nx^{n-2}$ —we set $k = n - 2$ because that is the exponent for x . We need to eliminate each occurrence of n in the summation and replace it with its equivalent expression in terms of k . From our definition of k for this summation, $n = k + 2$. Our summation now can be written:

$$\sum_{n=3}^{\infty} n(n-1)c_nx^{n-2} \xrightarrow{n=k+2} \sum_{k=1}^{\infty} (k+2)(k+2-1)c_{k+2}x^k$$

- For the second summation— $\sum_{n=0}^{\infty} c_nx^{n+1}$ —we set $k = n + 1$ because that is the exponent for x in this summation. This means $n = k - 1$; substituting that expression in our summation gives us:

$$\sum_{n=0}^{\infty} c_nx^{n+1} \xrightarrow{n=k-1} \sum_{k=1}^{\infty} c_{k-1}x^k$$

With these changes our original summation can be written:

$$2c_2 + \sum_{k=1}^{\infty} (k+2)(k+1)c_{k+2}x^k + \sum_{k=1}^{\infty} c_{k-1}x^k \quad (39)$$

The two summations are now ready to be joined into one as shown in Equation 40:

$$2c_2 + \sum_{k=1}^{\infty} [(k+2)(k+1)c_{k+2} + c_{k-1}] x^k \quad (40)$$

Everywhere you see an n in the original summation, replace it with a $k + 2$ and simplify.

Notice that I’ve made some obvious simplifications in the constant term and first summation.

Lecture 8 - Power Series Solutions at Ordinary Points

Objectives

The objectives of this lecture are:

- Introduce some definitions and concepts relevant for power series solutions of differential equations.
- Do some example problems.

Introduction

In this section we will restrict our attention to second-order, linear, homogeneous differential equations in standard form as shown in Equation 41.

$$u'' + P(x)u' + Q(x)u = 0 \quad (41)$$

Definition 11 (Ordinary Points and Singular Points)

A point x_0 is said to be an ordinary point of a differential equation if both $P(x)$ and $Q(x)$ in the standard form are analytic at x_0 . A point that is not an ordinary point is a singular point.

Theorem 6 (Existence of Power Series Solutions)

If $x = x_0$ is an ordinary point of the differential equation, we can always find two linearly independent solutions in the form of a power series centered at x_0 . A series solution converges at least on some interval defined by $|x - x_0| < R$ where R is the distance from x_0 to the closest singular point.

Let us emphasize: this theorem applies only to second-order, linear, homogeneous differential equations.

The basic strategy we will use to find power series solutions for linear differential equations with variable coefficients where $P(x)$ and $Q(x)$ are analytic in the domain of interest is:

1. Find solutions in the form of a power series by substituting $u = \sum_{n=0}^{\infty} c_n x^n$ into the differential equation.
2. Solve for the values of the coefficients by equating the coefficients on the left of Equation 41 with those on the right (e.g. zero for homogeneous equations); and

3. The equations (often 2- or 3-term recurrence relations) for the series coefficients *defines* the function that is the solution of the differential equation.

Examples

Example #1: Solve $u'' + u = 0$ using the power series method; compare with the known solution $u(x) = c_0 \cos x + c_1 \sin x$.

In accordance with our strategy, we will assume that the solution is of the form: $u(x) = \sum_{n=0}^{\infty} c_n x^n$. This means that $u' = \sum_{n=1}^{\infty} n c_n x^{n-1}$ and $u'' = \sum_{n=2}^{\infty} n(n-1) c_n x^{n-2}$. Plugging this into our differential equation gives us:

$$\begin{aligned} u'' + u &= 0 \\ \sum_{n=2}^{\infty} n(n-1) c_n x^{n-2} + \sum_{n=0}^{\infty} c_n x^n &= 0 \end{aligned}$$

We need to combine the two summations. It is clear that the summation indexes start at different values but we are lucky in that the summations are already “in phase” since the first term in each summation is a constant (x^0) term.

$$\underbrace{\sum_{n=2}^{\infty} n(n-1) c_n x^{n-2}}_{\substack{k=n-2 \\ n=k+2}} + \underbrace{\sum_{n=0}^{\infty} c_n x^n}_{\substack{k=n \\ n=k}} = 0$$

For the first summation $k = n - 2$, so $n = k + 2$; we will use these definitions to re-write the first summation. For the second summation, $k = n$ so all we need to do for the second summation is replace all the n 's with k 's. The results of these substitutions and the combined summation are shown below:

$$\begin{aligned} \sum_{k=0}^{\infty} (k+2)(k+1)c_{k+2}x^k + \sum_{k=0}^{\infty} c_k x^k &= 0 \\ \sum_{k=0}^{\infty} \underbrace{[(k+2)(k+1)c_{k+2} + c_k]}_{\text{coefficients for new power series}} x^k &= 0 \end{aligned}$$

THE EXPRESSION, $[(k+2)(k+1)c_{k+2} + c_k]$, is now a formula for the coefficients of a new power series. This power series, according to the equation, is equal to zero so that means, per Definition 9, all of the coefficients must be equal to zero:

$$(k+2)(k+1)c_{k+2} + c_k = 0, \quad \text{for all } k \in [0, 2, 3, \dots]$$

Important: The series “solution” is only valid if the series so-derived has a non-zero radius of convergence.

It is not *required* that a problem have variable coefficients in order to use the Power Series method; only that $P(x)$ and $Q(x)$ are analytic on the domain of interest. Constants are always analytic over the entire real number line so you can always use the Power Series method on linear, constant-coefficient differential equations.

Note that the $n = 0$ term in u' is omitted as is the $n = 0$ and $n = 1$ term in u'' . These terms are zero due to having taken the first- and second-derivative on the constant (x^0) and linear (x^1) terms of the power series.

This is called a *two-term recurrence* since the expression involves two terms; c_{k-2} and c_k .

By convention, we will re-write this recurrence relation to solve for the *higher*-index coefficients in terms of the *lower*-index coefficients.

We do this in Equation 42:

$$c_{k+2} = -\frac{c_k}{(k+2)(k+1)} \quad (42)$$

As we **should** expect, there are two unknown constants in this general solution— c_0 and c_1 . The first value of k from the summation in our solution is $k = 0$ which gives us an expression for c_2 in terms of c_0 . The second value, $k = 1$, will give us an expression for c_3 in terms of c_1 . Simplified equations for the first few coefficients are presented in the table below.

$k = 0$	$k = 2$	$k = 4$
$c_2 = \frac{-c_0}{(1)(2)}$	$c_4 = \frac{-c_2}{(3)(4)} = \frac{c_0}{4!}$	$c_6 = \frac{-c_4}{(5)(6)} = \frac{-c_0}{6!}$
$k = 1$	$k = 3$	$k = 5$
$c_3 = \frac{-c_1}{(2)(3)}$	$c_5 = \frac{-c_3}{(4)(5)} = \frac{c_1}{5!}$	$c_7 = \frac{-c_5}{(6)(7)} = \frac{-c_1}{7!}$

Each cell in the table above is a formula for the k^{th} -coefficient of our power series solution. Organizing this into a formula for our power series solution gives us:

$$u(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5 + c_6 x^6 + c_7 x^7 + \dots$$

$$\begin{aligned} u(x) &= c_0 \left(1 + \frac{c_2}{c_0} x^2 + \frac{c_4}{c_0} x^4 + \frac{c_6}{c_0} x^6 + \dots \right) + c_1 \left(x + \frac{c_3}{c_1} x^3 + \frac{c_5}{c_1} x^5 + \frac{c_7}{c_1} x^7 + \dots \right) \\ u(x) &= c_0 \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \right) + c_1 \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \right) \end{aligned}$$

where in the last line we have substituted the formulas for coefficients c_2 through c_7 in terms of c_0 and c_1 .

Recall from the Lecture 7 the power series representations of $\cos x$ and $\sin x$; we should be able to see them again here.

$$u(x) = c_0 \underbrace{\left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \right)}_{\cos x} + c_1 \underbrace{\left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \right)}_{\sin x}$$

$$u(x) = c_0 \cos x + c_1 \sin x$$

Which is exactly what we would have determined using our methods for constant coefficient linear equations.

We expect the general solution for any second order differential equation to have two unknown constants that can only be resolved by adding initial- or boundary-conditions.

Notice how the even-numbered coefficients are all dependent on c_0 and all of the odd-numbered coefficients are dependent on c_1 .

In general you are not expected to, nor will you be able to, identify common functions from a power series solution. This is a special case.

Example #2: Find the general solution to $u'' - xu = 0$.

Notice first that while this equation is linear and homogeneous it is not constant-coefficient. It is also not a Cauchy-Euler equation. We will use the power series method to solve this problem. Assuming $u = \sum_{n=0}^{\infty} c_n x^n$ and inserting this into the governing equation gives us:

$$\begin{aligned} u'' - xu &= 0 \\ \sum_{n=2}^{\infty} n(n-1)c_n x^{n-2} - x \sum_{n=0}^{\infty} c_n x^n &= 0 \\ \sum_{n=2}^{\infty} n(n-1)c_n x^{n-2} - \sum_{n=0}^{\infty} c_n x^{n+1} &= 0 \end{aligned}$$

The equation is separable but, in this case, the solution is not so easy to obtain using that method either.

We want to combine these summations but we see that they are both “out of phase” and the summation index, n , starts at different values for each summation.

$$\underbrace{\sum_{n=2}^{\infty} n(n-1)c_n x^{n-2}}_{\text{for } n=2, \ x^0} - \underbrace{\sum_{n=0}^{\infty} c_n x^{n+1}}_{\text{for } n=0, \ x^1} = 0$$

So we must separate out the first term in the first summation to get the summations in phase.

$$(2)(1)c_2 x^0 + \sum_{n=3}^{\infty} n(n-1)c_n x^{n-2} - \sum_{n=0}^{\infty} c_n x^{n+1} = 0$$

Next we must combine our indices using $k = n - 2$ for the first summation and $k = n + 1$ for the second summation.

$$2c_2 x^0 + \underbrace{\sum_{n=3}^{\infty} n(n-1)c_n x^{n-2}}_{\substack{k=n-2 \\ n=k+2}} - \underbrace{\sum_{n=0}^{\infty} c_n x^{n+1}}_{\substack{k=n+1 \\ n=k-1}} = 0$$

Reminder: We take our definition of k from the exponent for x in each summation term.

Doing this gives us:

$$2c_2 + \sum_{k=1}^{\infty} [(k+2)(k+1)c_{k+2} - c_{k-1}] x^k = 0$$

Do not forget the minus sign in front of the second summation. It is easy to miss.

In order to solve the differential equation, the coefficient for every power of x needs to be zero. To do this:

$$\Rightarrow c_2 = 0 + \sum_{k=1}^{\infty} \underbrace{[(k+2)(k+1)c_{k+2} - c_{k-1}]}_{\text{must equal zero}} x^k = 0$$

Our corresponding two-term recurrence relation is:

$$c_{k+2} = \frac{c_{k-1}}{(k+2)(k+1)}$$

We expect two arbitrary constants, c_0 and c_1 and we know from the work above that $c_2 = 0$ so we will start solving for constants starting with $k = 1$:

$k = 1$	$k = 4$
$c_3 = \frac{c_0}{(3)(2)}$	$c_6 = \frac{c_3}{(6)(5)} = \frac{c_0}{(2)(3)(5)(6)}$
$k = 2$	$k = 5$
$c_4 = \frac{c_1}{(3)(4)}$	$c_7 = \frac{c_4}{(6)(7)} = \frac{c_1}{(3)(4)(6)(7)}$
$k = 3$	$k = 6$
$c_5 = \frac{c_2}{(5)(4)} = 0$	$c_8 = \frac{c_5}{(8)(7)} = 0$
$k = 7$	$k = 8$
$c_9 = \frac{c_6}{(9)(8)} = \frac{c_0}{(2)(3)(5)(6)(8)(9)}$	$c_{10} = \frac{c_7}{(10)(9)} = \frac{c_1}{(3)(4)(6)(7)(9)(10)}$

Organizing the coefficients from the table into an equation we get:

$$u(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5 + c_6 x^6 + c_7 x^7 + c_8 x^8 + c_9 x^9 + c_{10} x^{10} + \dots$$

$$u(x) = c_0 \left(1 + \frac{c_3}{c_0} x^3 + \frac{c_6}{c_0} x^6 + \frac{c_9}{c_0} x^9 + \dots \right) + c_1 \left(x + \frac{c_4}{c_1} x^4 + \frac{c_7}{c_1} x^7 + \frac{c_{10}}{c_1} x^{10} + \dots \right)$$

which can be written:

$$u(x) = c_0 \left(1 + \frac{x^3}{(2)(3)} + \frac{x^6}{(2)(3)(5)(6)} + \frac{x^9}{(2)(3)(5)(6)(8)(9)} + \dots \right) + \\ c_1 \left(x + \frac{x^4}{(3)(4)} + \frac{x^7}{(3)(4)(6)(7)} + \frac{x^{10}}{(3)(4)(6)(7)(9)(10)} + \dots \right)$$

The equation we solved is known as Airy's Equation. The power series solution is not pretty, but is a perfectly adequate representation of the function provided that we have the wherewithal to evaluate the function for a reasonable number of terms.

Reminder: We should define our recurrence relation to give higher-order coefficients in terms of lower-order coefficients.

Assignment #3

Find the general solution to the following differential equations.

1. $x^2u'' - 2u = 0$

2. $xu'' + u' = 0$

3. $x^2u'' - 3xu' - 2u = 0$

Find the solution to the given initial value problem.

4. $x^2u'' + 3xu' = 0, \quad u(1) = 0, \quad u'(1) = 4$

Use MATLAB to plot the solution for $x \in [1, 10]$.

5. A very long cylindrical shell is formed by two concentric circular cylinders of different radii. A chemically reactive fluid fills the space between the concentric cylinders. The inner cylinder has a radius of 1 and is thermally insulated, while the outer cylinder has a radius of 2 and is maintained at a constant temperature T_0 . The rate of heat generation in the fluid due to the chemical reaction is proportional to T/r^2 , where $T(r)$ is the temperature of the fluid within the space bounded between the cylinders defined by $1 < r < 2$. Under these conditions the temperature of the fluid is defined by the following boundary value problem:

$$\frac{d^2T}{dr^2} + \frac{1}{r} \frac{dT}{dr} = -\frac{1}{r}T, \quad 1 < r < 2,$$

$$\left. \frac{dT}{dr} \right|_{r=1} = 0, \quad T(2) = T_0$$

Solve the boundary value problem to find the temperature of the fluid within the cylindrical shell.

For the following problems, find the radius and interval of convergence. For the intervals of convergence, you do not need to check the endpoints (unless you want to!).

$$6. \sum_{n=1}^{\infty} \frac{2^n}{n} x^n$$

$$7. \sum_{n=1}^{\infty} \frac{(-1)^n}{10^n} (x - 5)^n$$

Rewrite the given expression as a single power series whose general term involves x^k .

$$8. \sum_{n=1}^{\infty} 2nc_n x^{n-1} + \sum_{n=0}^{\infty} 6c_n x^{n+1}$$

Find two power series solutions of the given differential equation.

$$9. u'' - 2xu' + u = 0$$

Lecture 9 - Power Series Solutions with MATLAB

Objectives

The objectives of this lecture are:

- Illustrate the solution of a linear IVP (with a 3-term recurrence) using power series.
- Demonstrate a way to analyze these solutions using MATLAB; and
- demonstrate some expected elements of MATLAB style for this course.

Solution of an IVP using Power Series

Consider the following IVP:

$$\text{Governing Equation: } u'' - (1+x)u = 0, \quad u \in [0, 5]$$

$$\text{Initial Conditions: } u(0) = 5, \quad u'(0) = 1$$

Inserting our assumed power series solution into the governing equation gives us:

$$\begin{aligned} \sum_{n=2}^{\infty} n(n-1)c_nx^{n-2} - (1+x) \sum_{n=0}^{\infty} c_nx^n &= 0 \\ \sum_{n=2}^{\infty} n(n-1)c_nx^{n-2} - \sum_{n=0}^{\infty} c_nx^n - \sum_{n=0}^{\infty} c_nx^{n+1} &= 0 \end{aligned}$$

We need to evaluate the order of x for the first term in each summation to determine if the summations are in phase:

$$\underbrace{\sum_{n=2}^{\infty} n(n-1)c_nx^{n-2}}_{x^0} - \underbrace{\sum_{n=0}^{\infty} c_nx^n}_{x^0} - \underbrace{\sum_{n=0}^{\infty} c_nx^{n+1}}_{x^1} = 0$$

To get the three summations in phase we need to strip off the first terms in the first and second summations so that all three summations start at x^1 . This gives us:

$$2c_2 + \sum_{n=3}^{\infty} n(n-1)c_nx^{n-2} - c_0 - \sum_{n=1}^{\infty} c_nx^n - \sum_{n=0}^{\infty} c_nx^{n+1} = 0$$

As before, we will assume $u = \sum_{n=0}^{\infty} c_nx^n$.

This means that $u' = \sum_{n=1}^{\infty} nc_nx^{n-1}$, and
 $u'' = \sum_{n=2}^{\infty} n(n-1)c_nx^{n-2}$.

Note the effect of distributing $-(1+x)$ through the second summation.

$$2c_2 - c_0 + \underbrace{\sum_{n=3}^{\infty} n(n-1)c_n x^{n-2}}_{\substack{k=n-2 \\ n=k+2}} - \underbrace{\sum_{n=1}^{\infty} c_n x^n}_{\substack{k=n \\ n=k}} - \underbrace{\sum_{n=0}^{\infty} c_n x^{n+1}}_{\substack{k=n+1 \\ n=k-1}} = 0$$

Substituting within each summation and combining terms gives us:

$$(2c_2 - c_0)x^0 + \sum_{k=1}^{\infty} [(k+2)(k+1)c_{k+2} - c_k - c_{k-1}] x^k = 0$$

As usual, in order to satisfy this equation, the coefficients for each power of x must be equal to zero. For x^0 this means $2c_2 - c_0 = 0$. For all the other powers of x , a *three-term recurrence* involving c_{k-1} , c_k , and c_{k+2} must be satisfied:

$$c_{k+2} = \frac{c_k + c_{k-1}}{(k+2)(k+1)}$$

To help manage the complexity we will adopt the following strategy:

- Case 1: Arbitrarily set $c_0 \neq 0$, set $c_1 = 0$ and derive a solution.
- Case 2: Arbitrarily set $c_0 = 0$, set $c_1 \neq 0$ and derive a second solution.

These two solutions will be linearly independent since the first will not have a linear term (proportional to x) and the second solution will not have a constant term (proportional to 1).

Case 1: $c_0 \neq 0, c_1 = 0$

Since $c_0 \neq 0$, we get $c_2 = \frac{c_0}{2}$. The coefficients derived for the first few values of k are shown in the table to the right.

The solution we thus derive is shown below:

$$\begin{aligned} u_1 &= c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5 + \dots \\ u_1 &= c_0 \left(1 + \frac{c_1^0}{c_0} x + \frac{c_2}{c_0} x^2 + \frac{c_3}{c_0} x^3 + \frac{c_4}{c_0} x^4 + \frac{c_5}{c_0} x^5 + \dots \right) \\ u_1 &= c_0 \left(1 + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{30} x^5 + \dots \right) \end{aligned}$$

Case 1:

$$\begin{array}{l|l} k=1 & k=2 \\ c_3 = \frac{c_0+c_1^0}{(2)(3)} = \frac{c_0}{6} & c_4 = \frac{c_1^0+c_2}{(3)(4)} = \frac{c_0/2}{12} = \frac{c_0}{24} \\ \hline k=3 & \\ c_5 = \frac{c_2+c_3}{(4)(5)} = \frac{c_0/2+c_0/6}{20} = \frac{c_0}{30} & \end{array}$$

Case 2: $c_0 = 0, c_1 \neq 0$

Since $c_1 \neq 0$ and $c_2 = \frac{c_1}{6}$, $c_2 = 0$. The coefficients derived for the first few values of k are shown in the table.

The solution we thus derive is shown below:

$$\begin{aligned} u_2 &= c_0^0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5 + \dots \\ u_2 &= c_1 \left(x + \frac{c_2^0}{c_1} x^2 + \frac{c_3}{c_1} x^3 + \frac{c_4}{c_1} x^4 + \frac{c_5}{c_1} x^5 \dots \right) \\ u_2 &= c_1 \left(x + \frac{1}{6} x^3 + \frac{1}{12} x^4 + \frac{1}{120} x^5 + \dots \right) \end{aligned}$$

Case 2:

$$\begin{array}{l|l} k=1 & k=2 \\ c_3 = \frac{c_0^0+c_1}{(2)(3)} = \frac{c_1}{6} & c_4 = \frac{c_1^0+c_2}{(3)(4)} = \frac{c_1}{12} \\ \hline k=3 & \\ c_5 = \frac{c_2^0+c_3}{(4)(5)} = \frac{c_1/6}{20} = \frac{c_1}{120} & \end{array}$$

We now have two linearly independent solutions to the governing equation:

$$u(x) = u_1(x) + u_2(x) = c_0 \left(1 + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{30}x^5 + \dots \right) + \\ c_1 \left(x + \frac{1}{6}x^3 + \frac{1}{12}x^4 + \frac{1}{120}x^5 + \dots \right)$$

We are now ready to apply the initial conditions:

$$u(0) = c_0 = 5 \\ u'(0) = c_1 = 1$$

So the final solution is:

$$u(x) = 5 \left(1 + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{30}x^5 + \dots \right) + \\ \left(x + \frac{1}{6}x^3 + \frac{1}{12}x^4 + \frac{1}{120}x^5 + \dots \right)$$

Generating and Plotting Solutions with MATLAB

Most of the problems that we will solve in this class are derived from physical conservation laws and describe systems of interest to engineers. We solve the equations so we can get insight into how these systems will perform.

THERE ARE TWO PROBLEMS that I hope to address with this section:

1. It is both tedious and error-prone to generate the coefficients for the series solution. We worked hard to construct a small portion of the power series solutions and we hope that we did it without any errors. With a modest amount of programming effort, we will write a script that can build a series solution with as many terms as we would like. With conscientious debugging we can be sure that, floating point round-off errors aside, the calculations are done correctly and without undue tedium.
2. We may get more insight if we can visualize the solution; usually via a simple plot. If the plot can easily be made with the same computing tools used to generate the solution, we can get this insight with very little extra work.

We will use MATLAB to generate and plot the solutions. We start our MATLAB script in the same way we will start *all* of our MATLAB scripts, with the following three lines:

```
clear
clc
close 'all'
```

This is done in accordance with MATLAB Style Rule #1. The MATLAB Style Rules are listed in the Appendices and I will try to exemplify the rules in the code I provide as examples in the lectures.

Next we will specify the number of terms that we will retain from the infinite series and allocate arrays in which the coefficients will be stored.

```
n=25;
C1 = nan(1,n); % coefficients for u1(x)
C2 = nan(1,n); % coefficients for u2(x)
```

Recall that for $u_1(x)$ we applied our strategy for “case 1” in which we assumed that $c_0 \neq 0$ and $c_1 = 0$. This implied that $c_2 = c_1/2$ while the recurrence relation was used for all of the other coefficients. When we applied the initial conditions we found that $c_0 = 5$ for $u_1(x)$. The code snippet below accomplishes these tasks.

```
C1_0 = 5; % c_0 for u1(x), handled separately since MATLAB array
% indices start at 1
C1(1) = 0; % c_1 for u1(x)
C1(2) = C1_0/2; % c_2 for u1(x)

% handle the k=1 case separately since it involves the term C1_0
k = 1;
C1(k+2) = (C1(k) + C1_0)/((k+1)*(k+2));

for k=2:(n-2)
    C1(k+2) = (C1(k) + C1(k-1))/((k+1)*(k+2));
end
```

Now we have calculated all of the desired coefficients, we are ready to construct the first solution.

```
u1 = @(x) C1_0;
for k = 1:n
    u1 = @(x) u1(x) + C1(k)*x.^k;
end
```

We continue in this same vein to construct $u_2(x)$ as we did in our “case 2” strategy and construct the solution $u(x) = u_1(x) + u_2(x)$.

```
C2_0 = 0; % c_0 for u2(x)
C2(1) = 1; % from the initial condition
C2(2) = C2_0/2; % just adding for consistency's sake

k=1;
C2(k+2) = (C2(k) + C2_0)/((k+1)*(k+2));
for k = 2:(n-2)
    C2(k+2) = (C2(k) + C2(k-1))/((k+1)*(k+2));
end

u2 = @(x) C2_0;
for k = 1:n
    u2 = @(x) u2(x) + C2(k)*x.^k;
end

u = @(x) u1(x) + u2(x);
```

We will use two arrays to store the coefficients; one for $u_1(x)$ and the other for $u_2(x)$. Pre-allocation in this way is done in accordance with MATLAB Style Rule #7, and the comments are in accordance with Rule #2. Use of short but meaningful variable names is prescribed in Rule #3.

Note how each line of MATLAB in this listing is terminated with a semicolon. This is to suppress the output to the command line that would otherwise happen each time we make an assignment to a variable. While this output might be helpful during debugging, during any other time it is distracting (drowning out other, more useful output) and slows code execution. The extensive, non-readable output that would result from omitting the semi-colons is indicative of a violation of MATLAB Style Rule #4.

Here $u_1(x)$ is created as an *anonymous* function. We begin with the constant term on line 28 and build up the function term-by-term within the *for* loop on line 30. The indentation you see in these code blocks has been done automatically by the “smart indentation” feature of MATLAB’s built-in editor. This is retained in accordance with MATLAB Style Rule #6.

NOW THAT WE have a MATLAB representation of the solution, let us create a plot. One way to make such a plot is shown in the listing below; the output is shown in Figure 2.:

```

xMin = 0; xMax = 5;
figure(1)
fplot(u,[xMin, xMax], 'linewidth',2);
title_str = sprintf('Lecture 9 Series n = %d',n);
title(title_str , 'fontsize',18,'fontweight','bold');
xlabel('X', 'fontsize',16,'fontweight','bold');
ylabel('U(X)', 'fontsize',16,'fontweight','bold');
set(gca, 'fontsize',12,'fontweight','bold');
grid on

```

A FEW MORE details are worth noting:

1. We know our solution is inexact since we truncated the infinite power series and used finite-precision arithmetic while calculating the coefficients for those terms we *did* bother to include. Still, we might want to know *how wrong* the solution is.
2. Taking a more positive tack, we might ask how much *better* the solution gets when we add more terms to our solution.

To answer either of these questions, we will need access to the solution of the IVP. For this lecture, we will take a numeric solution generated using MATLAB's built-in IVP-solving tool *ODE45* as "the solution."

Power series results for various values of n are compared to the numerical solution in Figure 3. Some things to notice:

1. The solution gets worse the further one gets from zero; and
2. The solution gets better for larger values of n .

Neither of these observations should be surprising but there is value to seeing it in your results. It adds confidence to the proposition that your (approximate) solution is correct.

AS A LAST NOTE it should be pointed out that, while plots like that shown in Figure 3 gives a good qualitative feel for how the solution is improving as the number of power series terms increases, a quantitative measure for correctness is preferable. In Figure 4 a quantitative measure—the relative error in the 2-norm—is used to quantify the difference between different power series solutions and the solution generated using *ODE45*. Details of this error measure will be discussed in future lectures.

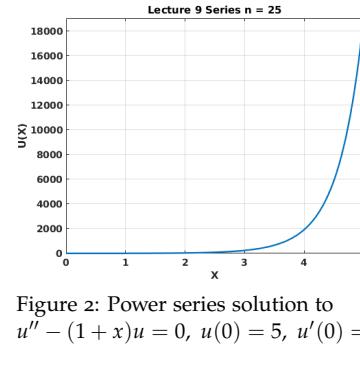


Figure 2: Power series solution to $u'' - (1+x)u = 0$, $u(0) = 5$, $u'(0) = 1$.

Note: Pay particular attention to the formatting details of the plot. There is a title along with axis-labels for both the x- and y-axis; fonts are bold and sized in a systematic way. Grid lines are also used to make the graph more readable. These details have all been added in observance of MATLAB Style Rule #4.

The variables *xMin* and *xMax* on line 39 has been included in accordance with Rule #8. The fact that both variables are initialized on the same line is a common exception to Rule #9.

Use of tools such as *ODE45* will be treated in the numerical methods section.

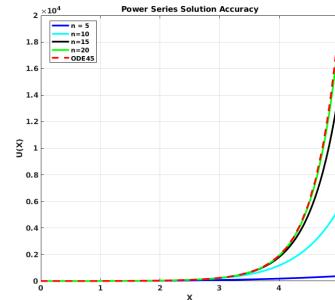


Figure 3: Power series solution with different values of n .

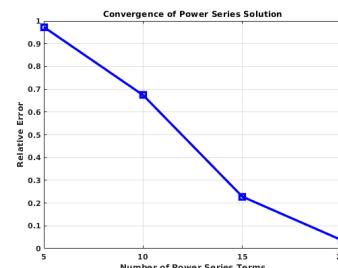


Figure 4: Convergence of the power series solution to the numeric solution.

Lecture 10 - Legendre's Equation

Objectives

The objectives of this lecture are:

- Illustrate the use of the power series method to solve Legendre's equation.
- Introduce some of the properties of Legendre polynomials.

Legendre's Equation

The following 2nd-order linear, homogeneous ODE is known as Legendre's equation:

$$(1-x^2) u'' - 2xu' + m(m+1)u = 0 \quad (43)$$

where m is a constant.

FIRST WE WILL put Equation 43 into standard form:

$$u'' - \frac{2x}{(1-x^2)}u' + \frac{m(m+1)}{(1-x^2)}u = 0$$

We should immediately note that $P(x) = \frac{2x}{(1-x^2)}$ and $Q(x) = \frac{m(m+1)}{(1-x^2)}$ are singular (and thus not analytic) at $x = \pm 1$. Recall from Theorem 6 that $P(x)$ and $Q(x)$ must be analytic for power series solutions to exist.

WE WILL RESTRICT our attention to the interval $x \in (-1, 1)$ and use the power series method to find a solution. Inserting our assumed power series solution into Equation 43 gives us:

$$(1-x^2) \sum_{n=2}^{\infty} n(n-1)c_n x^{n-2} - 2x \sum_{n=1}^{\infty} nc_n x^{n-1} + m(m+1) \sum_{n=0}^{\infty} c_n x^n = 0$$

$$\underbrace{\sum_{n=2}^{\infty} n(n-1)c_n x^{n-2}}_{x^0} - \underbrace{\sum_{n=2}^{\infty} n(n-1)c_n x^n}_{x^2} - \underbrace{\sum_{j=1}^{\infty} 2nc_n x^n}_{x^1} + \underbrace{m(m+1) \sum_{n=0}^{\infty} c_n x^n}_{x^0} = 0$$

where we see that all terms with order lower than x^2 need to be pulled outside of their summations so all four can be in phase.

$$(2)(1)c_2 + (3)(2)c_3x + \underbrace{\sum_{n=4}^{\infty} n(n-1)c_nx^{n-2}}_{\substack{k=n-2 \\ n=k+2}} - \underbrace{\sum_{n=2}^{\infty} n(n-1)c_nx^n}_{\substack{k=n \\ n=k}} - (2)(1)c_1x - \underbrace{\sum_{n=2}^{\infty} 2nc_nx^n}_{\substack{k=n \\ n=k}} + \dots$$

$$m(m+1)(c_0 + c_1x) + m(m+1)\underbrace{\sum_{n=2}^{\infty} c_nx^n}_{\substack{k=n \\ n=k}} = 0$$

Combining terms outside of the summations and making the indicated substitutions to combine the summations we get:

$$[m(m+1)c_0 + 2c_2] + \underbrace{[(m(m+1)-2)c_1 + 6c_3]}_{(m-1)(m+2)}x + \dots$$

$$\sum_{k=2}^{\infty} [(k+2)(k+1)c_{k+2} - \underbrace{k(k-1)c_k - 2kc_k + m(m+1)c_k}_{(m-k)(m+k+1)c_k}]x^k = 0$$

Applying the indicated algebraic simplifications leads us finally to:

$$[m(m+1)c_0 + 2c_2] + [(m+1)(m+2)c_1 + 6c_3]x + \dots$$

$$\sum_{k=2}^{\infty} [(k+2)(k+1)c_{k+2} + (m-k)(m+k+1)c_k]x^k = 0$$

Note: Obviously this is a tedious business. Be careful and make sure you understand each manipulation.

The next steps are to find formulas for the power series coefficients, c_n , so that the combined coefficient for each power of x in the equation above equals zero. For the constant term (x^0) we have:

$$c_2 = \frac{-m(m+1)c_0}{2}$$

For the linear term (x^1) we get:

$$c_3 = \frac{-(m-1)(m+2)}{6}c_1$$

For all other powers of x , we get a 2-term recurrence:

$$c_{k+1} = \frac{-(m-k)(m+k+1)}{(k+2)(k+1)}c_k$$

$$k = 2 \\ c_4 = \frac{-(m-2)(m+3)}{(4)(3)}c_2 = \frac{(m-2)(m+3)m(m+1)}{(4)(3)(2)}c_0 \\ k = 3 \\ c_5 = \frac{-(m-3)(m+4)}{(5)(4)}c_3 = \frac{(m-3)(m+4)(m-1)(m+2)}{(5)(4)(3)(2)}c_1$$

Organizing these into two solutions we get:

$$\begin{aligned} u_1(x) &= c_0 \left[1 + \frac{c_2}{c_0} x^2 + \frac{c_4}{c_0} x^4 + \dots \right] \\ &= c_0 \left[1 - \frac{m(m+1)}{2!} x^2 + \frac{(m-2)(m+3)m(m+1)}{4!} x^4 + \dots \right] \\ u_2(x) &= c_1 \left[x + \frac{c_3}{c_1} x^3 + \frac{c_5}{c_1} x^5 + \dots \right] \\ &= c_1 \left[x - \frac{(m-1)(m+2)}{3!} x^3 + \frac{(m-3)(m+4)(m-1)(m+2)}{5!} x^5 + \dots \right] \end{aligned}$$

So far what we have is messy but, when dealing with power series solutions, messiness is the order of the day. One point that we have quietly left to the side is whether or not we expect this (so called) power series solution to converge.

One way that we can permanently leave these questions behind us is if m is an integer. Notice that if $m = 0$ or is an even integer, $u_1(x)$ terminates with a finite number of terms.¹ Similarly with $u_2(x)$ in the case that m is an odd integer.

Reminder: If the power series that we purport to be a solution to the differential equation is divergent than we really have nothing.

¹ i.e. If m is even then $u_1(x)$ is a polynomial.

THESE POLYNOMIAL SOLUTIONS, where m is an integer, are referred to as Legendre polynomials. Legendre polynomials have several important applications; our primary use for them will be when solving equations in spherical coordinate systems.

Important Properties of Legendre Polynomials

Legendre polynomials are solutions to Legendre's equation where m is an integer:

$$(1 - x^2) u'' - 2xu' + m(m+1)u = 0$$

By convention the leading coefficients are chosen such that Legendre polynomials have a maximum value of 1 on the interval $x \in [-1, 1]$.

The first few Legendre Polynomials are shown in the Table 2. Higher order Legendre polynomials can be constructed using a three-term recurrence relation shown in Equation 44:

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0 \quad (44)$$

$P_0(x) = 1$	$P_1(x) = x$
$P_2(x) = \frac{1}{2}(3x^2 - 1)$	$P_3(x) = \frac{1}{2}(5x^3 - 3x)$

Table 2: The first four Legendre Polynomials

Some other properties include:

$$\begin{aligned} P_n(-x) &= (-1)^n P_n(x) \\ P_n(1) &= 1 \\ P_n(-1) &= (-1)^n \\ P_n(0) &= 0 \text{ for } n \in \text{odd} \\ P'_n(0) &= 0 \text{ for } n \in \text{even} \end{aligned}$$

A plot of the first few Legendre Polynomials is shown in Figure 5.

THE LAST PROPERTY that we will mention here, and that we will make use of extensively in this course, is the *orthogonality* property of Legendre polynomials. Legendre polynomials are orthogonal over the interval $x \in [-1, 1]$. This mean that Equation 45 holds:

$$\int_{-1}^1 P_n(x) P_m(x) dx = \begin{cases} 0, & n \neq m \\ \frac{2}{n+1}, & n = m \end{cases} \quad (45)$$

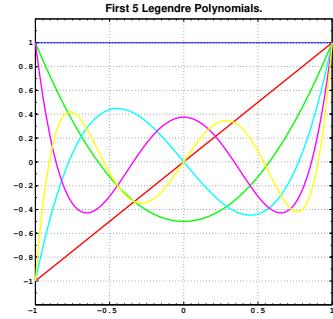


Figure 5: Legendre Polynomials of order 0 through 5.

Orthogonality of functions is analogous to orthogonality of vectors. Two functions, $f_1(x)$ and $f_2(x)$ are orthogonal on an interval $x \in [a, b]$ if $\int_a^b f_1(x) f_2(x) dx = 0$.

Lecture 11 - Solutions about Singular Points

Objectives

The objectives of this lecture are:

- Define regular and irregular singular points and give examples of their classification.
- Describe the extended power series method (method of Frobenius).
- Do an example problem.

Definitions

Consider a linear, homogeneous, second-order differential equation in standard form as shown below:

$$u'' + P(x)u' + Q(x)u = 0$$

Definition 12 (Singular Point)

A singular point, x_0 , is a point where $P(x)$ or $Q(x)$ are not analytic.

Definition 13 (Regular/Irregular Singular Point)

A singular point x_0 is said to be a regular singular point of the differential equation if the functions $p(x) = (x - x_0)P(x)$ and $q(x) = (x - x_0)^2Q(x)$ are both analytic at x_0 . If a singular point is not regular, it is irregular.

Example: Classify the singular points of $(x^2 - 4)^2u'' + 3(x - 2)u' + 5u = 0$.

In standard form, $P(x) = \frac{3(x-2)}{(x^2-4)^2} = \frac{3(x-2)}{(x+2)^2(x-2)^2}$; and $Q(x) = \frac{5}{(x+2)^2(x-2)^2}$. There are two singular points: -2 and 2. Work is shown in the margin for $p(x)$ and $q(x)$. From the work in the margin it should be clear for this problem that $q(x)$ is analytic at both $x = -2$ and $x = 2$ but, since $p(x)$ is not analytic at $x_0 = -2$, $x_0 = -2$ is an irregular singular point; $x_0 = 2$ is a regular singular point.

$x_0 = 2$: $p(x) = \frac{(x-2)3(x-2)}{(x+2)^2(x-2)^2}$, so $p(x) = \frac{3}{(x+2)^2}$ which is analytic at $x = 2$.
 $q(x) = \frac{(x-2)^25}{(x+2)^2(x-2)^2}$, so $q(x) = \frac{5}{(x+2)^2}$ which is analytic at $x = 2$.

$x_0 = -2$: $p(x) = \frac{(x+2)3(x-2)}{(x+2)^2(x-2)^2}$ so
 $p(x) = \frac{3(x-2)}{(x+2)(x-2)}$ which is not analytic
at $x = -2$. $q(x) = \frac{(x+2)^25}{(x+2)^2(x-2)^2}$, so
 $q(x) = \frac{5}{(x-2)^2}$ which is analytic at
 $x = -2$.

Theorem 7 (Frobenius' Theorem)

If $x = x_0$ is a regular singular point then there exists at least one non-zero solution of the form:

$$u(x) = (x - x_0)^r \sum_{n=0}^{\infty} c_n (x - x_0)^n = \sum_{n=0}^{\infty} c_n (x - x_0)^{n+r}$$

where r is to be determined. The series will converge at least on some radius of convergence defined by: $0 < x - x_0 < R$.

Method of Frobenius

The definitions and theorems provided above lead us to the method of Frobenius which we will illustrate through example.

Example: Find a series solution to: $3xu'' + u' - u = 0$.

Per Theorem 7, this equation should have a solution of the form: $u = \sum_{n=0}^{\infty} c_n x^{n+r}$ where r is constant. Taking the first and second derivatives we get:

$$\begin{aligned} u' &= \sum_{n=0}^{\infty} (n+r)c_n x^{n+r-1} \\ u'' &= \sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r-2} \end{aligned}$$

We insert these expressions into the differential equation and will combine the summations to derive recurrence relations.

$$\begin{aligned} 3x \sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r-2} + \sum_{n=0}^{\infty} (n+r)c_n x^{n+r-1} - \sum_{n=0}^{\infty} c_n x^{n+r} &= 0 \\ \sum_{n=0}^{\infty} 3(n+r)(n+r-1)c_n x^{n+r-1} + \sum_{n=0}^{\infty} (n+r)c_n x^{n+r-1} - \sum_{n=0}^{\infty} c_n x^{n+r} &= 0 \end{aligned}$$

$$x^r \left[\underbrace{\sum_{n=0}^{\infty} 3(n+r)(n+r-1)c_n x^{n-1}}_{\text{for } n=0 \atop x^{-1}} + \underbrace{\sum_{n=0}^{\infty} (n+r)c_n x^{n-1}}_{\text{for } n=0 \atop x^{-1}} - \underbrace{\sum_{n=0}^{\infty} c_n x^n}_{\text{for } n=0 \atop x^0} \right] = 0$$

We can see now that one term must be “peeled off” from the first two summations in order to get the summations in phase.

You should verify that this equation has a regular singular point at $x_0 = 0$.

Notice in this case that the summations for u' and u'' start at $n = 0$ while for the power series solution method the starting index for u' was $n = 1$ and the starting index for u'' was $n = 2$.

The reason for this is that, for the power series method, the constant term of u , corresponding to $n = 0$, becomes zero for u' ; so we omit that term and start with $n = 1$. Both the constant and linear term in u , corresponding to $n = 0$ and $n = 1$, are zero for u'' . So the series for u'' starts at $n = 2$.

The factor x^r included in the method of Frobenius means there may not be any constant terms in the series at all—i.e. if r is not an integer. Therefore there is no reason to omit terms for u' or u'' .

$$x^r \left[\sum_{n=1}^{\infty} 3(n+r)(n+r-1)c_n x^{n-1} + \sum_{n=1}^{\infty} (n+r)c_n x^{n-1} - \sum_{n=0}^{\infty} c_n x^n \right] + \dots$$

$\underbrace{x^r [3r(r-1) + r] c_0 x^{-1}}_{\substack{\text{first terms from} \\ \text{first two summations}}} = 0 \quad (46)$

Let us focus for a moment on the last term on the left-hand side of Equation 46. We know from our experience with the power series solution process that, in order to *solve* the equation, the coefficient for each power of x needs to be zero. Consider now specifically the coefficient for x^{r-1} . That coefficient needs to be zero and there are a couple of ways that it can be set to zero which are discussed in the margin note.

THE CUSTOMARY PROCEDURE for the method of Frobenius dictates that we go with option #2. We refer to $-3r(r-1) + r = 0$ —as the *indicial equation* and the roots of the indicial equation are known as the *indicial roots*.¹

FOR THIS CASE, the indicial equation can be factored:

$$\begin{aligned} f(x) &= 3r(r-1) + r \\ &= 3r^2 - 3r + r \\ &= 3r^2 - 2r \\ &= r(3r - 2) = 0 \end{aligned}$$

so the roots are $r_1 = 0$, and $r_2 = 2/3$. So long as r is chosen to be one of those values, the coefficient for x^{r-1} will be zero. Let us refocus our attention on the remaining terms:

$$x^r \left[\sum_{n=1}^{\infty} 3(n+r)(n+r-1)c_n x^{n-1} + \sum_{n=1}^{\infty} (n+r)c_n x^{n-1} - \sum_{n=0}^{\infty} c_n x^n \right] = 0$$

The summations are all in phase—recall that is how we obtained the indicial equation—but we need to combine the three summations under a common index.

$$x^r \left[\underbrace{\sum_{n=1}^{\infty} 3(n+r)(n+r-1)c_n x^{n-1}}_{\substack{k=n-1 \\ n=k+1}} + \underbrace{\sum_{n=1}^{\infty} (n+r)c_n x^{n-1}}_{\substack{k=n-1 \\ n=k+1}} - \underbrace{\sum_{n=0}^{\infty} c_n x^n}_{\substack{k=n \\ n=k}} \right] = 0$$

$$x^r [3r(r-1) + r] c_0 x^{-1} = 0$$

Option #1 set $c_0 = 0$;

Option #2 set r to a root of $3r(r-1) + r = 0$.

¹ For second-order problems, the form of the indicial equation will be a quadratic.

Making the indicated substitution in each summation gives us:

$$x^r \left\{ \sum_{k=0}^{\infty} \left[\underbrace{3(k+1+r)(k+r)c_{k+1}}_{(k+1+r)(3(k+r)+1)c_{k+1}} + \underbrace{(k+1+r)c_{k+1}}_{(k+1+r)(3(k+r)+1)c_{k+1}-c_k} - c_k \right] x^k \right\} = 0$$

The resulting recurrence relation for the coefficient of x^{k+r} is:

$$c_{k+1} = \frac{c_k}{(k+1+r)(3k+3r+1)}$$

We have two cases: one for $r = 0$; the other for $r = 2/3$.

$r = 0$:

$$c_{k+1} = \frac{c_k}{(k+1)(3k+1)}$$

Coefficients are shown in the table in the margin; the resulting solution is:

$$\begin{aligned} u_1(x) &= x^0 \left(c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \dots \right) \\ &= c_0 \left(1 + \frac{c_1}{c_0} x + \frac{c_2}{c_0} x^2 + \frac{c_3}{c_0} x^3 + \dots \right) \\ &= c_0 \left(1 + x + \frac{1}{8} x^2 + \frac{1}{168} x^3 + \dots \right) \end{aligned}$$

$r = 2/3$:

$$\begin{aligned} c_{k+1} &= \frac{c_k}{(k+1+\frac{2}{3})(3k+3(\frac{2}{3})+1)} \\ &= \frac{c_k}{(k+\frac{5}{3})(3k+3)} \\ &= \frac{c_k}{(3k+5)(k+1)} \end{aligned}$$

Coefficients are shown in the table in the margin; the resulting solution is:

$$\begin{aligned} u_2(x) &= x^{2/3} \left(c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \dots \right) \\ &= c_0 x^{2/3} \left(1 + \frac{c_1}{c_0} x + \frac{c_2}{c_0} x^2 + \frac{c_3}{c_0} x^3 + \dots \right) \\ &= c_0 x^{2/3} \left(1 + \frac{1}{5} x + \frac{1}{80} x^2 + \frac{1}{264} x^3 + \dots \right) \end{aligned}$$

case 1, r = 0
$k = 0$
$c_1 = \frac{c_0}{(1)(1)} = c_0$
$k = 1$
$c_2 = \frac{c_1}{(2)(4)} = \frac{c_0}{8}$
$k = 2$
$c_3 = \frac{c_2}{(3)(7)} = \frac{c_0}{(3)(7)(8)}$

case 2, r = 2/3
$k = 0$
$c_1 = \frac{c_0}{(5)(1)} = \frac{c_0}{5}$
$k = 1$
$c_2 = \frac{c_1}{(8)(2)} = \frac{c_0}{(2)(5)(8)}$
$k = 2$
$c_3 = \frac{c_2}{(11)(3)} = \frac{c_0}{(2)(3)(5)(8)(11)}$

A QUICK INSPECTION of $u_1(x)$ and $u_2(x)$ should be sufficient to convince you that the solutions are linearly independent. The general solution to the differential equation comprises a linear combination of $u_1(x)$ and $u_2(x)$.

Indicial Equation

It turns out that we could have determined the indicial roots before setting out upon the method of Frobenius. Recall that we use the method of Frobenius on differential equations with regular singular points; also recall that a regular singular point is one where $p(x) = xP(x)$ and $q(x) = x^2Q(x)$ are both analytic. By definition, if $p(x)$ and $q(x)$ are analytic, that means that they can be represented as a convergent power series. Suppose we did that, and expressed $p(x)$ and $q(x)$ as a power series; if we did they could be written as:

$$p(x) = \sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + \dots$$

$$q(x) = \sum_{n=0}^{\infty} b_n x^n = b_0 + b_1 x + b_2 x^2 + \dots$$

It can be shown that the indicial equation that we derive from the Method of Frobenius will be equal to:

$$r(r-1) + a_0 r + b_0 = 0$$

where $a_0 = p(0)$ and $b_0 = q(0)$. Applying this equation to our last example where $p(x) = xP(x) = 1/3$ and $q(x) = x^2Q(x) = -x/3$. We can see $a_0 = p(0) = 1/3$ and $b_0 = q(0) = 0$. Inserting these numbers into the indicial equation gives us:

$$\begin{aligned} r(r-1) + \frac{1}{3}r + 0 &= 0 \\ r^2 - r + \frac{1}{3}r &= 0 \\ r^2 - \frac{2}{3}r &= 0 \\ r(r - \frac{2}{3}) &= 0 \end{aligned}$$

which has the roots: $r = 0$, and $r = 2/3$.

Example: Use the indicial equation to determine the indicial roots to:

$$2xu'' - (3+2x)u' + u = 0$$

We see that $P(x) = -\frac{(3-2x)}{2x}$, so $p(x) = xP(x) = -\frac{(3-2x)}{2}$, and $p(0) = -3/2$. By inspection $q(x) = x^2Q(x) = \frac{x}{2}$, so $q(0) = 0$. The

indicial equation is:

$$\begin{aligned} r(r-1) - \frac{3}{2}r + 0 &= 0 \\ r^2 - r - \frac{3}{2}r &= 0 \\ r^2 - \frac{5}{2}r &= 0 \\ r \left(r - \frac{5}{2} \right) &= 0 \end{aligned}$$

so the indicial roots are: $r = 0$, and $r = 5/2$.

IN GENERAL, OF COURSE, the indicial equation is just a quadratic equation, so the roots may be real and repeated, real and distinct or complex conjugates. There are three cases that will be of immediate interest to us:

1. Two distinct real roots that do *not* differ by an integer. In this case it can be shown that there exist two linearly independent solutions: $u_1 = \sum_{n=0}^{\infty} c_n x^{n+r_1}$, and $u_2 = \sum_{n=0}^{\infty} c_n x^{n+r_2}$.
2. Two distinct real roots that differ by an integer. In this case there exist two linearly independent solutions of the form:

$$\begin{aligned} u_1(x) &= \sum_{n=0}^{\infty} c_n x^{n+r_1}, \quad c_0 \neq 0 \\ u_2(x) &= C u_1(x) \ln x + \sum_{n=0}^{\infty} b_n x^{n+r_2}, \quad b_0 \neq 0 \end{aligned}$$

In this case, the constant C *might* be zero.

3. If both roots are real and $r_1 = r_2$ then there exist two linearly independent solutions of the form:

$$\begin{aligned} u_1(x) &= \sum_{n=0}^{\infty} c_n x^{n+r_1}, \quad c_0 \neq 0 \\ u_2(x) &= u_1(x) \ln x + \sum_{n=0}^{\infty} b_n x^{n+r_2}, \quad b_0 \neq 0 \end{aligned}$$

Additional Notes:

- When the difference between the indicial roots is equal to an integer, find the solution with the smaller root first.
- The indicial equation could, in principle, have complex roots. We will avoid those cases for this class.
- If x_0 is an irregular singular point, the Frobenius theorem does not apply and we may not be able to find any solution to the differential equation using this method.

In both of these cases we implicitly assume that $c_0 \neq 0$.

Note: The goal in this treatment of method of Frobenius is not to make you a "Frobenius Genius." The goal is to provide a sufficiently thorough introduction so that you can understand where Bessel functions and other such mathematical objects come from. For this reason, we will emphasize systems that fall into case 1.

For ordinary differential equations that fall into the last two categories, do not fret: numerical methods are always available that are more than adequate for finding solutions to the equations.

Assignment #4

Find two power series solutions of the given differential equation.

1. $u'' + x^2 u' + xu = 0$

2. $u'' - (x + 1)u' - u = 0$

3. Solve the given initial value problem. Use MATLAB to represent the power series. Make 2 plots; for the first plot compare the partial sum of the power series with 5 terms to the exact solution which is $u = 8x - 2e^x$; for the second plot compare the partial sum of the power series with 15 terms to the exact solution. Submit the “published” version of your MATLAB script (PDF format) along with your written solution.

$$(x - 1)u'' - xu' + u = 0, \quad u(0) = -2, \quad u'(0) = 6$$

Determine the singular points for the differential equation. Classify each singular point as irregular or regular.

4. $(x^2 - 9)^2 u'' + (x + 3)u' + 2u = 0$

5. $x^3 (x^2 - 25) (x - 2)^2 u''' + 3x(x - 2)u' + 7(x + 5)u = 0$

Use the general form of the indicial equation to find the indicial roots.

6. $x^2 u'' + \left(\frac{5}{3}x + x^2\right) u' - \frac{1}{3}u = 0$

Use the method of Frobenius to obtain two linearly independent series solutions:

7. $3xu'' + (2 - x)u' - u = 0$

Lecture 12 - Bessel's Equation and Bessel Functions

Objectives

The objectives of this lecture are:

- Introduce Bessel's equation and solve it using the method of Frobenius.
- Discuss Bessel functions of the 1st and 2nd kind and use them to solve instances of Bessel's equation.

Bessel's Equation

Bessel's equation is given in Equation 47:

$$x^2 u'' + xu' + (x^2 - \nu^2) u = 0 \quad (47)$$

where ν is a constant.¹ You should spend a moment to verify that this equation has a singular point at $x_0 = 0$ and that it is a regular singular point. Therefore we should use the method of Frobenius to find solutions and that is what we shall do.

As a REMINDER, we will make the following substitutions into Equation 47:

$$\begin{aligned} u(x) &= \sum_{n=0}^{\infty} c_n x^{n+r} \\ u'(x) &= \sum_{n=0}^{\infty} (n+r)c_n x^{n+r-1} \\ u''(x) &= \sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r-2} \end{aligned}$$

which gives us:

$$x^2 \sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r-2} + x \sum_{n=0}^{\infty} (n+r)c_n x^{n+r-1} + (x^2 - \nu^2) \sum_{n=0}^{\infty} c_n x^{n+r} = 0$$

¹ Let me warn you for the first time here that, if $\nu = 0$, Bessel's equation bears a striking resemblance to the Cauchy-Euler equation. Notice the difference and try not to fall into that trap.

or, if we distribute terms through the sums:

$$\sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r} + \sum_{n=0}^{\infty} (n+r)c_n x^{n+r} + \sum_{n=0}^{\infty} c_n x^{n+r+2} - \sum_{n=0}^{\infty} \nu^2 c_n x^{n+r} = 0$$

Let us inspect the first term in each summation and see what needs to be done to get the summations in phase.

$$\underbrace{\sum_{n=0}^{\infty} (n+r)(n+r-1)c_n x^{n+r}}_{n=0, x^r} + \underbrace{\sum_{n=0}^{\infty} (n+r)c_n x^{n+r}}_{n=0, x^r} + \underbrace{\sum_{n=0}^{\infty} c_n x^{n+r+2}}_{n=0, x^{r+2}} - \underbrace{\sum_{n=0}^{\infty} \nu^2 c_n x^{n+r}}_{n=0, x^r} = 0$$

For reasons that (hopefully) will become apparent, we are going to go through this process in two steps. For $n = 0$, we will separate out all terms that are proportional to x^r .

$$\begin{aligned} r(r-1)c_0 x^r + x^r \sum_{n=1}^{\infty} (n+r)(n+r-1)c_n x^n + r c_0 x^r + x^r \sum_{n=1}^{\infty} (n+r)c_n x^n + \dots \\ x^r \sum_{n=0}^{\infty} c_n x^{n+2} - \nu^2 c_0 x^r - x^r \sum_{n=1}^{\infty} \nu^2 c_n x^n = 0 \end{aligned}$$

Now let us collect the terms outside of the summations and re-write the equation:

$$\begin{aligned} & \text{indicial equation} \\ & \overbrace{[r(r-1) + r - \nu^2]} c_0 x^r + \dots \\ & x^r \sum_{n=1}^{\infty} \underbrace{[(n+r)(n+r-1) + (n+r) - \nu^2]}_{\text{combined 3 of 4 summations}} c_n x^n + x^r \sum_{n=0}^{\infty} c_n x^{n+2} = 0 \end{aligned}$$

From the indicial equation:

$$\begin{aligned} r(r-1) + r - \nu^2 &= 0 \\ r^2 - r + r - \nu^2 &= 0 \\ r^2 - \nu^2 &= 0 \\ (r - \nu)(r + \nu) &= 0 \end{aligned}$$

we see that, to ensure the coefficient for $x^r = 0$, $r = \pm\nu$. To simplify the discussion to follow, let us take $r = \nu$ and continue with the solution. Our equation is now:

$$\begin{aligned} x^\nu \sum_{n=1}^{\infty} \left[\underbrace{(n+\nu)(n+\nu-1) + (n+\nu) - \nu^2}_{n^2 + 2n\nu + \nu^2 - \nu^2} \right] c_n x^n + x^\nu \sum_{n=0}^{\infty} c_n x^{n+2} &= 0 \\ x^\nu \sum_{n=1}^{\infty} \underbrace{[n(n+2\nu)] c_n x^n}_{n=1, x^1} + x^\nu \sum_{n=0}^{\infty} \underbrace{c_n x^{n+2}}_{n=0, x^2} &= 0 \end{aligned}$$

The first line of the equation is the indicial equation for this problem; we use it to determine allowable values of r . In the second line of the equation we have combined the first, second, and fourth summation because they were in phase and had a common index. The remaining summation needs to be put in phase before we can combine everything under a single summation.

Reminder: We need to ensure the coefficient for x^r is equal to zero. By convention we assume $c_0 \neq 0$. We could allow $c_0 = 0$ but then we would just need to derive another indicial equation for some other power of x . We adopt the convention $c_0 \neq 0$ so that our choices for indicial roots will be unique.

The two summations are out of phase, so we need to separate out the first term of the first summation.

$$\underbrace{(1)(1+2\nu)c_1}_{\text{coefficient for } x^{\nu+1}} x^{\nu+1} + \underbrace{x^\nu \sum_{n=2}^{\infty} c_n [n(n+2\nu)] x^n}_{n=2, x^{\nu+2}} + \underbrace{x^\nu \sum_{n=0}^{\infty} c_n x^{n+2}}_{n=0, x^{\nu+2}} = 0$$

In order to satisfy the equation, we need the coefficient for $x^{\nu+1}$ to be equal to zero; the only way to do this is to set $c_1 = 0$.²

The summations in the equation above are in-phase so we need to combine under a common index.

$$\underbrace{x^\nu \sum_{n=2}^{\infty} c_n [n(n+2\nu)] x^n}_{\substack{k=n \\ n=k}} + \underbrace{x^\nu \sum_{n=0}^{\infty} c_n x^{n+2}}_{\substack{k=n+2 \\ n=k-2}} = 0$$

$$x^\nu \sum_{k=2}^{\infty} \underbrace{[k(k+2\nu)c_k + c_{k-2}]}_{\text{coefficient for } x^{\nu+k}} x^k = 0$$

In order to set the coefficients for $x^{\nu+k}$ to zero, we derive the following two-term recurrence:

$$c_k = \frac{-c_{k-2}}{k(k+2\nu)}, \quad k = 2, 3, 4, \dots \quad (48)$$

Since we have already determined that $c_1 = 0$, Equation 48 tells us that $c_3 = c_5 = \dots = 0$; all the odd-numbered coefficients must be zero. To simplify the notation further, we will thus assume that $k = 2n$ and re-write our recurrence as:

$$c_{2n} = \frac{-c_{2n-2}}{2n(2n+2\nu)}, \quad n = 1, 2, 3, \dots$$

$$= \frac{-c_{2n-2}}{2^2 n(n+\nu)}, \quad n = 1, 2, 3, \dots$$

Expressions for the first few terms is given in Table 3. From this pattern you should be able to see that the general form of the coefficients is as shown in Equation 49.

$$c_{2n} = \frac{(-1)^n c_0}{2^{2n} n! (1+\nu)(2+\nu) \cdots (n+\nu)}, \quad n = 1, 2, 3, \dots \quad (49)$$

IT MAY BE WORTHWHILE to take a step back and summarize what we have found so far. We are solving Bessel's equation and we looked for solutions of the form $u(x) = \sum_{n=0}^{\infty} c_n x^{n+r}$. We found that r must be equal to $\pm\nu$ and, for the case $r = \nu$, derived a perfectly acceptable expression for the coefficients of this solution in Equation 49. What follows is a bit of, what we will call, "mathematical grooming" which we will do so that we can derive solutions to Bessel's equation in a form that appears elsewhere in the literature and that, it turns out, you will use for the remainder of this course.

² **Reminder:** We do not control what ν is; that is part of the equation.

$n = 1$	$c_2 = \frac{-c_0}{2^2(1)(1+\nu)}$
$n = 2$	$c_4 = \frac{-c_2}{2^2(2)(2+\nu)} = \frac{c_0}{2^4(1)(2)(1+\nu)(2+\nu)}$
$n = 3$	$c_6 = \frac{-c_4}{2^2(3)(3+\nu)} = \frac{-c_0}{2^6(1)(2)(3)(1+\nu)(2+\nu)(3+\nu)}$

Table 3: First few coefficients in solution to Bessel's Equation.

We will deal with the case $r = -\nu$, albeit in a perfunctory manner, below.

Gamma Function

The gamma function is defined in Equation 50.

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \quad (50)$$

One property of the gamma function is that $\Gamma(x+1) = x\Gamma(x)$ for any real argument x . If x is an integer, this makes the gamma function equivalent to a factorial:

$$\begin{aligned}\Gamma(1) &= \int_0^\infty t^{1-1} e^{-t} dt \\ &= \int_0^\infty e^{-t} dt \\ &= -e^{-t} \Big|_0^\infty \\ &= -[0 - 1] \\ &= 1\end{aligned}$$

Note: The gamma function is also defined when a complex argument is used, but that is beyond the scope of this class.

$$\begin{aligned}\Gamma(2) &= \Gamma(1+1) = 1\Gamma(1) = 1 \\ \Gamma(3) &= \Gamma(2+1) = 2\Gamma(2) = 2 \\ \Gamma(4) &= \Gamma(3+1) = 3\Gamma(3) = 6\end{aligned}$$

Put differently, the Gamma function is a *generalization* of a factorial.

Therefore, in general for $x \in \mathbb{I}$, $\Gamma(x) = (x-1)!$.

IN THE CONTEXT of our solution to Bessel's equation, we use the gamma function represent the term $(1+\nu)(2+\nu)\cdots(n+\nu)$ in the denominator of Equation 49 in a compact way:

$$\begin{aligned}\Gamma(1+\nu+1) &= (1+\nu)\Gamma(1+\nu) \\ \Gamma(1+\nu+2) &= (2+\nu)\Gamma(2+\nu) = (2+\nu)(1+\nu)\Gamma(1+\nu) \\ \Gamma(1+\nu+3) &= (3+\nu)\Gamma(3+\nu) = (3+\nu)(2+\nu)(1+\nu)\Gamma(1+\nu) \\ &\vdots \\ \Gamma(1+\nu+n) &= (n+\nu)\cdots(1+\nu)\Gamma(1+\nu)\end{aligned}$$

Bessel Function of the First Kind of order ν

We will use everything that we have done thus far to define a Bessel function of the first kind of order ν . We will start with our series solution $u(x) = \sum_{n=0}^{\infty} c_n x^{n+\nu}$ and the formula for the non-zero (even) coefficients given in Equation 49 and take a couple of steps:

1. We will set $c_0 = \frac{1}{2^\nu \Gamma(1+\nu)}$.

$$\begin{aligned} c_{2n} &= \frac{(-1)^n c_0}{2^{2n} n! (1+\nu)(2+\nu) \cdots (n+\nu)}, \quad n = 1, 2, 3, \dots \\ &= \frac{(-1)^n}{2^{2n} n! (1+\nu)(2+\nu) \cdots (n+\nu)} \frac{1}{2^\nu \Gamma(1+\nu)} \\ &= \frac{(-1)^n}{2^{2n+\nu} n! (1+\nu) \cdots (n+\nu) \Gamma(1+\nu)} \\ &= \frac{(-1)^n}{2^{2n+\nu} n! \Gamma(1+\nu+n)} \end{aligned}$$

2. Combining this new expression for c_{2n} into the solution gives us the standard definition for a Bessel function of the first kind:

$$J_\nu(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{n! \Gamma(1+\nu+n)} \left(\frac{x}{2}\right)^{2n+\nu} \quad (51)$$

We can similarly handle the case where $r = -\nu$:

$$J_{-\nu} = \sum_{n=0}^{\infty} \frac{(-1)^n}{n! \Gamma(1-\nu+n)} \left(\frac{x}{2}\right)^{2n-\nu}$$

We will not prove this, but if ν is *not* an integer, then J_ν and $J_{-\nu}$ are linearly independent. In that case, the solution (at long last) to Bessel's equation is just a linear combination of $J_\nu(x)$ and $J_{-\nu}(x)$ as shown in Equation 52.

$$u(x) = c_1 J_\nu(x) + c_2 J_{-\nu}(x) \quad (52)$$

Bessel Function of the Second Kind of order ν

If ν is an integer, then $J_\nu(x)$ and $J_{-\nu}(x)$ are *not* linearly independent so we need to find another solution to Bessel's equation. To this end, we define the Bessel function of the second kind of order ν , given in Equation 53:

$$Y_\nu(x) = \frac{\cos \nu \pi J_\nu(x) - J_{-\nu}(x)}{\sin \nu \pi} \quad (53)$$

which is linearly independent of J_ν even if ν is an integer. The solution to Bessel's equation can thus alternately be expressed:

$$u(x) = c_1 J_\nu(x) + c_2 Y_\nu(x)$$

NOW THAT WE KNOW a pair of linearly independent solutions to Bessel's equation, we no longer need to go through the rigmarole of *actually solving* the equation; we can simply *use* the solution we have derived.

Remember c_0 is just an arbitrary constant. This decision allows us, with the help of gamma functions, to express the coefficients to the solution in a compact form. The resulting solution can then be multiplied by *another* arbitrary constant if needed to satisfy a given initial/boundary condition.

It can be shown that if $\nu \geq 0$ the series converges for all x .

We were sly about it, but we quietly added the $n = 0$ term to the summation. A more verbose expression would be:

$$\begin{aligned} u(x) &= c_0 x^0 + \sum_{n=1}^{\infty} c_{2n} x^{2n+\nu} \\ &= \frac{1}{2^\nu \Gamma(1+\nu)} + \sum_{n=1}^{\infty} \frac{(-1)^n}{2^{2n+\nu} n! \Gamma(1+\nu+n)} x^{2n+\nu} \\ &= \frac{1}{2^{2(0)+\nu} 0! \Gamma(1+\nu+0)} + \cdots \\ &\quad \cdots \sum_{n=1}^{\infty} \frac{(-1)^n}{2^{2n+\nu} n! \Gamma(1+\nu+n)} x^{2n+\nu} \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{2^{2n+\nu} n! \Gamma(1+\nu+n)} x^{2n+\nu} \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{n! \Gamma(1+\nu+n)} \left(\frac{x}{2}\right)^{2n+\nu} \end{aligned}$$

I recommend that you always use J_ν and Y_ν . It's not hard to decide if ν is an integer or not but consistency has its benefits.

Example: Find the general solution to:

$$x^2 u'' + xu' + \left(x^2 - \frac{1}{9}\right) u = 0$$

We recognize the given equation as Bessel's equation of order $\nu = 1/3$.

Therefore the general solution is:

$$u(x) = c_1 J_{1/3}(x) + c_2 Y_{1/3}(x)$$

Alternatively we could, of course, have used: $u(x) = c_1 J_{1/3}(x) + c_2 J_{-1/3}(x)$.

Example: Find the general solution to:

$$xu'' + u' + xu = 0$$

To the uninitiated this may not look like Bessel's equation but with practice you will learn to automatically see the above equation as:

$$\begin{aligned} & xu'' + u' + xu = 0, \quad \text{multiply by } x \\ & x^2 u'' + xu' + x^2 u = 0 \\ & x^2 u'' + xu' + \left(x^2 - 0^2\right) u = 0 \end{aligned}$$

and recognize it to be Bessel's equation of order zero. The general solution is:

$$u(x) = c_1 J_0(x) + c_2 Y_0(x)$$

Lecture 13 - Solving ODEs Reducible to Bessel's Equation

Objectives

Demonstrate reducing an ODE to Bessel's Equation by:

- changing the dependent variable;
- changing the independent variable; and
- changing both the dependent and independent variables.

IF WE HAVE LEARNED one thing over the course of the last couple of lectures it is that using the method of Frobenius—whether we are solving Bessel's equation or some other differential equation with regular singular points—is tedious and error-prone. The good news is that if we are trying to solve a problem and recognize that the problem is Bessel's equation of some order, we can simply write down the solution in terms of Bessel functions.¹ Of course, if the equation is *not* Bessel's equation, this ability offers little benefit.

In this and the next lecture we will learn some techniques by which a broad range of differential equations can be transformed into or expressed as Bessel's equation, thereby expanding the range of equations for which we may write the solution in terms of Bessel functions. The best way to learn is by doing, so we will simply start with the examples.

Example: Find the general solution to the following differential equation by applying the given transformation to the dependent variable:
 $u = v/x^2$.

$$xu'' + 5u' + xu = 0$$

¹ Let me reiterate that this was the point to learning how to solve Bessel's equation.

You can think of this as cleverly distorting the y -axis in order to make the problem easier.

We need to replace all appearances of u with the equivalent in terms of v .

$$\begin{aligned} xu'' &= \frac{v''}{x} - \frac{4v'}{x^2} + \frac{6v}{x^3} \\ 5u' &= \frac{5v'}{x^2} - \frac{10v}{x^3} \\ xu &= \frac{v}{x} \end{aligned}$$

Combining these terms together gives us:

$$\begin{aligned} xu'' + 5u' + xu &= 0 \\ \frac{v''}{x} - \frac{4v'}{x^2} + \frac{6v}{x^3} + \frac{5v'}{x^2} - \frac{10v}{x^3} + \frac{v}{x} &= 0, \quad \text{combine like terms} \\ \frac{v''}{x} + \frac{v'}{x^2} + \left(\frac{1}{x} - \frac{4}{x^3}\right)v &= 0, \quad \text{multiply by } x^3 \\ x^2v'' + xv' + (x^2 - 4)v &= 0 \end{aligned}$$

where on the last line we recognize the ODE as Bessel's equation of order $\nu = 2$. The solution is:

$$v(x) = c_1 J_2(x) + c_2 Y_2(x)$$

Of course, we were trying to solve for $u(x)$ so we must undo the transformation to the dependent variable:

$$u(x) = \frac{v(x)}{x^2} = \frac{1}{x^2} [c_1 J_2(x) + c_2 Y_2(x)]$$

Example: Find the general solution to the following differential equation by applying the given transformation to the independent variable: $\sqrt{x} = z$.

$$4xu'' + 4u' + u = 0$$

In this case we need to change occurrences of x into its equivalent in terms of z and we need to change all derivatives with respect to x to derivatives with respect to z . We are given $\sqrt{x} = z$ which is, of course, equivalent to $x = z^2$. For the derivatives we have:

$$\begin{aligned} u' &= \frac{du}{dx} = \frac{du}{dz} \frac{dz}{dx} = u_z \frac{d}{dx} (x^{1/2}) = \frac{1}{2} \underbrace{x^{-1/2}}_{z^{-1}} u_z \\ &= \frac{1}{2z} u_z \\ u'' &= \frac{d}{dx} \left(\frac{du}{dx} \right) = \frac{d}{dz} \left(\frac{du}{dx} \right) \frac{dz}{dx} \\ &= \frac{d}{dz} \left[\frac{1}{2z} u_z \right] \frac{1}{2z} = \left[-\frac{1}{2z^2} u_z + \frac{1}{2z} u_{zz} \right] \frac{1}{2z} \\ &= \frac{1}{4z^2} u_{zz} - \frac{1}{4z^3} u_z \end{aligned}$$

Using the product rule:

$$\begin{aligned} u &= \frac{v}{x^2} \\ u' &= \frac{-2v}{x^3} + \frac{v'}{x^2} \\ u'' &= \frac{6v}{x^4} - \frac{2v'}{x^3} - \frac{2v''}{x^2} + \frac{v''}{x^2} \\ &\quad \underbrace{-\frac{4v'}{x^3}}_{x^3} \end{aligned}$$

where $dv/dx = v'$.

Seeing the necessary transformations comes with practice; that is what homework is for.

This is like cleverly distorting the x -axis with the goal of making the problem easier.

Note: It is important that you purge all expressions including x out of these derivatives. For example, when computing the equivalent of u' it was essential that we make the substitution $x^{-1/2} = z^{-1}$. When we used that result in calculating u'' and took derivatives with respect to z , any occurrence of x needed to be replaced with its equivalent in z or the derivative would have been wrong.

We now substitute these results into the original equation:

$$\begin{aligned} 4xu'' &= 4z^2 \left[\frac{1}{4z^2} u_{zz} - \frac{1}{4z^3} u_z \right] \\ 4u' &= 4 \left[\frac{1}{2z} u_z \right] \end{aligned}$$

So the transformed equation is:

$$\begin{aligned} u_{zz} - \frac{1}{z} u_z + \frac{2}{z} u_z + u &= 0, \quad \text{combine like terms} \\ u_{zz} + \frac{1}{z} u_z + u &= 0, \quad \text{multiply by } z^2 \\ z^2 u_{zz} + zu_z + \underbrace{z^2}_{(z^2-0^2)} u &= 0 \end{aligned}$$

and we can immediately recognize this as Bessel's equation of order $\nu = 0$. The general solution is:

$$\begin{aligned} u(z) &= c_1 J_0(z) + c_2 Y_0(z), \quad \text{undo transformation: } z \rightarrow x \\ u(x) &= c_1 J_0(\sqrt{x}) + c_2 Y_0(\sqrt{x}) \end{aligned}$$

Example: Find the general solution to the equation below by transforming the dependent variable $u = v\sqrt{x}$, and the independent variable $\sqrt{x} = z$.

In this case we are distorting *both* the x - and y -axis to "simplify" the problem.

$$x^2 u'' + \frac{1}{4} \left(x + \frac{3}{4} \right) u = 0$$

We will first transform the dependent variable: $u = v\sqrt{x} = x^{1/2}v$. As before we will replace all appearances of u with the equivalent in terms of v . Using the product rule:

$$\begin{aligned} u &= x^{1/2}v \\ u' &= \frac{1}{2}x^{-1/2}v + x^{1/2}v' \\ u'' &= -\frac{1}{4}x^{-3/2}v + \frac{1}{2}x^{-1/2}v' + \underbrace{\frac{1}{2}x^{-1/2}v'}_{x^{-1/2}v'} + x^{1/2}v'' \end{aligned}$$

and inserting into our equation gives us:

$$x^2 \left[x^{1/2}v'' + x^{-1/2}v' - \frac{1}{4}x^{-3/2}v \right] + \frac{1}{4} \left[x + \frac{3}{4} \right] x^{1/2}v = 0$$

Now we transform the independent variable $\sqrt{x} = z$, which is the same transformation that we did for the last example so we will not repeat the work.

From the last example:

$$\begin{aligned} v' &= \frac{1}{2z} v_z \\ v'' &= \frac{1}{4z^2} v_{zz} - \frac{1}{4z^3} v_z \end{aligned}$$

Inserting these expressions for v' and v'' into the equation and if we also add the following identities: $x^{1/2} = z$, $x = z^2$, and $x^2 = z^4$ we get:

$$\begin{aligned} z^4 \left[z \left(\frac{1}{4z^2} v_{zz} - \frac{1}{4z^3} v_z \right) + \frac{1}{z} \left(\frac{1}{2z} v_z \right) - \frac{1}{4} z^{-3} v \right] + \dots \\ \dots \frac{1}{4} \left(z^3 + \frac{3}{4} \right) zv = 0 \end{aligned}$$

Distributing the z 's and grouping terms gives us:

$$\begin{aligned} \frac{z^3}{4} v_{zz} + \left(-\frac{z^2}{4} + \frac{z^2}{2} \right) v_z + \left(-\frac{z}{4} + \frac{z^3}{4} + \frac{3z}{16} \right) v = 0, \text{ combining like terms} \\ \frac{z^3}{4} v_{zz} + \frac{z^2}{4} v_z + \left(\frac{z^3}{4} - \frac{z}{16} \right) v = 0, \text{ multiply by } 4/z \\ z^2 v_{zz} + z v_z + \left(z^2 - \frac{1}{4} \right) v = 0 \end{aligned}$$

which, at long last, we recognize as Bessel's equation of order $\nu = 1/2$.

The solution, by inspection, is:

$$v(z) = c_1 J_{1/2}(z) + c_2 Y_{1/2}(z), \text{ un-transform the dependent variable.}$$

$$u(z) = \sqrt{x} (c_1 J_{1/2}(z) + c_2 Y_{1/2}(z)), \text{ un-transform the independent variable.}$$

$$u(x) = \sqrt{x} (c_1 J_{1/2}(\sqrt{x}) + c_2 Y_{1/2}(\sqrt{x}))$$

Notes:

- Obviously, one would need to have spectacular insight to know in advance what transformations should be made in order to convert a given differential equation into Bessel's equation.
- In the next lecture we will make use of some tools that have been developed to simplify these transformations.

Lecture 14 - Modified Bessel Function and Parametric Modified Bessel Function

Objectives

- Show how to use the parametric Bessel equation of order ν .
- Describe the modified Bessel equation and, their solutions, modified Bessel functions.
- Introduce and illustrate a tool for solving second-order ODEs in terms of Bessel functions.

Many differential equations can be solved in terms of Bessel functions. This lecture will introduce some relatively simple and powerful tools for doing so.

Parametric Bessel Equation of Order ν

The parametric Bessel equation of order ν has the form given in Equation 54:

$$x^2 u'' + xu' + \left(\alpha^2 x^2 - \nu^2\right) u = 0 \quad (54)$$

Rather than state the solution outright, let us take a different shortcut and apply the well-known transformation that will convert Equation 54 into Bessel's equation that we can solve, by inspection, with Bessel functions.

THE TRANSFORMATION IS $t = \alpha x$. This, of course, means that $x = t/\alpha$ and the derivatives of u with respect to t are shown in the margin.

Applying these substitutions gives us:

$$\begin{aligned} x^2 u'' + xu' + \left(\alpha^2 x^2 - \nu^2\right) u &= 0 \\ \frac{t^2}{\alpha^2} \alpha^2 u_{tt} + \frac{t}{\alpha} \alpha u_t + \left(\alpha^2 \frac{t^2}{\alpha^2} - \nu^2\right) u &= 0 \\ t^2 u_{tt} + tu_t + \left(t^2 - \nu^2\right) u &= 0 \end{aligned}$$

The derivatives of u with respect to t :

$$\begin{aligned} \frac{dt}{dx} &= \alpha \\ u' &= \frac{du}{dx} = \frac{du}{dt} \frac{dt}{dx} = \alpha u_t \\ u'' &= \frac{d}{dx} \left(\frac{du}{dx} \right) = \dots \\ &= \frac{d}{dt} \left(\frac{du}{dx} \right) \frac{dt}{dx} = \dots \\ &= \frac{d}{dt} (\alpha u_t) \alpha = \alpha^2 u_{tt} \end{aligned}$$

The last line is, of course, Bessel's equation and the solution is:

$$u(t) = c_1 J_\nu(t) + c_2 Y_\nu(t)$$

Undoing the change of independent variables to express the answer in terms of x gives us the solution shown in Equation 55.

$$u(x) = c_1 J_\nu(\alpha x) + c_2 Y_\nu(\alpha x) \quad (55)$$

Example: Use the parametric Bessel equation to find the general solution to:

$$x^2 u'' + xu' + \left(36x^2 - \frac{1}{4}\right) u = 0$$

We recognize the equation as a parametric Bessel equation; the parameter $\alpha = \sqrt{36} = 6$ and $\nu^2 = 1/4 \Rightarrow \nu = 1/2$. The solution is:

$$u(x) = c_1 J_{1/2}(6x) + c_2 Y_{1/2}(6x)$$

Modified Bessel Equations and Bessel Functions

A subtle but non-trivial variation to Bessel's equation is when we flip one crucial sign as shown in Equation 56:

$$x^2 u'' + xu' - \left(x^2 + \nu^2\right) u = 0 \quad (56)$$

This equation can be converted into Bessel's equation by transforming the dependent variable— $u = i^{-\nu} v$ —and independent variable— $t = ix$. We will omit these details and instead simply give the solution as shown in Equation 57 which includes modified Bessel functions of the first¹ and second kind²:

$$u(x) = c_1 I_\nu(x) + c_2 K_\nu(x) \quad (57)$$

The modified Bessel's equation also has a parametric form as shown in Equation 58:

$$x^2 u'' + xu' - \left(\alpha^2 x^2 + \nu^2\right) u = 0 \quad (58)$$

with the general solution given in Equation 59.

$$u(x) = c_1 I_\nu(\alpha x) + c_2 K_\nu(\alpha x) \quad (59)$$

At this point in the course you should be developing a list, of sorts, of differential equations that you recognize and know how to analyze. Call it something like: "A Field Guide to Differential Equations I Know How To Solve". This list should include:

- first-order linear equations
- separable equations
- linear constant-coefficient equations
- linear equations with variable coefficients, including:
 - Cauchy-Euler equations
 - Legendre's equation
 - Bessel's equation; and now
 - parametric Bessel's equation.

For these last problem types you "solve" them by recognizing the equation and writing down the solution.

Note: for the example, instead of using $Y_{1/2}(6x)$ as the second linearly independent solution, we could have used $J_{-1/2}(6x)$; it is entirely up to you.

¹ Modified Bessel functions of the first kind are defined as:

$$I_\nu(x) = i^{-\nu} J_\nu(ix)$$

² Modified Bessel functions of the second kind, analogous to Bessel functions of the second kind, are defined in terms of modified Bessel functions of the first kind:

$$K_\nu(x) = \frac{\pi}{2} \frac{I_\nu(x) - I_\nu(-x)}{\sin \nu \pi}$$

Tool for Solving Second-Order ODEs

A more general-purpose tool for solving linear, homogeneous, second-order ODEs in terms of Bessel functions is presented in Zill³ and shown below in Equation 60.

³ Dennis G Zill. *Advanced Engineering Mathematics*. Jones & Bartlett Learning, 2020

$$u'' + \frac{1-2a}{x}u' + \left(b^2c^2x^{2c-2} + \frac{a^2-p^2c^2}{x^2} \right) u = 0, \quad p \geq 0 \quad (60)$$

The general solution for equations of this form is given in Equation 61.

$$u = x^a [c_1 J_p(bx^c) + c_2 Y_p(bx^c)] \quad (61)$$

Using this tool requires you to solve four non-linear equations as shown below:

$$u'' + \frac{\boxed{1-2a}}{x}u' + \left(\boxed{\frac{b^2c^2}{2}}x^{\boxed{2c-2}} + \frac{\boxed{a^2-p^2c^2}}{x^2} \right) u = 0, \quad p \geq 0$$

Example: Use Equation 60 to find the general solution to the following differential equation:

$$x^2u'' + (x^2 - 2)u = 0$$

Re-writing the equation in the form of Equation 60 gives us:

$$\begin{aligned} x^2u'' + (x^2 - 2)u &= 0 \\ u'' + \frac{x^2-2}{x^2}u &= 0 \\ u'' + 0u' + \left(1x^0 + \frac{-2}{x^2}\right)u &= 0 \end{aligned}$$

Now we solve the four equations:

- ① $1-2a = 0 \Rightarrow a = 1/2$
- ③ $2c-2 = 0 \Rightarrow c = 1$
- ② $b^2c^2 = 1, b^2(1) = 1, \Rightarrow b = 1$
- ④ $a^2-p^2c^2 = -2$

$$\begin{aligned} \left(\frac{1}{2}\right)^2 - p^2(1)^2 &= -2 \\ p^2 &= \frac{1}{4} + 2 = \frac{9}{4} \\ \Rightarrow p &= \frac{3}{2} \end{aligned}$$

Using Equation 61 the general solution is:

$$u(x) = x^{1/2} [c_1 J_{3/2}(x) + c_2 Y_{3/2}(x)]$$

Probably the most challenging or, at least, error-prone part of this process is writing a given ODE in the form of Equation 60.

Assignment #5

Find the general solution to the following differential equations in terms of Bessel Functions:

1. $4x^2u'' + 4xu' + (4x^2 - 25)u = 0$

2. $x^2u'' + xu' + (9x^2 - 4)u = 0$

3. $x^2u'' + xu' - \left(16x^2 + \frac{4}{9}\right)u = 0$

Use the indicated change of variables to find the general solution of the given differential equation.

4. $x^2u'' + 2xu' + \alpha^2x^2u = 0, \quad u = x^{-1/2}v(x)$

Use Equation 6o to find the general solution of the following differential equation in terms of Bessel functions.

5. $xu'' + 2u' + 4u = 0$

Review Problems #1

Solve the following differential equations:

1. $6x^2u'' + 5xu' - u = 0$

2. $x^2u'' - 7xu' + 12u = 0, \quad u(0) = 0, \quad u(1) = 0$

Use the method of power series to solve the following initial value problem:

3. $u'' + xu' + 2u = 0, \quad u(0) = 3, \quad u'(0) = -2$

Use the method of Frobenius to solve the following differential equation:

4. $2xu'' + u' + u = 0$

Find the general solution of the given differential equation in terms of Bessel functions:

5. $4x^2u'' + 4xu' + (64x^2 - 9)u = 0$

Part III

Orthogonal Functions and Fourier Series

Lecture 15 - Introduction to Orthogonal Functions

Objectives

- Define orthogonal functions, weighted orthogonality, function norms, and complete sets of orthogonal functions.
- Provide analogies of these concepts as applied to vectors and functions.

IN PREVIOUS LECTURES we were able to solve some differential equations by representing the solution in the form of an infinite series. For second-order, homogeneous, linear, variable coefficient ODEs where $P(x)$ and $Q(x)$ are analytic throughout the domain of interest, we used power series:

$$u(x) = \sum_{n=0}^{\infty} c_n x^n$$

For equations with regular singular points in the domain of interest, we used the method of Frobenius and expressed the solutions:

$$u(x) = \sum_{n=0}^{\infty} c_n x^{n+r}$$

It should be stressed that, if you calculate the coefficients (c_n) in exact arithmetic and if you sum *all* of the terms ($n \rightarrow \infty$), the representation is *exact*. Each term in the series is linearly independent from all other terms, so as we keep adding terms to the representation of $u(x)$, greater accuracy is achieved.

OUR NEXT IDEA is to generalize this approach by representing our solution, $u(x)$, as a linear combination of *orthogonal functions*.

Inner Product and Orthogonality of Functions

From previous courses in calculus, you should be familiar with the concept of orthogonality of vectors. We test for orthogonality by

Recall that ODEs of this type can be expressed in standard form as:

$$u'' + P(x)u' + Q(x)u = 0$$

Emphasis: For power series and method of Frobenius, each term in the series is a monomial. Going forward we will use series solutions where each term is an orthogonal function.

taking the “dot product” or *inner product*; if the inner product is equal to zero, the vectors are orthogonal, otherwise they are not.

Orthogonality can also be defined for functions. Consider two functions $u_1(x)$ and $u_2(x)$ defined on an interval $x \in [a, b]$. The inner product of $u_1(x)$ and $u_2(x)$ is defined in Equation 62.

$$(u_1, u_2) = \int_a^b u_1(x)u_2(x) dx \quad (62)$$

If $(u_1, u_2) = 0$ then $u_1(x)$ and $u_2(x)$ are said to be *orthogonal* on interval $x \in [a, b]$.

Example: Show that the functions $u_1(x) = x^2$ and $u_2(x) = x^3$ are orthogonal on the interval $x \in [-1, 1]$.

$$\begin{aligned} (u_1, u_2) &= \int_{-1}^1 x^2 x^3 dx \\ &= \int_{-1}^1 x^5 dx = \frac{1}{6}x^6 \Big|_{-1}^1 \\ &= \frac{1}{6} - \frac{1}{6} = 0 \end{aligned}$$

A SLIGHT GENERALIZATION is *weighted* orthogonality, where we apply a *weight function*, $p(x)$, to the inner product:

$$(u_1, u_2) = \int_a^b u_1(x)u_2(x)p(x) dx \quad (63)$$

where if $(u_1, u_2) = 0$ then we say $u_1(x)$ and $u_2(x)$ are orthogonal with respect to weight function $p(x)$.

WE CAN ASSEMBLE a set of orthogonal functions on a specified interval. If $\{\phi_1, \phi_2, \phi_3, \dots, \phi_n\}$ is a set of orthogonal functions on the interval $x \in [a, b]$, then:

$$(\phi_i, \phi_j) = \int_a^b \phi_i \phi_j dx = 0, \text{ if } i \neq j$$

We can then use a linear combination of the members of this set of orthogonal functions to represent practically *any* continuous, or piecewise-continuous function on the interval. This concept will be used *extensively* later in this course.

WHEN DEALING WITH vectors, it is sometimes the case that we want to work with *unit vectors*.¹ Even if we are not in need of unit vectors it is often the case that we need some standard definition of the *size* of a vector. Such a measure is referred to as a *norm*.² Norms are

Vector dot/inner product:

$$\begin{aligned} (\vec{a}, \vec{b}) &= \sum_{i=1}^n (a_i)(b_i) \\ \vec{a} \cdot \vec{b} &= a_1 b_1 + a_2 b_2 + \dots + a_n b_n \end{aligned}$$

¹ Example *unit vectors* include the classic Cartesian basis vectors of $\hat{i} = (1, 0, 0)$, $\hat{j} = (0, 1, 0)$ and $\hat{k} = (0, 0, 1)$.

² A norm is a functional that assigns a measure to a mathematical object like a vector or a function. To qualify as a norm, the functional must satisfy three basic properties:

1. $\|f\| \geq 0$, and $\|f\| = 0$ if and only if $f = 0$
 2. $\|\alpha f\| = |\alpha| \|f\|$ for any constant α ; and
 3. $\|f + g\| \geq \|f\| + \|g\|$
- where f and g are mathematical objects subject to the norm.

denoted $\|\cdot\|$ —i.e. the norm of $f(x)$ is $\|f(x)\|$ —and several types of norms have been defined for vectors, matrices, and functions. The norm we will use for this class is defined in Equation 64.

$$\|f(x)\|^2 = \int_a^b f(x)^2 dx \quad (64)$$

Example: Find the norm of the functions $f_0(x) = 1$ and $f_n(x) = \cos nx$ on the interval $[-\pi, \pi]$.

$$\begin{aligned} \|f_0(x)\|^2 &= \int_{-\pi}^{\pi} (1)(1) dx = x \Big|_{-\pi}^{\pi} \\ &= 2\pi \\ \Rightarrow \|f_0\| &= \sqrt{2\pi} \end{aligned}$$

$$\begin{aligned} \|f_n(x)\|^2 &= \int_{-\pi}^{\pi} \cos^2 nx dx \\ &= \frac{1}{2} \int_{-\pi}^{\pi} (1 + \cos 2nx) dx \\ &= \frac{1}{2}x + \frac{1}{2n} \sin 2nx \Big|_{-\pi}^{\pi} \\ &= \pi \\ \Rightarrow \|f_n\| &= \sqrt{\pi} \end{aligned}$$

Recall the “double-angle” identity:
 $\cos 2x = 2\cos^2 x - 1$.

WE CAN APPLY norms to define *orthonormal* sets of functions, where $\{\phi_0, \phi_1, \dots, \phi_n\}$ are orthonormal if the following is true:

$$(\phi_n, \phi_m) = \begin{cases} 0 & n \neq m \\ 1 & n = m \end{cases}$$

Now, instead of expanding $u(x)$ in a power series or an extended power series, we could expand $u(x)$ in terms of orthonormal functions:

$$u(x) = \sum_{n=0}^{\infty} c_n \phi_n = c_0 \phi_0 + c_1 \phi_1 + \dots$$

were $\phi_n(x)$ are members of an orthogonal set of functions.³ Suppose we wished to expand $u(x)$ in terms of an infinite set of orthogonal functions $\{\phi_0, \phi_1, \dots\}$ on the interval $x \in [a, b]$:

$$u(x) = c_0 \phi_0 + c_1 \phi_1 + c_2 \phi_2 + \dots + c_n \phi_n + \dots$$

This is analogous to a (possibly) familiar operation in vector analysis. Suppose the vector u is expanded as a linear combination of three orthogonal vectors v_1, v_2 , and v_3 :

$$u = c_1 v_1 + c_2 v_2 + c_3 v_3$$

Suppose we know u and know v_1, v_2 , and v_3 ; we merely wish to find the coefficients c_1, c_2 , and c_3 . We can find them by using the inner product for vectors:

$$\begin{aligned} (u, v_1) &= c_1 \overbrace{(v_1, v_1)}^{||v_1||^2} + c_2 \overbrace{(v_2, v_1)}^0 + c_3 \overbrace{(v_3, v_1)}^0 \\ \Rightarrow c_1 &= \frac{(u, v_1)}{||v_1||^2} \end{aligned}$$

Generalizing for all three coefficients:

$$u = \sum_{n=1}^3 \frac{(u, v_n)}{||v_n||^2} v_n$$

³ The orthogonal set of functions may be—in fact, in many cases *is*—infinite as is indicated here.

To get individual coefficient values, c_n , take the inner product—i.e. multiply both sides by the orthogonal function, ϕ_n , and integrate:

$$\begin{aligned}(u, \phi_n) &= \int_a^b u(x)\phi_n(x) dx \\&= \int_a^b c_0\phi_0\phi_n + c_1\phi_1\phi_n + \cdots + c_n\phi_n\phi_n + \cdots dx \\&= c_n\|\phi_n\|^2 \\&\Rightarrow c_n = \frac{(u, \phi_n)}{\|\phi_n\|^2}\end{aligned}$$

Therefore we can construct or expansion as shown in Equation 65.

$$u(x) = \sum_{n=0}^{\infty} \frac{(u, \phi_n)}{\|\phi_n\|^2} \phi_n \quad (65)$$

As was the case with power series and extended power series: subject to some fairly lenient restrictions on $u(x)$, the expansion shown in Equation 65 is *exact*. Sadly, some practical matters will sully this pristine mathematical paradise. The obvious example is that we will not *actually* be able to sum all of the terms and we will not be able to calculate all of the coefficients, c_n , exactly. In particular, we will favor the use of numeric integration to compute the inner products specified in Equation 65.

Note: If ϕ is an orthonormal function, $\|\phi_n\|^2 = 1$.

It takes a while to add an infinite number of terms.

Lecture 16 - Fourier Series

Objectives

- Review trigonometric series.
- Derive/show the formulas for expansion of a function as a trigonometric (Fourier) series.
- Discuss periodic extensions of non-periodic functions, sine/cosine expansions, and convergence behavior.

Review of Fourier Series

In the last lecture we learned about orthogonal functions and sets of orthogonal functions. We stated that most any function can be expressed as a linear combination of those orthogonal functions:

$$u(x) = \sum_{n=0}^{\infty} c_n \phi_n$$

where ϕ_n are members of a set of orthogonal functions and c_n are determined by:

$$c_n = \frac{(u, \phi_n)}{||\phi_n||^2}$$

You should already have experience with expansions such as this from your previous classes in differential equations in the form of Fourier series expansions. In that case the orthogonal functions, $\phi_n(x)$, were:

$$\left\{ 1, \cos \frac{\pi x}{p}, \cos \frac{2\pi x}{p}, \dots, \sin \frac{\pi x}{p}, \sin \frac{2\pi x}{p}, \dots \right\}$$

where p indicates the period.¹

It can be directly shown that members of this set of functions are all mutually orthogonal. We will demonstrate this for the members of the form $\phi_n(x) = \cos n\pi x/p$; other cases are left for homework exercises.

The function $\phi(x) = 1$ could also be written: $\cos \frac{0\pi x}{p}$.

¹ Reminder that a function, $f(x)$, is periodic with period p if $f(x + p) = f(x)$.

Example: Show that functions of the form $\phi_n(x) = \cos \frac{n\pi x}{p}$ are orthogonal over the interval $x \in [-p, p]$:

Consider two functions, $\phi_n(x)$ and $\phi_m(x)$ where m, n are integers and $m \neq n$. The functions are orthogonal on the interval $x \in [-p, p]$ if $(\phi_n, \phi_m) = 0$. From the definition of the inner product:

$$\begin{aligned} (\phi_n, \phi_m) &= \int_{-p}^p \cos \frac{n\pi x}{p} \cos \frac{m\pi x}{p} dx \\ &= \frac{1}{2} \int_{-p}^p \cos(n+m)\frac{\pi x}{p} + \cos(n-m)\frac{\pi x}{p} dx \\ &= \frac{1}{2} \left[\frac{1}{n+m} \frac{p}{\pi} \sin(n+m)\frac{\pi x}{p} \Big|_{-p}^p + \frac{1}{n-m} \frac{p}{\pi} \sin(n-m)\frac{\pi x}{p} \Big|_{-p}^p \right] \\ &= 0 \end{aligned}$$

where the last terms are zero since we are evaluating the sine function at integer multiples of π . This shows, at least, all of the cosine members are orthogonal. For the case $m = n$ we get:

$$\begin{aligned} (\phi_n, \phi_n) &= \int_{-p}^p \cos \left(\frac{n\pi x}{p} \right)^2 dx \\ &= \frac{p}{n\pi} \left[\frac{1}{2} \frac{n\pi x}{p} + \frac{1}{4} \sin \frac{2n\pi x}{p} \right] \Big|_{-p}^p \\ &= \frac{p}{2} - \frac{-p}{2} \\ &= p \end{aligned}$$

WE CAN USE this infinite set of orthogonal functions to represent (nearly) any other continuous function over the interval $[-p, p]$. In your differential equations class you did this by using Equation 66:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos \frac{n\pi x}{p} + b_n \sin \frac{n\pi x}{p} \right] \quad (66)$$

We can solve for the coefficients a_0 , a_n and b_n one at a time by multiplying both sides of Equation 66 by the corresponding orthogonal function, and integrating.² The orthogonal function corresponding to a_0 is 1; so to find a_0 we multiply both sides of Equation 66 by 1 and

Here we use the identity: $\cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta$. So that:

$$\begin{aligned} \cos(\alpha + \beta) &= \cos \alpha \cos \beta - \sin \alpha \sin \beta \\ + \cos(\alpha - \beta) &= \cos \alpha \cos \beta + \sin \alpha \sin \beta \\ &= 2 \cos \alpha \cos \beta \end{aligned}$$

and therefore:

$$\cos \alpha \cos \beta = \frac{1}{2} [\cos(\alpha + \beta) + \cos(\alpha - \beta)].$$

Rather than derive this rigorously, we will combine a tabulated result of standard integrals: $\int \cos^2 u du = \frac{1}{2}u + \frac{1}{4} \sin 2u + C$, with u substitution.

You might be wondering at this point why you would ever want to represent a function $f(x)$ as a linear combination of orthogonal functions. The answer is that the members of the set of orthogonal functions are solutions to a linear homogeneous boundary value problem and the function $f(x)$ will be a boundary condition for a partial differential equation that we are trying to solve.

² In more formal mathematical terms: we take the *inner product* of both sides with respect to the orthogonal function, but of course that means the same thing.

integrate:

$$\begin{aligned}
 f(x) &= \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos \frac{n\pi x}{p} + b_n \sin \frac{n\pi x}{p} \right] \\
 \int_{-p}^p f(x)(1) dx &= \int_{-p}^p \frac{a_0}{2}(1) dx + \underbrace{\int_{-p}^p \sum_{n=1}^{\infty} \left[a_n \cos \frac{n\pi x}{p} + b_n \sin \frac{n\pi x}{p} \right] (1) dx}_{=0 \text{ due to orthogonality}} \\
 \int_{-p}^p f(x) dx &= \frac{a_0}{2} 2p + 0 \\
 \Rightarrow a_0 &= \frac{1}{p} \int_{-p}^p f(x) dx
 \end{aligned}$$

To get the value of a_1 , we multiply both sides by $\cos \frac{\pi x}{p}$ and integrate:

$$\begin{aligned}
 \int_{-p}^p f(x) \cos \frac{\pi x}{p} dx &= \frac{a_0}{2} \int_{-p}^p (1) \cos \frac{\pi x}{p} dx + \dots \\
 &\quad \underbrace{a_1 \int_{-p}^p \cos \frac{\pi x^2}{p} dx}_{=p} + b_1 \int_{-p}^p \sin \frac{\pi x}{p} \cos \frac{\pi x}{p} dx + a_2 \int_{-p}^p \cos \frac{2\pi x}{p} \cos \frac{\pi x}{p} dx + \dots
 \end{aligned}$$

Solving for a_1 we get:

$$\begin{aligned}
 \int_{-p}^p f(x) \cos \frac{\pi x}{p} dx &= a_1 p \\
 \Rightarrow a_1 &= \frac{1}{p} \int_{-p}^p f(x) \cos \frac{\pi x}{p} dx
 \end{aligned}$$

We repeat the process for b_1 by multiplying both sides by $\sin \frac{\pi x}{p}$.

In general, for a_n we use $\cos n\pi x/p$ and for b_n , $\sin n\pi x/p$. The resulting formulas for the coefficients are given in Equation 67.

$$\begin{aligned}
 a_0 &= \frac{1}{p} \int_{-p}^p f(x) dx \\
 a_n &= \frac{1}{p} \int_{-p}^p f(x) \cos \frac{n\pi x}{p} dx \\
 b_n &= \frac{1}{p} \int_{-p}^p f(x) \sin \frac{n\pi x}{p} dx
 \end{aligned} \tag{67}$$

Since this is an infinite series, we need to concern ourselves with convergence. The theorem below provides us assurance of convergence for continuous and piece-wise continuous functions on the interval $[-p, p]$.

Theorem 8 (Convergence of Fourier Series)

If f and df/dx are piece-wise continuous on an interval $[-p, p]$, then for all x in the interval $[-p, p]$ the Fourier series converges to f at points where the function is continuous; at points of discontinuity, the Fourier series converges to:

$$\frac{f(x^-) + f(x^+)}{2}$$

where $f(x^-)$ and $f(x^+)$ denote the limit of $f(x)$ from the left and right at the point of discontinuity.

In the next few lectures we will define other orthogonal function expansions similar to the Fourier series. Nonetheless, for periodic functions defined on a finite interval, the Fourier series provides the best representation of a function. There are some special cases, however, where we can take advantage of structural properties of $f(x)$ to reduce the amount of work we need to do in carrying out the Fourier series expansions.

Even Functions and Odd Functions

When doing a Fourier series expansion it is sometimes helpful to consider whether a function is *even* or *odd*.

Definition 14 (Even Function)

A function is even if, for all real values x , $f(-x) = f(x)$.

An example of an even function is shown in Figure 6.

Definition 15 (Odd Function)

A function is odd if, for all real values x , $f(-x) = -f(x)$.

An example of an odd function is shown in Figure 7.

SOME PROPERTIES of even and odd functions include:³

1. An even function times an even function results in an even function.
2. An odd function times an odd function results in an even function.
3. An even function times an odd function results in an odd function.
4. Adding or subtracting two even functions results in an even function.
5. Adding or subtracting two odd functions results in an odd function.
6. $\int_{-p}^p f_{\text{even}}(x) dx = 2 \int_0^p f_{\text{even}}(x) dx$ and $\int_{-p}^p f_{\text{odd}}(x) dx = 0$.

In coming lectures and when doing assignments you will see that issues of continuity of f and df/dx have obvious visible influence on the convergence behavior of Fourier series.

When we say “best” representation, we (more or less) mean two things:

1. $\|\tilde{f}_n - f\|$, where \tilde{f}_n is the power series representation of f up to n terms, gets smaller with fewer terms than other series expansions; and
2. calculation of the coefficients a_n and b_n can be carried out with greater numeric stability than for other expansions.

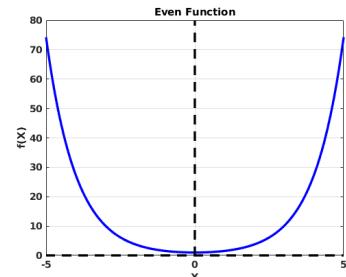


Figure 6: An example even function.

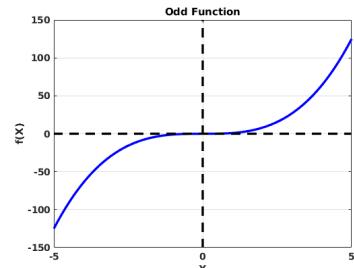


Figure 7: An example odd function.

³ Students are welcome to prove these assertions.

The “even-ness” or “odd-ness” of a function is relevant to Fourier series expansions. If you expand an *even* function in a Fourier series you will find that all of the b_n coefficients are zero. If you expand an *odd* function in a Fourier series you will find that a_0 and a_n terms are all zero.

You can still use the formulas presented in Equation 67 when expanding even or odd functions. Alternatively, you can use the formulas for the Cosine expansion or Sine expansion below for even or odd functions respectively.

Cosine series:

$$\begin{aligned} f(x) &= \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos \frac{n\pi x}{p} \\ a_0 &= \frac{2}{p} \int_0^p f(x) dx \\ a_n &= \frac{2}{p} \int_0^p f(x) \cos \frac{n\pi x}{p} dx \end{aligned} \quad (68)$$

Sine series:

$$\begin{aligned} f(x) &= \sum_{n=1}^{\infty} b_n \sin \frac{n\pi x}{p} \\ b_n &= \frac{2}{p} \int_0^p f(x) \sin \frac{n\pi x}{p} dx \end{aligned} \quad (69)$$

The cosine and sine series expansions are sometimes referred to as “half-wave” expansions since the calculations, as shown in the formulas, only involve the portion of the wave in the interval $[0,p]$ —the positive half-wave.

Lecture 17 - Generating and Plotting Fourier Series in MATLAB

Objectives

- Demonstrate how to carry out Fourier series expansions using MATLAB.
- Give a qualitative demonstration of convergence behavior of Fourier series.
- Demonstrate Cosine and Sine series expansions.

In this lecture, we will illustrate the process of Fourier series expansions with three examples.

Example #1: Carry out the Fourier series expansion of the function given in Equation 70, illustrated in Figure 8:

$$f(x) = \begin{cases} 0, & -\pi \leq x < 0 \\ \pi - x, & 0 \leq x \leq \pi \end{cases} \quad (70)$$

We wish to represent this function as:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos \frac{n\pi x}{p} + b_n \sin \frac{n\pi x}{p} \right]$$

where, in this case, $p = \pi$. Even though we are only interested in the function in the interval $[-\pi, \pi]$, since the Fourier series represents the function in terms of a constant and an infinite linear combination of periodic functions, we should think of the function that we are representing as periodic.¹

We have everything we need; it is just a matter of calculating the coefficients from Equation 67. Rather than carrying out the calculations with pencil and paper we will use MATLAB. In the listing below we will step through the code necessary to calculate Fourier coefficients through $N=5$.

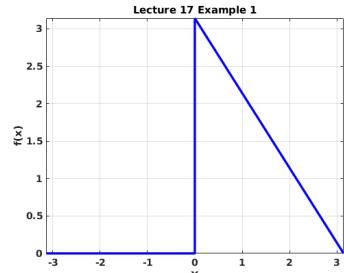


Figure 8: Example #1 $f(x)$.

¹ This outlook will help us understand the convergence behavior of the Fourier series; especially at the boundaries.

```

1 clear
2 clc
3 close 'all'

4 N = 5; % specify number of coefficients
5
6 f = @(x) ex1(x);
7 p = pi; % specifiy period
8

```

We start, as always, by clearing out the workspace memory and command-prompt output and closing any open figures. We also need to represent $f(x)$ in MATLAB; we will do this with a local function named $\text{ex1}(x)$.²

The integrals needed to determine the Fourier coefficients will be evaluated numerically with the MATLAB built-in function `integral()`.³ Let us start with a_0 which, as a reminder, is computed by:

$$a_0 = \frac{1}{p} \int_{-p}^p f(x) dx$$

```

9 ao = (1/p)*integral(f,-p,p);
10 FF = @(x) ao/2;

```

The first line of the listing numerically evaluates a_0 ; the second line creates an anonymous function and initializes it to the first term in the Fourier expansion.

We will use a loop to construct the remaining terms in the Fourier expansion.

```

11 for n = 1:N
12     an = (1/p)*integral(@(x) f(x).*cos(n*pi*x/p),-p,p);
13     bn = (1/p)*integral(@(x) f(x).*sin(n*pi*x/p),-p,p);
14     FF = @(x) FF(x) + an*cos(n*pi*x/p) + bn*sin(n*pi*x/p);
15 end

```

Note in the last line where we append the newly computed terms to the Fourier series expansion $\text{FF}(x)$. Now we have a function, $\text{FF}(x)$, that represents the Fourier series expansion with $N = 5$ terms. In the next listing we add the code to plot the function and verify that it makes sense.

```

16 Nx = 1000;
17 X = linspace(-p,p,Nx);
18
19 plot(X,f(X),'-b',...
20      X,FF(X),'--r',...
21      'LineWidth',3)
22 title_str = sprintf('Example 1, n = %d',n);❶
23 title(title_str,'FontSize',16,...
24      'FontWeight','bold');
25 xlabel('X','FontSize',14,... ❷
26      'FontWeight','bold');
27 ylabel('f(X)','FontSize',14,...
28      'FontWeight','bold');

```

² Since local functions must appear after all of the other code in a MATLAB script file, the code for $\text{ex1}(x)$ will be presented last.

³ This function has default signature `Q = integral(FUN,A,B)` and it approximates the integral of the function `FUN` over the interval `A` to `B` using global adaptive quadrature. The error tolerances for this numeric integration algorithm can be specified by the user; in most cases we will use default values. There are several standard algorithms for numerically computing integrals—called quadrature—that the interested student can read about in the numerical methods portion of this text.

Recall:

$$a_n = \frac{1}{p} \int_{-p}^p f(x) \cos \frac{n\pi x}{p} dx$$

$$b_n = \frac{1}{p} \int_{-p}^p f(x) \sin \frac{n\pi x}{p} dx$$

Note how you can practically read the equation directly from the MATLAB code.

Referring to the annotations:

❶ Using `sprintf()` allows us to combine the variable `n` in the title string.

❷ Optional name-value pairs such as '`LineWidth`',³ '`FontSize`',¹⁶, and '`FontWeight`',^{'bold'} help make the plot and labels more readable.

```

grid on
legend('f(x)', 'FF(x)')❸
set(gca, 'FontSize', 12, ...
    'FontWeight', 'bold');❹

```

The resulting Fourier series expansion is shown in Figure 9. If we want to increase the number of Fourier series terms in the expansion, we need only change N. Figure 10 shows the series expansion with N=15 terms.

Some things to note about the resulting Fourier series representation of $f(x)$:

1. As n increases, FF(x) generally “looks more like” $f(x)$.
2. At the discontinuity in $f(x)$, the Fourier series representation appears to be converging on the midpoint between $f(x^-)$ and $f(x^+)$ as the theory says it should.
3. The Fourier series representation near the point of discontinuity has “wigginess” that doesn’t go away as n increases.
4. In particular, note the undershoot and overshoot of $f(x)$ to the left and right respectively of $f(0)$. This is called “Gibbs phenomena” and it does not go away as n increases but it moves closer to the point of discontinuity.

As Figure 11 shows, as N is increased, we can make FF(x) arbitrarily close to $f(x)$ with the exception of the perturbations at the point of discontinuity.

AN IMPORTANT MATTER that we have not yet dealt with is how to represent piece-wise continuous functions like $f(x)$ in MATLAB.⁴ As stated previously, we will use a *local function* to do this. The code is shown in the listing below.

```

%% Local functions
function y = ex1(x)
[m,n] = size(x); ❶
y = nan(m,n); ❷
for i = 1:length(x) ❸
    if (x(i) >= -pi) && (x(i) < 0)
        y(i) = 0;
    elseif (x(i) >= 0) && (x(i) <= pi) ❹
        y(i) = pi - x(i);
    end
end
end

```

Some notes on the annotations for this listing:

- ❶ We use the MATLAB built-in function `size()` to get the dimensions of the input vector. The return values [m,n] give the number of rows

❸ Make a habit of using legends for graphs that include multiple data series. Once again, this makes the plot more readable.

❹ The argument `gca` means “get current axis.” Calling the `set()` function with name-value pairs `'FontSize',10` and `'FontWeight','bold'` sets the font size and weight for the axis markings.

In general it is important that your plots look good.

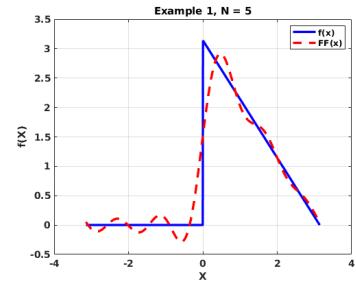


Figure 9: Fourier series expansion with N=5.

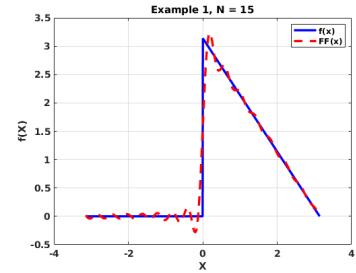


Figure 10: Fourier series expansion with N=15.

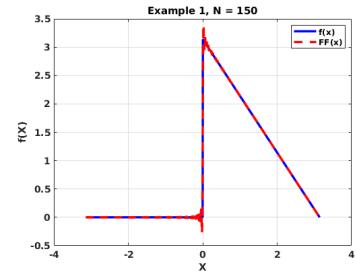


Figure 11: Fourier series expansion with N=150.

⁴ For whatever reason, piece-wise continuous functions are intensively used in *textbooks* on partial differential equations—I am not so sure that they are as important in real-life applications.

and columns of x respectively. For this function we are implicitly expecting x to be a vector, but it can be either a row-vector or a column vector.

② We construct the output vector y using the `nan()` function. This line makes the vector y the same size and shape as the input vector x .

③ We use the built-in function `length()` to get the number of elements of x . This is a bit of a hack since, if x were *not* a vector, `ex1(x)` would no longer work properly.⁵

④ The symbol `&&` is “element-wise and.” Pay attention to use of `>=` and `<=` operators to get the details of the intended function correct.

Example #2: Carry out the Fourier series expansion of the function given in Equation 71, illustrated in Figure 12.

$$f(x) = \begin{cases} -1, & -\pi \leq x < 0 \\ 1, & 0 \leq x \leq \pi \end{cases} \quad (71)$$

Fourier series expansions of this function are shown in Figures 13 through 15.

Some notes:

1. As with the first example, the function has the Gibbs phenomena near the discontinuity at $x = 0$.
2. Also, as with the first example, the Gibbs phenomena does not go away as N increases, but it gets more “peaked” and closer to the origin.
3. Unlike the first example, we get the Gibbs phenomena and wiggliness at the ends also. This is because the Fourier series representation is periodic; the periodic extension of this function has discontinuities at the endpoints since $f(-\pi) \neq f(\pi)$.
4. You should also note that this function is *even*. That means we expect a_0 and all values of a_n to be equal to zero. If we modify the for-loop to output values for the a_n coefficients we get all zeros.

```

for n = 1:N
    an = (1/p)*integral(@(x) f(x).*cos(n*pi*x/p),-p,p);
    fprintf('a_%d = %g\n',n,an);
    bn = (1/p)*integral(@(x) f(x).*sin(n*pi*x/p),-p,p);
    FF = @(x) FF(x) + an*cos(n*pi*x/p) + bn*sin(n*pi*x/p);
end

```

⁵ It would be a good idea to verify that the input x is actually a vector. MATLAB, like most other languages, includes features to enforce assumptions like this. The code: `assert(min(size(x)) == 1, 'x must be a vector')` would raise an error in MATLAB if the minimum dimension of x is anything other than 1. That would be one way to ensure x is a vector.

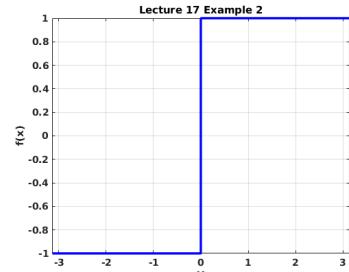


Figure 12: Example #2 $f(x)$.

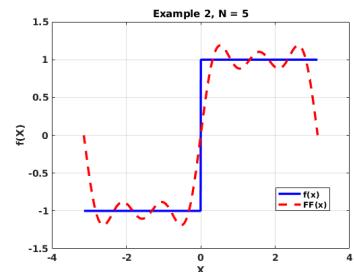


Figure 13: Fourier series expansion with $N=5$.

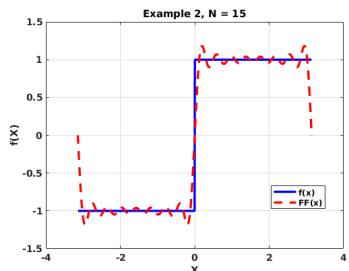


Figure 14: Fourier series expansion with $N=15$.

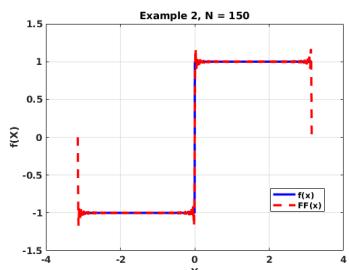


Figure 15: Fourier series expansion with $N=150$.

Example #3: Construct the Fourier series expansion of the function given in Equation 72.

$$f(x) = x^2, \quad x \in [0, 2] \quad (72)$$

This function is not periodic and, unlike the previous examples, does not even span a symmetric interval about the origin. In this case we will still use the same Fourier series formulas but we will construct a “reflection” about the origin. This reflection can be *even-*, *odd-*, or it can be an *identity-reflection* with respect to the y-axis; these correspond to the Cosine expansion, Sine expansion and the full Fourier series expansions. These different expansion options are shown in Figure 16.

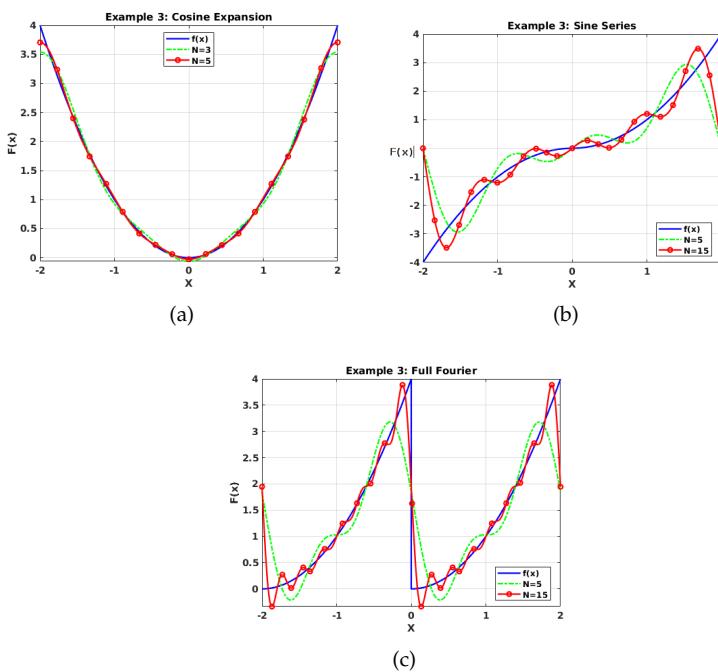


Figure 16: Even-, odd- and identity-reflection for $f(x) = x^2$.

Note that the convergence behavior for the Fourier expansion is different for each case.

- For the even-reflection Cosine expansion convergence is very rapid. Both $f(x)$ and $f'(x)$ are continuous throughout the interval $x \in (-2, 2)$. The function itself is continuous at the end-points but notice that the derivative is not. If you were to draw an additional period on the left and right-hand side of the cosine expansion, $f'(x)$ would have a discontinuity; that explains the (relatively) poor convergence of the series at the end-points.
- For the odd-reflection Sine expansion the function and derivative are continuous throughout the domain. The derivative of the func-

As you can see, in cases where you can choose which expansion you use, some choices are good and some are bad. As we will see in coming chapters, we often do not have a choice in which set of orthogonal functions we will use to do our expansion so, sadly, we cannot pick one that we think will be best. What we *can* do is analyze the expansion that we *do* get and understand the convergence behavior by examining the continuity of the functions and derivatives of functions that we are representing.

tion is continuous at the periodic end-points but $f(x)$ itself is not. This explains why the Sine series expansion converges to zero at both end-points.

- The identity-reflection full Fourier series has discontinuities in $f(x)$ and $f'(x)$ at both endpoints and at $x = 0$. The convergence behavior is correspondingly bad.

Assignment #6

Show that the given functions are orthogonal on the given interval.

1. $f_1(x) = e^x, f_2(x) = xe^{-x} - e^{-x}, \quad x \in [0, 2]$

2. $f_1(x) = x, f_2(x) = \cos 2x, \quad x \in [-\pi/2, \pi/2]$

Show that the given set of functions is orthogonal on the indicated interval. Find the norm of each function in the set.

3. $\{\sin x, \sin 3x, \sin 5x, \dots\}, \quad x \in [0, \pi/2]$

Use MATLAB to verify by numeric integration that the functions are orthogonal with respect to the indicated weight function on the given interval.

4. $H_0(x) = 1, H_1(x) = 2x, H_2(x) = 4x^2 - 2; \quad w(x) = e^{-x^2}, \quad x \in (-\infty, \infty)$

Note: use the built-in function `integral()` to do the numeric integration. In MATLAB, $-\infty$ and ∞ are represented by `-inf` and `inf` respectively.

Use MATLAB to construct the Fourier series expansion of the given function $f(x)$ on the given interval.

For each problem create a plot that shows: a) $f(x)$ along with b) the truncated Fourier series of $f(x)$ with $N=5$ and c) $N=15$ terms. Also give the number to which the Fourier series expansion converges at any point(s) of discontinuity in $f(x)$.

5. $f(x) = \begin{cases} 0, & -\pi < x < 0 \\ 1, & 0 \leq x < \pi \end{cases}$

$$6. f(x) = \begin{cases} 0, & -\pi < x < 0 \\ x^2, & 0 \leq x < \pi \end{cases}$$

$$7. f(x) = \begin{cases} 0, & -2 < x < 0 \\ -2, & -1 \leq x < 0 \\ 1, & 0 \leq x < 1 \\ 0, & 1 \leq x < 2 \end{cases}$$

Determine whether the given function is even, odd, or neither.

$$8. f(x) = x^2 + x$$

$$9. f(x) = \begin{cases} x^2, & -1 < x < 0 \\ -x^2, & 0 \leq x < 1 \end{cases}$$

Use MATLAB to expand the given function in an appropriate cosine or sine series. For each function create a plot showing: a) $f(x)$ along with b) the truncated Fourier series of $f(x)$ with $N=5$ and; c) $N=15$ terms.

$$10. f(x) = |x|, \quad -\pi < x < \pi$$

$$11. f(x) = \begin{cases} x - 1, & -\pi < x < 0 \\ x + 1, & 0 \leq x < \pi \end{cases}$$

$$12. f(x) = \begin{cases} 1, & -2 < x < -1 \\ -x, & -1 \leq x < 0 \\ x, & 0 \leq x < 1 \\ 1, & 1 \leq x < 2 \end{cases}$$

$$13. f(x) = \begin{cases} x, & 0 < x < \pi/2 \\ \pi - x, & \pi/2 \leq x < \pi \end{cases}$$

Lecture 18 - Sturm-Liouville Problems

Objectives

- Define regular/singular Sturm-Liouville eigenvalue problems and give properties of their solutions.
- Do an example problem for finding eigenvalues and eigenfunctions.
- Do an example problem for transforming a linear, second-order, homogeneous boundary value problem into self-adjoint form.

Regular Sturm-Liouville Eigenvalue Problem

We have solved several differential equations in this class. All of the boundary value problems that we have solved so far are special cases of a more general framework called Sturm-Liouville eigenvalue problems. That is what we wish to discuss in this lecture.

For the regular Sturm-Liouville Eigenvalue problem we hope to solve:

$$\frac{d}{dx} [r(x)u'] + (q(x) + \lambda p(x))u = 0, \quad x \in (a, b) \quad (73)$$

subject to the boundary conditions:

$$A_1 u(a) + B_1 u'(a) = 0, \text{ where } A_1, \text{ and } B_1 \text{ are not both zero.}$$

$$A_2 u(b) + B_2 u'(b) = 0, \text{ where } A_2, \text{ and } B_2 \text{ are not both zero.}$$

Note that these boundary conditions are referred to as *homogeneous*. The same rule that we use to decide if a differential equation is homogeneous apply in the same way to the boundary conditions. For a boundary value problem to be homogeneous, *both* the differential equation and boundary conditions must be homogeneous.

Note: for Equation 73, $r(x)$, $r'(x)$, $q(x)$, and $p(x)$ must be real-valued and continuous on the interval $x \in (a, b)$. Also $p(x) > 0$ and $r(x) > 0$ for all $x \in (a, b)$. These are important conditions that should be verified each time you encounter a new problem. The constant λ is referred to as an *eigenvalue*.

For ODEs that we solved in earlier lectures, we routinely dealt with problems having non-homogeneous boundary conditions. As we go forward to solve linear partial differential equations using separation of variables, it will be *essential* that the boundary conditions are homogeneous. So you should be sure that you know how to check/verify that condition.

Properties of the Regular Sturm-Liouville problem:

1. There exist an infinite number of real eigenvalues that can be arranged in increasing order. (e.g. $\lambda_1 < \lambda_2 < \lambda_3 < \dots < \lambda_n < \dots$)
2. For each eigenvalue, λ_n , there is exactly one eigenfunction, $u_n(x)$, that is a solution to the problem.
3. Eigenfunctions corresponding to different eigenvalues are linearly independent.
4. The set of eigenfunctions is orthogonal with respect to $p(x)$ on the interval $[a, b]$. In other words: $\int_a^b u_n(x)u_m(x)p(x) dx = 0$ if $n \neq m$.
5. The set of eigenfunctions is complete on the interval $[a, b]$. In other words, for any (reasonable) $f(x)$, we can represent $f(x)$ as a linear combination of those eigenfunctions: $f(x) = \sum_{n=0}^{\infty} c_n u_n(x)$.¹

If $r(x)$ in Equation 73 is zero at either boundary, the problem is said to be a singular boundary value problem. If $r(a) = r(b)$, with suitable boundary conditions, the problem is said to be a periodic boundary value problem.

Example: Find the eigenvalues and eigenfunctions of the following boundary value problem:

Equation:	$u'' + \lambda u = 0, \quad x \in [0, 1]$
BCs:	$u(0) = 0, \quad u(1) + u'(1) = 0$

To fully analyze this problem we will have to consider three cases for λ : $\lambda < 0$, $\lambda = 0$, and $\lambda > 0$.

$\lambda = 0$: In this case, the differential equation reduces to:

$$u'' = 0$$

with general solution: $u(x) = c_1(x) + c_2$. If we apply the boundary condition $u(0) = 0$, this implies that $u(0) = c_1(0) + c_2 = c_2 = 0$. So the solution is simplified to $u(x) = c_1(x)$. The second boundary condition: $u(1) + u'(1) = c_1(1) + c_1 = 2c_1 = 0 \Rightarrow c_1 = 0$. The only solution that satisfies the equation and boundary conditions for $\lambda = 0$ is the trivial solution $u(x) = 0$.²

¹ Another way of saying this is that no function, $f(x)$, can be orthogonal to *all* of the eigenfunctions, $u_n(x)$, on the interval $[a, b]$.

To obtain values for the coefficients, c_n , we need only take the inner product with the corresponding eigenfunction, u_n . i.e. multiply both sides by an orthogonal function and integrate.

Note that this problem is not presented in self-adjoint form. Have faith that it is, indeed, a Sturm-Liouville eigenvalue problem and could be presented in self-adjoint form. We will practice making this transformation later in the lecture.

² The trivial solution, $u(x) = 0$ will always satisfy a homogeneous boundary value problem and, in general, is of little interest to us. What we take from this part of the analysis is that we will rule out $\lambda = 0$ since there are no *interesting* solutions in that case.

$\lambda < 0$: For this case we will assume $\lambda = -\alpha^2$ where $\alpha > 0$. The differential equation reduces to:

$$u'' - \alpha^2 u = 0$$

This equation has the general solution of:

$$u(x) = c_1 e^{-\alpha x} + c_2 e^{\alpha x}$$

or:

$$u(x) = c_1 \cosh \alpha x + c_2 \sinh \alpha x$$

Since this problem is posed on a bounded interval, we will choose the second form above. Applying the first boundary condition gives us: $u(0) = c_1 \cosh 0 + c_2 \sinh 0 = c_1(1) + c_2(0) = 0 \Rightarrow c_1 = 0$. Applying the second boundary condition to the current solution gives us: $u(1) + u'(1) = c_2 \sinh 1 + c_2 \cosh 1 = 0$.

We recall that both $\sinh x$ and $\cosh x$ are strictly positive on $x \in (0, 1)$ so the only way the second boundary condition can be met is for $c_2 = 0$. Consequently only the trivial solution, $u(x) = 0$, satisfies the governing equation and boundary conditions for the case that $\lambda < 0$.

$\lambda > 0$: For this case we will assume $\lambda = \alpha^2$ where $\alpha > 0$. The differential equation reduces to:

$$u'' + \alpha^2 u = 0$$

This equation has the general solution of $u(x) = c_1 \cos \alpha x + c_2 \sin \alpha x$. Applying the first boundary condition gives us: $u(0) = c_1 \cos 0 + c_2 \sin 0 = c_1(1) + c_2(0) = 0 \Rightarrow c_1 = 0$. Applying the second boundary condition to the current solution gives us: $u(1) + u'(1) = c_2 \sin \alpha + \alpha c_2 \cos \alpha = 0$, or:

$$u(x) = c_2 [\sin \alpha + \alpha \cos \alpha] = 0 \quad (74)$$

This equation can be satisfied simply by setting $c_2 = 0$, but we will resist that temptation since that would then imply that there are *no* values of λ that admit a non-trivial solution for this problem. Instead we will look for values of α such that:

$$\sin \alpha + \alpha \cos \alpha = 0 \quad (75)$$

We can see from Figure 17 that there are values of α that satisfy this condition.³ We will denote these eigenvalues $\alpha_1^2 = \lambda_1, \alpha_2^2 = \lambda_2, \dots, \alpha_n^2 = \lambda_n$ and the corresponding eigenfunctions are denoted: $u_n(x) = \sin \alpha_n x$.

Recall that these two solutions are equivalent. We will generally use the first form on *unbounded* intervals; the second form on *bounded* intervals.

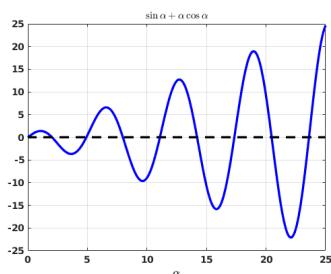
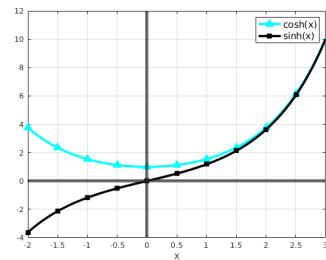


Figure 17: Plot of $\sin \alpha + \alpha \cos \alpha$.

³ We probably should not assume as much by looking at the plot in Figure 17 but it turns out that there are infinitely many zeros.

WE WILL DEFER, for the moment, the problem of finding the roots to Equation 75. Suffice it to say that there are infinitely many distinct roots yielding the infinitely many eigenvalues to go with the infinitely many eigenfunctions. They can be found with a non-linear equation solver (“root-finder”) for which there are several reliable algorithms.

Transforming Equations to Self-Adjoint Form

Apart from sharing some theoretical tid-bits regarding Sturm-Liouville eigenvalue problems, the *point* of this lecture is to highlight: a) the eigenfunctions that solve the eigenvalue problem; and b) their property of weighted orthogonality. Recalling the last two lectures where we used an infinite set of trigonometric functions for functional expansion in a Fourier series, we will want to use *other* functions for such expansions. Those other functions will be the set of eigenfunctions associated with a Sturm-Liouville eigenvalue problem.

As previously mentioned, the eigenfunction solutions are linearly independent and orthogonal with respect to weight function $p(x)$. We need to know what that weight function is in order to carry out an orthogonal function expansion like Fourier series.

CONSIDER THE LINEAR, homogeneous, second-order boundary value problem shown in Equation 76:

$$a(x)u'' + b(x)u' + (c(x) + \lambda d(x))u = 0 \quad (76)$$

where $a(x) \neq 0$ and $a(x)$, $b(x)$, $c(x)$, and $d(x)$ are continuous. We will convert to the self-adjoint form: $\frac{d}{dx} [r(x)u'] + [q(x) + \lambda p(x)]u = 0$ by determining the functions $r(x)$, $q(x)$, and $p(x)$ as follows:

$$1. \ r(x) = e^{\int b(x)/a(x) dx}$$

$$2. \ q(x) = \frac{c(x)}{a(x)}r(x)$$

$$3. \ p(x) = \frac{d(x)}{a(x)}r(x)$$

The sines and cosines used in Fourier series fit within this theory. It turns out that the weight function $p(x)$ in that case is $p(x) = 1$.

Example: Express the following equation, which has solutions $P_n(x)$ in self-adjoint form and give the orthogonality relation.

$$\underbrace{(1-x^2)}_{a(x)} u'' \underbrace{-2x u'}_{b(x)} + \underbrace{\lambda n(n+1)}_{c(x)} u = 0, \quad x \in (-1, 1)$$

From the equation, $a(x)$ and $b(x)$ are annotated; $c(x) = 0$ and $d(x) = 1$. We first compute $r(x)$:

$$\begin{aligned} r(x) &= e^{\int \frac{-2x}{(1-x^2)} dx} \\ &= e^{\int \frac{1}{u} du} \\ &= e^{\ln u} \\ &= u \\ &= 1-x^2 \end{aligned}$$

Now we compute $q(x)$:

$$\begin{aligned} q(x) &= \frac{c(x)}{a(x)} r(x) \\ &= \frac{0}{(1-x^2)} (1-x^2) \\ &= 0 \end{aligned}$$

Then $p(x)$:

$$\begin{aligned} p(x) &= \frac{d(x)}{a(x)} r(x) \\ &= \frac{1}{(1-x^2)} (1-x^2) \\ &= 1 \end{aligned}$$

Therefore the boundary value problem in self-adjoint form is:

$$\frac{d}{dx} \left[(1-x^2) u' \right] + \lambda_n u = 0 \quad (77)$$

where $\lambda_n = n(n+1)$. As given in the problem statement, the eigenfunctions are $P_n(x)$ and the weight function $p(x) = 1$. The orthogonality relation is:

$$(P_m, P_n) = \int_{-1}^1 P_m(x) P_n(x) (1) dx = \begin{cases} 0, & m \neq n \\ \frac{2}{2n+1}, & m = n \end{cases}$$

This is Legendre's equation that we solved in a previous lecture. $P_n(x)$ is standard notation for Legendre polynomials of order n .

Here we use a u -substitution:

$$\begin{aligned} u &= (1-x^2) \\ du &= -2x dx \\ \text{so } e^{\int \frac{-2x}{(1-x^2)} dx} &= e^{\int \frac{1}{u} du} \text{ following this substitution.} \end{aligned}$$

Note: You will not be expected to know, by inspection, the value of (P_n, P_n) but it is provided here for your information.

Lecture 19 - Fourier-Bessel Series Expansions

Objectives

- Present the parametric Bessel equation as a Sturm-Liouville problem and derive the orthogonality relation.
- Do an example to show a Fourier-Bessel expansion of a function.
- Demonstrate use of the MATLAB function `besselzero()`.

Parametric Bessel Equation

The parametric Bessel equation is a second-order linear, homogeneous differential equation that also fits within Sturm-Liouville theory. As a reminder, the equation is:

$$x^2 u'' + xu' + (\alpha^2 x^2 - \nu^2) u = 0$$

and the general solution is given by:

$$u(x) = c_1 J_\nu(\alpha x) + c_2 Y_\nu(\alpha x)$$

The solutions, $J_\nu(\alpha x)$ and $Y_\nu(\alpha x)$ are, of course, linearly independent but they also are orthogonal with respect to some weight function $p(x)$. We can use them to construct an orthogonal function expansion in exactly the same way we did with Fourier series. That is what we will do in this lecture. To accomplish this we want to put the parametric Bessel equation in self-adjoint form and we will proceed in this effort just as we did in the last lecture.

Let us first put the parametric Bessel equation in standard form:

$$\begin{aligned} a(x)u'' + b(x)u' + [c(x) + \lambda d(x)] u &= 0 \\ x^2 u'' + xu' + [-\nu^2 + \alpha^2 x^2] u &= 0 \end{aligned}$$

so, $a(x) = x^2$, $b(x) = x$, $c(x) = -\nu^2$, and $d(x) = x^2$.

It may not be clear immediately that λ corresponds to values of α but that is the correct inference; when we do the orthogonal function expansion with Bessel functions it will be more clear why that is the case.

Next we will compute $r(x)$:

$$\begin{aligned} r(x) &= e^{\int \frac{b(x)}{a(x)} dx} \\ &= e^{\int \frac{x}{x^2} dx} \\ &= e^{\int \frac{1}{x} dx} \\ &= e^{\ln x} \\ &= x \end{aligned}$$

Now we compute $q(x)$:

$$\begin{aligned} q(x) &= \frac{c(x)}{a(x)} r(x) \\ &= \frac{-\nu^2}{x^2} x \\ &= -\frac{\nu^2}{x} \end{aligned}$$

Then $p(x)$:

$$\begin{aligned} p(x) &= \frac{d(x)}{a(x)} r(x) \\ &= \frac{x^2}{x^2} x \\ &= x \end{aligned}$$

So the self-adjoint form of the parametric Bessel equation is:

$$\frac{d}{dx} [xu'] + \left(-\frac{\nu^2}{x} + \alpha^2 x \right) u = 0$$

The corresponding orthogonality relation is shown in Equation 78:

$$\int_a^b J_\nu(\alpha_n x) J_\nu(\alpha_m x) x dx = 0, \quad n \neq m \quad (78)$$

where a and b are the bounds of the interval on which orthogonality is expressed.

Example: Expand $f(x) = x$, $0 < x < 3$, in a Fourier-Bessel series, using Bessel functions of order $\nu = 1$ that satisfy the boundary condition $J_1(3\alpha) = 0$.

So what we want is:

$$f(x) = x = \sum_{n=1}^{\infty} c_n J_1(\alpha_n x)$$

Note that we omit Bessel functions of the second kind, $Y_n(x)$, because as is shown in the figure, they diverge to negative infinity as x goes

Admittedly, the real reason why we want to do this is to obtain the weight function $p(x)$ which, in this case is $p(x) = x$.

Like other Sturm-Liouville problems we will find that there are infinitely many distinct eigenvalues, λ_n , which for this equation we will refer to as α_n . Note the weight function x now appears in the inner product.

Remember that it is the *boundary conditions* that allow us to determine the eigenvalues.

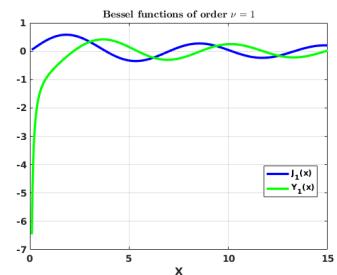


Figure 18: Bessel functions of order 1.

to zero. This *implicit boundary condition*, where one solution of the differential equation diverges at the problem boundary, often needs to be considered when solving boundary value problems.

The other boundary condition applies at $x = 3$: $J_1(3\alpha_n) = 0$ or, put differently, we select the values of α_n such that $3\alpha_n$ is a root of $J_1(x)$. While our plot of $J_1(x)$ does not extend out to infinity, it turns out that $J_1(x)$ has infinitely many roots and we need to find them.

Interlude on Open-Source Software

At some point in time in your life as an engineer it is inevitable that a problem will arise that you are not prepared to tackle yourself. The tools you have been given to do your job do not fully answer to the task at hand. This is one such occasion. We need to find the roots of $J_1(x)$. You know those values exist but you do not know what they are and it turns out that MATLAB does not (at this time) have any built-in functions to give you the roots of such functions either.¹

Some options available to you include:

1. Go to the library and check out a book that tabulates some roots of $J_1(x)$ —possibly also including scads of additional Bessel function lore²—and enter the desired roots by hand into your MATLAB code.
2. Implement an algorithm to find the roots of $J_1(x)$, possibly using a root-finding tool in MATLAB such as `fzero()` or `fsolve()`.
3. Find a third-party function or library that has already been written that solves the problem.

In this case we will take the last option since, it turns out, someone *has* already solved this problem for us and it is a safe bet that they did a better job than what we would be prepared to do. The MATLAB file exchange is an online repository where people can freely share code that they find useful and other users can look there in search of helpful code when needed.³

WE HAVE A LOT of experience with proprietary software. From operating systems like Microsoft Windows or Apple's iOS to office productivity tools like Microsoft Word or Excel, to valuable and important engineering tools like MATLAB or COMSOL. We also have experience with free software, such as many applications that you download onto your smartphones. I want to write a few words in hopes of dispelling any negative connotations that you may have developed in relation to open-source software in comparison to proprietary software.

¹ MATLAB *does* have built-in functions to represent several types of Bessel functions; $J_\nu(x)$ and $Y_\nu(x)$ are represented, respectively, by `besselj(nu,x)` and `bessely(nu,x)`. We will learn about more Bessel functions in future lectures.

² Frank Bowman. *Introduction to Bessel functions*. Courier Corporation, 2012

The Fourier-Bessel expansion that we are learning about in this lecture is a standard element in the analytical methods repertoire; *of course* someone else has already figured out how to find the roots of Bessel functions.

³ Note that a (free) MathWorks account is required to use the MATLAB file exchange.

- Scientists and engineers of all types—not just computer scientists—write and share software. Sometimes this software is open-source. Online repositories like GitHub and GitLab are meant expressly for developing open-source software in a collaborative way and then sharing the results freely.
- “Open-source” ensures the source code is available. Sometimes the code is also free but that is not the essential part.⁴
- Open-source software is a *hugely* important contribution to science. Some free and open-source tools include:
 - The L^AT_EX typesetting tools and almost all of the other software installed on the computer used to prepare this manuscript, including the Linux operating system.⁵
 - Programming languages that have been a part of the scientific computing landscape for generations. Some examples are Python, C++, Java and FORTRAN among others.
 - OpenMC⁶ - a powerful particle transport simulation tool similar to MCNP.
 - MOOSE - Multi-physics Object-Oriented Simulation Environment which combines the open-source finite element library libMesh⁸ and the Portable, Extensible Toolkit for Scientific Computation (PETSc)⁹ along with a host of other free, open-source libraries to create an enormously powerful and flexible tool-set that is used to create *the majority of all new multi-physics nuclear analysis codes in the United States*.¹⁰
- As the previous item should help illustrate, open-source software can be of very high quality. The developers of MOOSE-based applications at the Department of Energy labs are highly trained scientists following nuclear quality assurance standards to ensure that the resulting software tools work correctly and do what they are supposed to do.

If you have any interest in scientific computing, now is a good time to also develop an interest in open-source software.

Back to the Example

We want to expand $f(x) = x$ for $0 < x < 3$ in a Fourier-Bessel series expansion using Bessel functions of the first kind of order 1 that satisfy the boundary condition: $J_1(3\alpha_n) = 0$. We will use MATLAB along with the function `besselzero()` that we obtained from the MATLAB file exchange to carry out this task. In particular we

⁴ To paraphrase the famous open-source software icon, Richard Stallman: free software means “free speech,” not “free beer.” But sometimes it is a lot like free beer too.

⁵ MATLAB is a notable exception to this list. There is a free and open-source alternative called Octave. <https://octave.org/>

⁶ OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy*, 82: 90–97, 2015

⁷ Alexander D. Lindsay et al. 2.0 - MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX*, 20: 101202, 2022. ISSN 2352-7110

⁸ Benjamin S Kirk, John W Peterson, Roy H Stogner, and Graham F Carey. libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22:237–254, 2006

⁹ Balay et al. PETSc/TAO Users Manual. Technical Report ANL-21/39 - Revision 3.19, Argonne National Laboratory, 2023

¹⁰ For a list of current applications tracked by the MOOSE development team see: https://mooseframework.inl.gov/application_usage/tracked_apps.html. Not all of these codes are open-source, but they have all been created with open-source tools.

will compute the truncated expansion with $N = 15$ terms:

$$f(x) = x = \sum_{n=1}^{15} c_n J_1(\alpha_n x)$$

1. Use `besselzero()` to get $\alpha_1, \alpha_2, \dots, \alpha_N$ for our expansion.

```

clear
clc
close 'all'

N = 15; % number of eigenvalues
a = 0; b = 3; % bounds of the domain
nu = 1; kind = 1;
k = besselzero(nu,N,kind); % get roots ❶
alpha = k/b; ❷

```

1
2
3
4
5
6
7
8
9

❶ `besselzero()` takes up to three arguments; the first, ν , is mandatory and refers to the order of the Bessel function; the second is the number of roots requested. This argument defaults to 5 if omitted. The third argument is to indicate the *kind*—first or second—of Bessel function for which you want the roots (indicated by `kind=1` or `kind=2`); default is 1 for first kind.

❷ since $J_1(\alpha_n 3) = k_n$, where k_n is the n^{th} root of J_1 , α_n must be equal to $k_n/3$.

Now we have the first $N=15$ values of α_n .

2. Compute the coefficients of the expansion c_n . As with the Fourier series, we do this by multiplying both sides of our equation by an orthogonal function *and the weight function*, $p(x) = x$, and integrating. For example, to get c_1 , we do the following:

$$\begin{aligned}
 f(x) &= x = c_1 J_1(\alpha_1 x) + c_2 J_1(\alpha_2 x) + \dots \\
 \int_0^3 x J_1(\alpha_1 x) x \, dx &= c_1 \int_0^3 J_1(\alpha_1 x)^2 x \, dx + c_2 \int_0^3 J_1(\alpha_2 x) J_1(\alpha_1 x) x \, dx + \dots \\
 &\quad \underbrace{\qquad\qquad\qquad}_{=0 \text{ by orthogonality}} \\
 \Rightarrow c_1 &= \frac{\int_0^3 x J_1(\alpha_1 x) x \, dx}{\int_0^3 J_1(\alpha_1 x)^2 x \, dx}
 \end{aligned}$$

For the calculation of c_1 , all of the remaining terms are zero due to the weighted orthogonality of the eigenfunctions $J_1(\alpha_n x)$. We repeat the process for all values of c_n and, in MATLAB, we implement this process in the form of a loop.

```

f = @(x) x;
cn = nan(N,1); % store the coefficients (optional)
FB = @(x) 0; % initialize the Fourier-Bessel expansion
for n = 1:N
    % compute the i-th coefficient
    cn(n) = ...
        integral(@(x) f(x).*besselj(nu,alpha(n)*x).*x,a,b) ./ ... ❸
        integral(@(x) x.*besselj(nu,alpha(n)*x).^2,a,b);
    % update the Fourier-Bessel expansion
    FB = @(x) FB(x) + cn(n)*besselj(nu,alpha(n)*x);
end
end

```

10
11
12
13
14
15
16
17
18
19
20
21

❸ these three lines are actually one long line of MATLAB that calculates the coefficients:

$$c_n = \frac{\int_0^3 x J_1(\alpha_1 x) x \, dx}{\int_0^3 J_1(\alpha_1 x)^2 x \, dx}$$

We are now ready to plot the resulting Fourier expansion.

```

Nx = 1000;
X = linspace(a,b,Nx); ❹

```

22
23

❹ Create a vector to represent the x -axis.

```

figure(1)
plot(X,FB(X), '-b', 'LineWidth', 3);
xlabel('X', 'fontsize', 14, 'fontweight', 'bold');
ylabel('f(X)', 'fontsize', 14, 'fontweight', 'bold');
titlestr = sprintf('Fourier-Bessel expansion, N = %d', N);
title(titlestr, 'fontsize', 16, 'fontweight', 'bold');
grid on
set(gca, 'fontsize', 12, 'fontweight', 'bold');

```

The Fourier-Bessel expansion of $f(x) = x$ with $N = 15$ is shown in Figure 19. Note that the expansion for $N = 15$ looks pretty rough. There are many wiggles through the domain and the expansion drops suddenly to zero as the function approaches $x = 3$. The reason for this is that it had to. We are building the expansion with orthogonal functions that are all equal to zero at $x = 3$. Of course $f(x) = x$ is equal to 3 at $x = 3$ so something had to give.

We can improve the quality of the expansion by taking more terms. Luckily, since we are using a computer, it is no problem at all to simply increase N ; the computer does the same thing, just more of it. The result is shown in Figure 20 where the wigginess remains—including the Gibbs phenomena we saw with Fourier series—but overall the representation is much more exact.

Measuring Expansion Accuracy

There is a straight-forward way to be more precise when we speak of the accuracy of an orthogonal function expansion. A frequently used relative error measure is shown in Equation 79:

$$\text{Relative error} = \frac{(f(x) - FB(x), f(x) - FB(x))}{(f(x), f(x))} = \dots$$

$$\frac{\int_a^b (f(x) - FB(x))^2 dx}{\int_a^b f(x)^2 dx} \quad (79)$$

MATLAB code for quantitatively measuring the relative error as the number of terms increases is shown below. From Figure 21 we can see that, as expected, the relative error steadily goes down.¹¹

```

clear
clc
close 'all'

N = 500; % number of eigenvalues
a = 0; b = 3; % bounds of the domain
nu = 1; kind = 1;
k = besselzero(nu,N,kind); % get roots
alpha = k/b;

f = @(x) x;

```

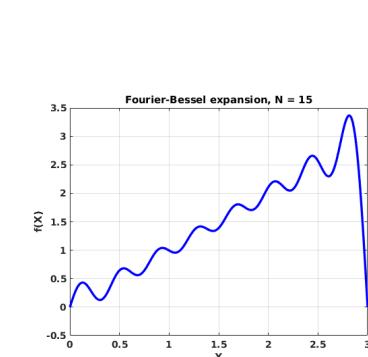


Figure 19: Fourier-Bessel expansion of $f(x) = x$.

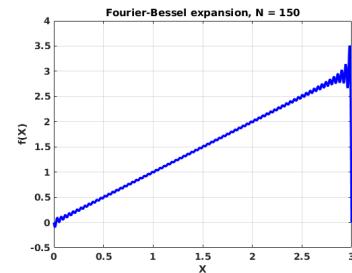


Figure 20: Fourier-Bessel expansion of $f(x) = x$.

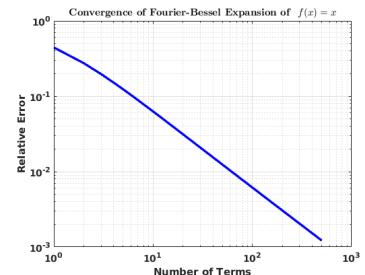


Figure 21: Convergence of the Fourier-Bessel expansion of $f(x) = x$.

¹¹ Note that it is conventional to show convergence graphs such as this on a log-log plot. Eventually, we should expect errors in the determination of Bessel function roots and/or errors in carrying out the numeric integration to prevent further reduction in relative error.

```

cn = nan(N,1); % store the coefficients (optional)
rel_err = nan(N,1);

FB = @(x) o; % initialize the Fourier-Bessel expansion
for n = 1:N
    % compute the i-th coefficient
    cn(n) = ...
        integral(@(x) f(x).*besselj(nu,alpha(n)*x).*x,a,b) ./ ...
        integral(@(x) x.*besselj(nu,alpha(n)*x).^2,a,b);
    % update the Fourier-Bessel expansion
    FB = @(x) FB(x) + cn(n)*besselj(nu,alpha(n)*x);

    % calculate square norm of the relative "error"
    err_fn = @(x) FB(x) - f(x);
    rel_err(n) = integral(@(x) err_fn(x).^2,a,b) ./ ...
        integral(@(x) f(x).^2,a,b);
end

figure(1)
loglog(1:N,rel_err,'-b',...
    'LineWidth',3);
title('Convergence of Fourier-Bessel Expansion of f(x)=x^3',...
    'Interpreter','latex');
ylabel('Relative Error','FontSize',14,...
    'FontWeight','bold');
xlabel('Number of Terms','FontSize',14,...
    'FontWeight','bold');
grid on
set(gca,'FontSize',12,'FontWeight','bold');

```


Lecture 20 - Fourier-Legendre Series Expansion

Objectives

- Revisit the Legendre equation as a Sturm-Liouville problem and give its orthogonality relation.
- Give an example to show the expansion of a function in terms of Legendre polynomials.

Orthogonality with Legendre Polynomials

We have some experience with Legendre's equation and their solutions, Legendre polynomials. As a recap, however, Legendre's equation is shown in Equation 80.

$$(1-x^2) u'' - 2xu' + n(n+1)u = 0, \quad x \in (-1,1) \quad (80)$$

The general solution is $u(x) = c_n P_n(x)$ where $P_n(x)$ is the Legendre polynomial of order n .

As demonstrated in Lecture 18, the self-adjoint form for Legendre's equation is given in Equation 81.

$$\frac{d}{dx} \left[(1-x^2) u' \right] + \overbrace{n(n+1)}^{\lambda} u = 0 \quad (81)$$

The orthogonality relation is shown below:

$$\int_{-1}^1 P_m(x) P_n(x) (1) dx = \begin{cases} 0, & m \neq n \\ \frac{2}{2n+1}, & m = n \end{cases}$$

If we need to represent a function $f(x)$ in terms of Legendre polynomials, we can carry out a *Fourier-Legendre* expansion as shown below:

$$f(x) = \sum_{n=0}^{\infty} c_n P_n(x) \quad (82)$$

where:

$$c_n = \frac{(f(x), P_n(x))}{(P_n(x), P_n(x))} = \frac{\int_{-1}^1 f(x) P_n(x) dx}{2/2n+1} \quad (83)$$

As a reminder the first few Legendre polynomials are: $P_0(x) = 1$, $P_1(x) = x$, $P_2(x) = \frac{1}{2}(3x^2 - 1)$, and $P_3(x) = \frac{1}{2}(5x^3 - 3x)$.

Recall that for Legendre's equation, the weight function $p(x)$ is equal to 1. Also recall that $(P_n, P_n) = 2/2n+1$.

As usual we can derive the formulas for the coefficients c_n of Equation 83 by multiplying both sides of Equation 82 by $P_n(x)$ and integrating.

The convergence behavior of Fourier-Legendre series expansions is similar to that for the Fourier series expansion using trigonometric polynomials. This behavior is recapitulated in the next theorem.

Theorem 9 (Convergence of Fourier-Legendre Series)

Let $f(x)$ and $f'(x)$ be piece-wise continuous on the interval $[-1, 1]$. Then for all x in the interval, the Fourier-Legendre series of f converges to $f(x)$ at a point where $f(x)$ is continuous and to the average:

$$\frac{f(x^+) + f(x^-)}{2}$$

at points where $f(x)$ is discontinuous.

This also happens to be true for Fourier-Bessel expansions.

Example: Construct the Fourier-Legendre expansion of:

$$f(x) = \begin{cases} 0, & -1 < x < 0 \\ 1, & 0 \leq x < 1 \end{cases}$$

Since the tools we need to use have largely been introduced already, I will simply present the necessary MATLAB code in a single listing.

```

clear
clc
close 'all'

f = @(x) ex1(x);

N = 15; % number of terms
a = -1; b = 1; % boundaries

% handle P0 coefficient separately
co = (1/2)*integral(@(x) f(x),a,b); ❶
cn = nan(N-1,1);
error_norm = nan(N,1);

FL = @(x) co;

% calculate relative error
err_fn = @(x) FL(x) - f(x);
error_norm(1) = integral(@(x) err_fn(x).^2,a,b) ./ ...
    integral(@(x) f(x).^2,a,b);

for n = 1:(N-1)
    % compute the n'th coefficient ❷
    cn(n) = ((2*n+1)/2)*integral(@(x) f(x).*legendreP(n,x),a,b);
    FL = @(x) FL(x) + cn(n)*legendreP(n,x); %update the
    expansion

    % compute the error.
    err_fn = @(x) FL(x) - f(x);
    error_norm(n+1) = integral(@(x) err_fn(x).^2,a,b) ./ ...
        integral(@(x) f(x).^2,a,b); % normalize error by size of
        function.
end

```

❶ Recall that $P_0(x) = 1$ and, according to our formula,

$$\begin{aligned} (P_0(x), P_0(x)) &= \frac{2}{2n+1} \\ &= \frac{2}{2(0)+1} \\ &= 2. \end{aligned}$$

Hence:

$$\begin{aligned} c_0 &= \frac{(f(x), P_0)}{(P_0, P_0)} \\ &= \frac{\int_{-1}^1 f(x)(1) dx}{2} \end{aligned}$$

❷ In addition to using the formula for $(P_n(x), P_n(x))$ we use the built-in MATLAB function for constructing $P_n(x)$: `legendreP(n,x)`.

```

%% Plot the result
Nx = 1000;
X = linspace(a,b,Nx);

figure(1)
plot(X,FL(X),'-g',...
      X,f(X),'--b',...
      'LineWidth',3);
grid on
xlabel('X','fontsize',14,'fontweight','bold');
ylabel('f(X)','fontsize',14,'fontweight','bold');
titlestr = ...
    sprintf('Fourier-Legendre expansion, N = %d',N);
title(titlestr,'fontsize',16,'fontweight','bold');
set(gca,'fontsize',12,'fontweight','bold');

%% Plot the error
figure(2)
loglog(1:N,error_norm,'-ok','linewidth',3);
title('Convergence behavior','fontsize',16,'fontweight','bold');
grid on
xlabel('Number of Fourier-Legendre Terms','fontsize',14,...
      'fontweight','bold');
ylabel('Relative Error','fontsize',14,'fontweight','bold');
set(gca,'fontsize',12,'fontweight','bold');

%% Local functions
function y = ex1(x)
[m,n] = size(x);
% expect vector inputs.
assert(min(m,n) == 1,'Bad input for ex1'); ③
% construct y so that it has the same shape as x
y = nan(m,n);
for i = 1:length(x)
    if (x(i) > -1) && (x(i) < 0)
        y(i) = 0;
    elseif (x(i) >= 0) && (x(i) < 1)
        y(i) = 1;
    end
end
end

```

③ Here we used an assert() function to enforce the requirement that inputs to ex1(x) be scalars or vectors but not a matrix. Any time you write a piece of code that relies on some kind of assumption—the input x must be a vector, for example—you really should add something like this assert() function to ensure that your assumption really is true. For larger software projects this sort of small-scale testing is essential for code reliability and maintainability.

A PLOT OF the Fourier-Legendre expansion is shown in Figure 22 and the convergence behavior is shown in Figure 23. Several things should be noted.

1. Clearly we can see from Figure 22 that the Fourier-Legendre expansion is converging to the average value at the point of discontinuity at $x = 0$.
2. Like other Fourier expansions, perturbations (“wiggliness”) is introduced by that discontinuity and this is something that we should learn to expect.
3. Also note that from Figure 23 we see that the expansion improves when we add the c_0 term, the c_1 term, c_3 term, and all

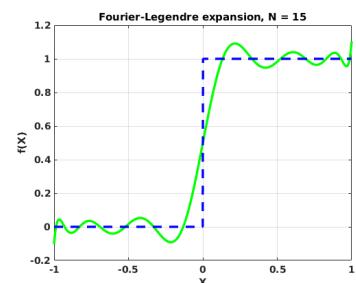


Figure 22: Fourier-Legendre expansion with $N = 15$.

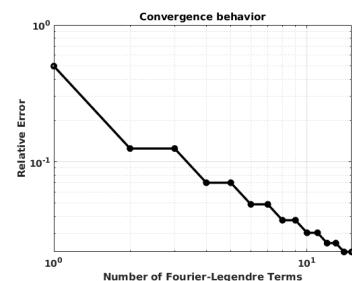


Figure 23: Convergence of Fourier-Legendre expansion

odd-numbered terms but the relative error does not change for the even-numbered terms c_2, c_4, \dots, c_{14} . Looking at $f(x)$ it should be apparent that, in some sense anyway, the function is *odd*—or at least “odd-ish”; you could make it odd by subtracting out a constant term (i.e. $f(x) - 0.5$ is odd and the c_0 coefficient is equal to that 0.5). The even-order Legendre polynomials are even and orthogonal to $f(x)$.

Assignment #7

Find the eigenfunctions and the equation that defines the eigenvalues for the boundary-value problem. Use MATLAB to estimate the first 4 eigenvalues $\lambda_1, \lambda_2, \lambda_3$, and λ_4 . Give the eigenfunctions corresponding to these eigenvalues and find the square norm of each eigenfunction.

1. $u'' + \lambda u = 0, \quad u'(0) = 0, \quad u(1) + u'(1) = 0$

2. Consider $u'' + \lambda u = 0$ subject to $u'(0) = 0, u'(L) = 0$. Show that the eigenfunctions are:

$$\left\{ 1, \cos \frac{\pi x}{L}, \cos \frac{2\pi x}{L}, \dots \right\}$$

This set, which is orthogonal on $x \in [0, L]$, is the basis for the Fourier cosine series.

3. Consider the following boundary value problem:

$$\begin{aligned} x^2 u'' + xu' + \lambda u &= 0, \quad x \in (1, 5) \\ u(1) &= 0, \quad u(5) = 0 \end{aligned}$$

(a) Find the (non-trivial) eigenvalues and eigenfunctions of the boundary value problem. Note: this is a Cauchy-Euler equation with solutions of the form $u = x^m$.

(b) Put the differential equation into self-adjoint form.

(c) Give the orthogonality relation. Use MATLAB to verify the orthogonality relation for the first two eigenfunctions.

4. Consider Laguerre's differential equation defined on the semi-infinite interval $x \in (0, \infty)$:

$$xu'' + (1-x)u' + \frac{\lambda}{n}u = 0, \quad n = 0, 1, 2, \dots$$

This equation has polynomial solutions $L_n(x)$. Put the equation into self-adjoint form and give an orthogonality relation.

For the next two problems, please use MATLAB along with the provided function `besselzero(nu,n,kind)` as shown in class.

5. Find the first four $\alpha_n > 0$ defined by $J_1(3\alpha) = 0$.
6. Expand $f(x) = 1$, $0 < x < 2$, in a Fourier-Bessel series using Bessel functions of order zero that satisfy the boundary condition: $J_0(2\alpha) = 0$. Make a plot in MATLAB of the given function and the Fourier-Bessel expansion of the function with the first four terms.

For the next problem, use the MATLAB built-in function `legendreP(n,x)` to represent Legendre Polynomials for Fourier-Legendre expansions.

7. Use MATLAB to calculate and print out the value of the first five non-zero terms in the Fourier-Legendre expansion of the given function. Make a plot in MATLAB of the given function and the Fourier-Legendre partial sum with five (non-zero) terms.

$$f(x) = \begin{cases} 0, & -1 < x < 0 \\ x, & 0 < x < 1 \end{cases}$$

Part IV

Boundary Value Problems in Rectangular Coordinates

Lecture 21 - Introduction to Separable Partial Differential Equations

Objectives

- Review description of linear second-order, Partial Differential Equations (PDEs).
- Introduce a classification scheme for second-order linear PDEs.
- Illustrate the use of separation of variables to find solutions to some PDEs.

Linear Partial Differential Equations

Consider the linear, second-order, partial differential equation in two independent variables shown in Equation 84:

$$A \frac{\partial^2 u}{\partial x^2} + B \frac{\partial^2 u}{\partial x \partial y} + C \frac{\partial^2 u}{\partial y^2} + D \frac{\partial u}{\partial x} + E \frac{\partial u}{\partial y} + Fu = G \quad (84)$$

where coefficients A through G are constants or functions of the *independent* variables x and/or y only.¹ If $G = 0$ then the equation is *homogeneous*, otherwise the equation is *non-homogeneous*.

¹ If the coefficients are functions of the dependent variable u or any of its partial derivatives, the equation would, of course, be non-linear.

Classification of Linear 2nd-Order PDEs

The solution of a PDE is a *function* of two (or more) independent variables that satisfies the PDE and boundary/initial conditions in some region of the space defined by the independent variables. Some important qualitative features of the solutions can be anticipated by using the following classification scheme for linear second-order PDEs.

Hyperbolic: $B^2 - 4AC > 0$

Hyperbolic differential equations are characteristic of wave-type phenomena. In the linear homogeneous case, waves travel through

the domain without distortion until a boundary is encountered. We will examine problems such as vibrating strings and membranes that are governed by hyperbolic PDEs and will exhibit this wave-type behavior.

Parabolic: $B^2 - 4AC = 0$

Parabolic differential equations are characteristic of *diffusive* phenomena like transient heat conduction. The time evolution of the solution of these equations typically has a “smoothing” behavior. Even if the initial data is only piece-wise smooth, as time evolves the solution tends to diffuse into a smooth function.

Elliptic: $B^2 - 4AC < 0$

Elliptic differential equations are characteristic of *steady-state* phenomena like static electrical potential and the steady-state heat equation.

Example: Classify the following linear partial differential equations.

$$1. \ 3\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial y}$$

$$2. \ \frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial y^2}$$

$$3. \ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Separation of Variables

The basic technique we will use to solve second-order, linear, homogeneous PDEs is called separation of variables. Once again, I will illustrate this method by way of doing an example.

Example: use separation of variables to find product solutions of:

$$\frac{\partial^2 u}{\partial x^2} = 4\frac{\partial u}{\partial y} \quad (85)$$

Step #1: Assume a solution can be expressed as a product of functions—one function for each independent variable.

$$u(x, y) = F(x)G(y)$$

There is an important first-order PDE that does not conform to this classification scheme but is considered hyperbolic. A typical example is the scalar linear advection equation:

$$u_t + a \cdot \nabla u = f(x, y)$$

This equation exhibits similar wave-type behavior.

A common non-linear variation is:

$$u_t + \nabla \cdot f(u) = 0$$

where $f(u)$ is called a *flux function*. This equation plays a role in modeling a variety of physical conservation laws often associated with transport phenomena. These equations are known for being capable of producing shocks—discontinuities in the solution—even when the initial condition is smooth.

Step #2: Insert the proposed solution into the governing equation.

$$\frac{\partial^2}{\partial x^2} [F(x)G(y)] = 4 \frac{\partial}{\partial y} [F(x)G(y)]$$

$$F_{xx}G = 4FG_y$$

Here we will use subscript notation to denote partial derivatives.

Step #3: Separate variables and introduce a separation constant. In this example we will separate variables by dividing both sides of the equation by $4FG$.

$$\begin{aligned}\frac{F_{xx}G}{4FG} &= \frac{4FG_y}{4FG} \\ \frac{F_{xx}}{4F} &= \frac{G_y}{G}\end{aligned}$$

In this last equation, the terms on the left are only a function of x ; the terms on the right are only a function of y . The left- and right-hand side of the equality must be the same for *all* values of x and y . The only way this can be expected to be true is if *both* sides are equal to a *constant*. We will denote this constant: $-\lambda$.

$$\frac{F_{xx}}{4F} = \frac{G_y}{G} = -\lambda$$

We can now decompose the partial differential equation in two independent variables into two ordinary differential equations:

$$F_{xx} + 4\lambda F = 0$$

$$G_y + \lambda G = 0$$

This bit of reasoning is a key element of separation of variables.

You might wonder why we chose $-\lambda$ rather than λ . In honesty there is no good answer to this question; let us chalk it up to a bias towards having a plus-sign in the separated equations.

Step #4: Form product solutions for all possible values of λ .

$\lambda = 0$:

$$F_{xx} = 0 \Rightarrow F(x) = c_1 + c_2x$$

$$G_y = 0 \Rightarrow G(x) = c_3$$

$$\begin{aligned}u(x, y) &= F(x)G(x) = (c_1 + c_2x)c_3 \\ &= A_1 + B_1x\end{aligned}$$

$\lambda < 0$: For this case we will let $\lambda = -\alpha^2$, $\alpha > 0$.

$$F_{xx} - 4\alpha^2 F = 0 \Rightarrow F(x) = c_1 \cosh 2\alpha x + c_2 \sinh 2\alpha x$$

$$G_y - \alpha^2 G = 0 \Rightarrow G(y) = c_3 e^{\alpha^2 y}$$

$$\begin{aligned}u(x, y) &= F(x)G(y) = (c_1 \cosh 2\alpha x + c_2 \sinh 2\alpha x)c_3 e^{\alpha^2 y} \\ &= (A_2 \cosh 2\alpha x + B_2 \sinh 2\alpha x)e^{\alpha^2 y}\end{aligned}$$

The “possible values” of λ can be put into three familiar categories: λ can be *positive*, *negative*, or *zero*.

Note how in this case and the cases to follow, we will simply write down the general solution to the separated ODEs with little/no to-do over deriving that solution. By this point in the course you *need* to be able to quickly recognize those equations. In most cases you should be able to write down the solutions by inspection.

We will assume, for this problem, that the x -dimension is bounded and thus it is convenient to use the $\cosh 2\alpha x$ and $\sinh 2\alpha x$ form of the solution. If the domain is unbounded you would use $e^{2\alpha x}$ and $e^{-2\alpha x}$. It will be up to you to make this determination.

$\lambda > 0$: For this case we will let $\lambda = \alpha^2$, $\alpha > 0$.

$$\begin{aligned} F_{xx} + 4\alpha^2 F = 0 &\Rightarrow F(x) = c_1 \cos 2\alpha x + c_2 \sin 2\alpha x \\ G_y + \alpha^2 G = 0 &\Rightarrow G(y) = c_3 e^{-\alpha^2 y} \end{aligned}$$

$$\begin{aligned} u(x, y) = F(x)G(y) &= (c_1 \cos 2\alpha x + c_2 \sin 2\alpha x) c_3 e^{-\alpha^2 y} \\ &= (A_3 \cos 2\alpha x + B_3 \sin 2\alpha x) e^{-\alpha^2 y} \end{aligned}$$

Notes:

- There is no assurance that a linear 2nd-order PDE will be separable. We will spend a lot of time in this course in dealing with equations that happen to be separable. In reality, many are not, in particular if the problem is non-homogeneous. It is a good idea to check to see if an equation is homogeneous before launching down the separation-of-variables path.
- This example is a bit of an anomaly. We will usually not attempt to find *general* solutions to PDEs, but only *particular* solutions. Therefore a problem statement will not be fully meaningful without boundary/initial conditions by which we will be able to derive particular solutions.
- Specific values of λ that result in non-trivial solutions will depend on the boundary conditions.
- Since the PDEs are linear, the *superposition principle* will apply. That is, if u_1, u_2, \dots, u_k are solutions of a linear homogeneous PDE (including boundary conditions) then a linear combination:

In this equation c_i are constants.

$$u = c_1 u_1 + c_2 u_2 + \cdots + c_k u_k$$

is also a solution.

Lecture 22 - Classical PDEs and BVPs

Objectives

- Describe three important PDEs: heat equation, wave equation, and Laplace equation.
- Describe the physical meaning of common boundary conditions.
- Discuss important modifications to the three equations to incorporate additional physical phenomena.

The Heat Equation

The time-dependent heat equation in one spatial dimension is given in Equation 86:

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad a < x < b \quad (86)$$

where α^2 corresponds to thermal diffusivity which, in turn is given in Equation 87:

$$\alpha^2 = \frac{k}{\rho c_p} \quad (87)$$

where k is thermal conductivity, ρ is the density, and c_p is the specific heat at constant pressure and the dependent variable u is the temperature. All of these material properties must be positive for physically meaningful materials and, for the time being at least, we will consider all of these properties to be constant.¹

WE WILL NOT delve into the derivation of Equation 86; this is left for your class in heat transfer. Suffice it to say here that the equation is a mathematical expression of conservation of energy. The following assumptions are incorporated into this expression:

1. Heat is flowing in one spatial direction only. This is the reason why the equation is only a function of x . Think of this as heat flowing in a wire.

¹ This is a very important assumption mathematically and it is also untrue for relevant materials. The thermal conductivity of most materials is temperature dependent as is the density and specific heat. If we allowed for this bit of realism to slip into our mathematical analysis, however, the differential equation would become nonlinear— α would become a function of the dependent variable u —and we would not be able to solve it with methods taught in this class. Tools based on the finite element method such as MOOSE and COMSOL are specifically designed to deal with this sort of nonlinearity.

2. Since heat is assumed to only flow in the x -direction, you should assume that the lateral surfaces of this wire are insulated.
3. We assume that no heat is generated in the domain.
4. We assume that the material is homogeneous.
5. We also assumed a particular relationship between heat flow and the temperature gradient:

$$q = -k \frac{\partial u}{\partial x} \quad (88)$$

where q is the *heat flux*. Relationships such as given in Equation 88 are referred to as *constitutive* relationships.

To BE FULLY meaningful as an initial boundary value problem (IBVP), Equation 86 must be accompanied by an initial condition—say an initial temperature profile, $u(x, 0) = f(x)$ —and two boundary conditions. We categorize the boundary conditions into three types:

Type 1: These are also called *Dirichlet boundary conditions*.² These conditions apply to the dependent variable itself. For example:

$$u(a) = T_a, \quad u(b) = f(t)$$

Type 2: These are also called *Neumann boundary conditions* and they apply to the *derivative* of the dependent variable. For example:

$$\left. \frac{\partial u}{\partial x} \right|_{x=a} = 0$$

For the heat equation a homogeneous boundary condition of this type would indicate insulation at a boundary.³

Type 3: These are called *mixed* or *Robin* boundary conditions and they apply to *both* the dependent variable and its derivative. For example:

$$\left. \frac{\partial u}{\partial x} \right|_{x=b} = -h(u(b, t) - u_m), \quad h > 0$$

where, in this case, u_m is a constant reference temperature of the surrounding medium and h is a convective heat transfer coefficient. This boundary condition would correspond to convective heat transfer at the boundary in which the heat flux is proportional (h being the proportionality constant) to the difference in temperature between the boundary surface and the surrounding medium.

Strictly speaking, Equation 88 should reference an outward-pointing unit-normal vector. In a more generic case we would express the relationship as:

$$q = -k \nabla u \cdot \hat{n}$$

where \hat{n} is the outward-pointing unit normal on the surface through which the heat flux flows and, of course, ∇u is the temperature gradient. In a one-dimensional problem like this, ∇u reduces to $\partial u / \partial x$. To get the physics right (or, more specifically, to get the sign of the heat flux correct) for a particular problem, however, one will need to remember the dot-product with the outward-pointing unit normal.

² Named after the German mathematician Peter Gustav Lejeune Dirichlet who was known as a popular instructor at the Prussian Military Academy in the mid 19th century. He is also famous for having first established the convergence proofs that we have cited for Fourier series and he also studied and proved a unique solution for the first boundary value problem. That, to the best of my knowledge, is why this boundary condition type is named for him.

³ Insulation implies no heat transfer through a surface—i.e. no heat flux. Since heat flux is proportional to $\nabla u \cdot \hat{n}$ this implies, for one-dimensional problems, that $\partial u / \partial x = 0$.

Wave Equation

The wave equation is given by Equation 89:

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad a < x < b \quad (89)$$

where $\alpha^2 = \frac{T}{\rho}$ and T is tension and ρ is density.⁴ The dependent variable u refers to lateral displacement of the string and t is time.

This equation is derived as a mathematical expression of mechanical equilibrium of an elastic string held under tension. The assumptions built-in to this derivation include:

1. The string is “perfectly flexible.”
2. The string is homogeneous.
3. Displacements in the string are small relative to the string length.
4. Tension is constant; and
5. there are no other forces acting on the string.

A properly stated boundary value problem based on the wave equation will have two boundary conditions. For this course we will usually apply Dirichlet boundary conditions but others are possible. Since the equation is second-order in time we also need two temporal boundary conditions. Typically these are given as the initial displacement, $u(x, 0) = f(x)$, and initial velocity, $u_t(x, 0) = g(x)$.

Laplace Equation

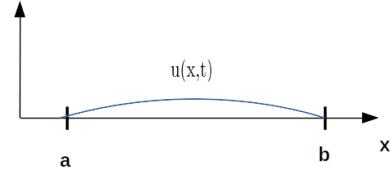
The Laplace equation in two dimensions in a Cartesian coordinate system is given by Equation 90:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (90)$$

The Laplace equation arises in studies of *steady state* phenomena involving potentials such as: electrostatic potential, gravitational potential, velocity, and heat conduction. The meaning of the dependent variable u , of course, depends upon what is being modeled.

Equation 90 can more concisely and generically be expressed using the Laplace operator ∇^2 . Using this notation, Equation 90 could be written: $\nabla^2 u = 0$. This same expression is valid for 1-, 2-, or 3-dimensional Cartesian coordinates but it is also valid for polar, cylindrical and spherical coordinates. The specialization comes in the definition of ∇ . We will address this further when we examine problems in those coordinate systems.

⁴ The variable α is also often referred to as the *wave speed*.



The Laplace operator ∇^2 is short-hand for:

$$\nabla^2 = \nabla \cdot \nabla$$

where in Cartesian coordinates:

$$\begin{aligned} \nabla &= \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle \\ \nabla \cdot \nabla &= \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle \cdot \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle \\ &= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \end{aligned}$$

also expressed as:

$$\nabla^2 = \Delta$$

Boundary Value Problems

In all of the above cases, a full statement of the boundary value problem must include:

1. The partial differential equation,
2. all boundary conditions, and
3. initial conditions for time-dependent problems.

Example: Wave Equation Boundary Value Problem.

$$\text{PDE} \quad \frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < L, \quad t > 0$$

$$\text{BCs: } u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0$$

$$\text{ICs: } u(x, 0) = f(x), \quad u_t(x, 0) = g(x), \quad 0 < x < L$$

Note that, technically speaking, the PDE does not apply at the domain boundaries or at $t = 0$.

Important Variations to Classic BVPs

Both the heat equation and the wave equation incorporated several assumptions in the derivation. If these assumptions are modified or eliminated we can still derive an equation but the form of the equation will change. Some important variations are described here.

IN EQUATION 91 we show the heat equation in the case where there is an internal heat source and convection from lateral surfaces to a surrounding medium maintained at a constant temperature u_m :

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + S(x, t) \underbrace{-h(u - u_m)}_{\substack{\text{convection from} \\ \text{lateral surfaces}}} \quad (91)$$

where h is the convective heat transfer coefficient.

IN EQUATION 92 we show the wave equation in a case where we have an external force, damping and restoring forces.

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + f(x, t) \underbrace{-c \frac{\partial u}{\partial t}}_{\substack{\text{damping}}} \underbrace{-ku}_{\substack{\text{restoring} \\ \text{force}}} \quad (92)$$

AN IMPORTANT SKILL that an engineer needs to develop is the ability to translate a description of a physical system into a properly formulated boundary value problem that you can solve. The *point* is to be able to describe a system mathematically so that, by solving the math problem, you gain *insight* into the behavior of the physical system. Here are a couple of examples to get started.

Example: Consider a rod of length L that is insulated along its lateral surfaces. There is heat transfer from the left end of the rod into a surrounding medium at temperature 20° and the right end is insulated. The initial temperature is $f(x)$ throughout. We would like to know what the temperature distribution is as a function of time and space. The corresponding BVP is:

$$\text{PDE: } \frac{\partial u}{\partial t} = k \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < L, \quad t > 0$$

$$\text{BCs: } \left. \frac{\partial u}{\partial x} \right|_{x=0} = -h(u(0,t) - 20), \quad \left. \frac{\partial u}{\partial x} \right|_{x=L} = 0, \quad t > 0$$

$$\text{IC: } u(x,0) = f(x), \quad 0 < x < L$$

This is something that students in this class often struggle with.

Example: Consider a string of length L held in tension. The ends are secured to the x -axis, and the string is initially at rest on that axis. An external vertical force proportional to the horizontal distance from the left end acts on the string for $t > 0$. The corresponding BVP is:

$$\text{PDE: } \frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + hx, \quad 0 < x < L, \quad t > 0$$

$$\text{BCs: } u(0,t) = 0, \quad u(L,t) = 0, \quad t > 0$$

$$\text{ICs: } u(x,0) = 0, \quad u_t(x,0) = 0, \quad 0 < x < L$$

Example: Consider a semi-infinite plate coinciding with the region $0 \leq x \leq \pi, y \geq 0$. The left end is held at temperature e^{-y} , and the right end is held at temperature 100°C for $0 < y \leq 1$ and 0°C for $y > 1$. The bottom of the plate, $y = 0$, is held at temperature $f(x)$. The corresponding BVP is:

$$\text{PDE: } \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad 0 < x < \pi, \quad y > 0$$

$$\text{BCs: } u(0,y) = e^{-y}, \quad y > 0, \quad u(\pi,y) = \begin{cases} 100 & 0 < y \leq 1 \\ 0 & y > 1 \end{cases}$$

$$u(x,0) = f(x), \quad 0 < x < \pi$$

One implicit constraint that may need to be applied in this cases is:
 $\lim_{y \rightarrow \infty} u(x,y) < \infty$.

Assignment #8

Use separation of variables to find, if possible, product solutions for the given partial differential equations. Be sure to consider cases for all possible values of the separation constant.

1. $\frac{\partial u}{\partial x} = \frac{\partial u}{\partial y}$

2. $\alpha^2 \frac{\partial^2 u}{\partial x^2} - u = \frac{\partial u}{\partial t}, \alpha > 0$

Note: for this problem, when separating variables, divide by $\alpha^2 X(x)T(t)$ and keep all terms with α^2 together.

Classify the given partial differential equation as hyperbolic, parabolic, or elliptic.

3. $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 u}{\partial y^2} = 0$

4. $\frac{\partial^2 u}{\partial x^2} = 0 \frac{\partial^2 u}{\partial x \partial y}$

5. $\frac{\partial^2 u}{\partial x^2} + 2 \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial u}{\partial x} - 6 \frac{\partial u}{\partial y} = 0$

Show that the given partial differential equation possesses the indicated product solution.

6. $\frac{\partial u}{\partial t} = k \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} \right), u(r, t) = e^{-k\alpha^2 t} (c_1 J_0(\alpha r) + c_2 Y_0(\alpha r))$

For the following problems, a rod of length L coincides with the interval $[0, L]$ on the x-axis. Set up the boundary-value problem for the temperature $u(x, t)$.

7. The left end is held at temperature zero and the right end is insulated. The initial temperature is $f(x)$ throughout.

8. The left end is at temperature $\sin(\pi t/L)$, the right end is held at zero, and there is heat transfer from the lateral surface of the rod into the surrounding medium held at temperature zero. The initial temperature is $f(x)$ throughout.

For the following problems a string of length L coincides with the interval $[0, L]$ on the x-axis. Set up the boundary-value problem for the displacement $u(x, t)$.

9. The ends are secured to the x-axis. The string is released from rest from the initial displacement $u(x, 0) = x(L - x)$.
10. The left end is secured to the x-axis but the right end moves in a transverse manner according to $\sin(\pi t)$. The string is released from rest from the initial displacement $f(x)$. For $t > 0$ the transverse vibrations are damped with a force proportional to the transverse velocity of the string.

For the next problem, set up the boundary-value problem for a steady-state temperature $u(x, y)$.

11. A thin rectangular plate coincides with the region in the xy-plane defined by: $0 \leq x \leq 4$, $0 \leq y \leq 2$. The left end and the bottom of the plate are insulated. The top of the plate is held at temperature zero, and the right end of the plate is held at a temperature $f(y)$.

Lecture 23 - The Heat Equation

Objectives

- Demonstrate use of separation of variables to solve the heat equation.
- Show the code for a MATLAB implementation of an example problem.

Analytic Solution

Consider the following boundary value problem based on the heat equation:

$$\text{Governing Equation : } \frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$$

$$\text{Boundary Conditions : } u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = f(x), \quad 0 < x < L$$

This boundary value problem models transient heat conduction in a one-dimensional bar. The ends of the bar are held at a constant temperature of zero degrees and there is an initial temperature distribution described by $f(x)$.

We will follow the steps to find the solution using separation of variables.

Step #1: Assume a product solution:

$$u(x, t) = F(x)G(t)$$

Step #2: Insert proposed solution into the governing equation:

$$\begin{aligned} \frac{\partial}{\partial t} [F(x)G(t)] &= \alpha^2 \frac{\partial^2}{\partial x^2} [F(x)G(t)] \\ FG_t &= \alpha^2 F_{xx}G \end{aligned}$$

Note: once again we will use subscripts to denote partial derivatives.

Step #3: Separate variables:

$$\begin{aligned}\frac{FG_t}{\alpha^2 FG} &= \frac{\alpha^2 F_{xx} G}{\alpha^2 FG} \\ \frac{G_t}{\alpha^2 G} &= \frac{F_{xx}}{F} = -\lambda \\ G_t + \alpha^2 \lambda G &= 0, \quad F_{xx} + \lambda F = 0\end{aligned}$$

On the middle line we see that $\frac{G_t}{\alpha^2 G}$ is only a function of y ; $\frac{F_{xx}}{F}$ is only a function of x and yet they must be equal to each other for all values of x and y . The only way this makes sense is if they are both, in fact, constant. We will denote this constant $-\lambda$.

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

The boundary conditions must be applied to the separated equation for $F(x)$.¹

$$F_{xx} + \lambda F = 0, \quad F(0) = 0, \quad F(L) = 0, \quad 0 < x < L$$

We need to examine all possible values of λ .

$\lambda = 0$:

$$\begin{aligned}F_{xx} &= 0 \\ F(x) &= c_1 x + c_2 \\ F(0) &= c_1(0) + c_2 = 0 \\ \Rightarrow c_2 &= 0 \\ F(L) &= c_1(L) = 0 \\ \Rightarrow c_1 &= 0\end{aligned}$$

¹ The only way $G(t)$ can satisfy the homogeneous spatial boundary conditions would be for us to set $G(t) = 0$. Thus the product solution would be $u(x,t) = F(x)G(t) = F(x)(0) = 0$. Obviously a trivial solution, $u(x,t) = 0$, is not what we are looking for.

Thus we will disregard $\lambda = 0$ since only the trivial solution satisfies the governing equation and boundary conditions in that case.

$\lambda < 0$: Here we will set $\lambda = -\nu^2$, $\nu > 0$.

$$\begin{aligned}F_{xx} - \nu^2 F &= 0 \\ F(x) &= c_1 \cosh \nu x + c_2 \sinh \nu x \\ F(0) &= c_1 \cosh 0 + c_2 \sinh 0 \\ F(0) &= c_1 + 0 = 0 \Rightarrow c_1 = 0 \\ F(L) &= c_2 \sinh \nu L = 0\end{aligned}$$

Note again that we use the $\cosh()$ and $\sinh()$ form of the solution since the domain is bounded.

Here we have to recall that $\sinh x$ is strictly positive for $x > 0$. Therefore $c_2 = 0$ and, again, only the trivial solution satisfies the governing equation and boundary conditions for the case $\lambda < 0$ so we will discard this possibility.

$\lambda > 0$: Here we will set $\lambda = \nu^2$, $\nu > 0$.

$$\begin{aligned} F_{xx} + \nu^2 F &= 0 \\ F(x) &= c_1 \cos \nu x + c_2 \sin \nu x \\ F(0) &= c_1 \cos 0 + c_2 \sin 0 \\ F(0) &= c_1 + 0 = 0 \Rightarrow c_1 = 0 \\ F(L) &= c_2 \sin \nu L = 0 \end{aligned}$$

Finally we have something we can work with! Rather than setting $c_2 = 0$, we can observe that $\sin \nu L = 0$ whenever $\nu L = n\pi$, and n is a positive integer. Thus there are infinitely many values—which we will designate ν_n —that satisfy the condition: $\nu_n = n\pi/L$ with $n \in \mathbb{Z}^+$.²

For $\lambda = \nu^2$ we can now also solve the separated equation for $G(t)$:

$$\begin{aligned} G_t + \alpha^2 \nu^2 G &= 0 \\ G(t) &= c_3 e^{-(\alpha \nu)^2 t} \end{aligned}$$

We combine these values of ν_n with $F(x)$ and $G(x)$ —which we will now call eigenfunctions:

$$\begin{aligned} \nu_n^2 &= \left(\frac{n\pi}{L}\right)^2 \\ F_n(x) &= c_2 \sin \frac{\nu_n x}{L} = c_2 \sin \frac{n\pi x}{L} \\ G_n(t) &= c_3 e^{-(\alpha \nu_n)^2 t} = c_3 e^{-(\alpha \frac{n\pi}{L})^2 t} \end{aligned}$$

Recall that there are an infinite number of eigenfunctions; the solution will be formed by a linear combination of *all* of them. So our product solution is:

$$u(x, t) = F(x)G(t) = \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} e^{-(\alpha \frac{n\pi}{L})^2 t}$$

Step #5: Satisfy the initial condition.

$$\begin{aligned} u(x, 0) &= \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} e^{-(\alpha \frac{n\pi}{L})^2 0} \\ &= \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} = f(x) \end{aligned}$$

On the left we have an infinite series of eigenfunctions; on the right we have $f(x)$. Our job is to find the values of c_n so that they are equal. This is *exactly* the reason why we spent time learning about

Note that we exclude the case where $n = 0$ since that implies $\nu L = 0$ but we stipulated that $\nu > 0$.

² Traditionally, in mathematical literature, \mathbb{Z} denotes the set of all integers. The notation \mathbb{Z}^+ is used here to denote the set of all positive integers.

Here we implicitly take c_2 and c_3 from the separated solutions and combine them into c_n .

Fourier series and orthogonal function expansions. We will multiply both sides by our (orthogonal) eigenfunctions and integrate.

For c_1 we will do this explicitly:

$$u(x, 0) = c_1 \int_0^L \sin\left(\frac{\pi x}{L}\right)^2 dx + c_2 \int_0^L \sin\frac{2\pi x}{L} \sin\frac{\pi x}{L} dx + \dots = \dots$$

$\int_0^L f(x) \sin\frac{\pi x}{L} dx$

= 0, by orthogonality

The only non-zero term on the left will be the one corresponding to $\sin\frac{\pi x}{L}$; all others will be zero due to the orthogonality of the set of functions $\sin\frac{n\pi x}{L}$.

By orthogonality of the eigenfunctions $F_n(x)$ we can find the coefficients, c_n , one at a time by using the formula:

$$c_n = \frac{\int_0^L f(x) \sin\frac{n\pi x}{L} dx}{\int_0^L \sin\left(\frac{n\pi x}{L}\right)^2 dx} \quad (93)$$

You might recognize Equation 93; it is the same as the Sine series expansion given in Lecture 16. In particular the value of $\int_0^L \sin n\pi x/L^2 dx$ is equal to $L/2$ so the formula for the coefficients can be stated more concisely as:

$$c_n = \frac{2}{L} \int_0^L f(x) \sin\frac{n\pi x}{L} dx$$

In summary, the solution to our boundary value problem is:

$$u(x, t) = \sum_{n=1}^{\infty} c_n \sin\frac{n\pi x}{L} e^{-(\alpha\frac{n\pi}{L})^2 t}$$

$$c_n = \frac{2}{L} \int_0^L f(x) \sin\frac{n\pi x}{L} dx$$

To GET QUANTITATIVE information, we need to specify values for the length of the bar (L), the thermal diffusivity (α), and the initial temperature distribution, $f(x)$. But we can consider some qualitative aspects of the solution before we start computing.

1. What will the temperature profile look like as $t \rightarrow \infty$?
2. What will the solution look like initially if the temperature profile is piece-wise linear with discontinuities in the interval $[0, L]$?
3. What will happen to the solution as time evolves for temperature profiles that are initially discontinuous?

Answers:

1. Owing to the exponential term in the solution, $u(x, t) \rightarrow 0$ as $t \rightarrow \infty$.
2. Recalling our experience from Fourier series expansions of functions with discontinuities, the representation will be "wiggly."
3. Since the heat equation is a parabolic equation characteristic of diffusive phenomena, we expect the solution to "smooth-out" over time. This should jibe with our own personal intuition and experience with heat transfer.

MATLAB implementation

To demonstrate the answer to these questions and help build more insight into the behavior of the transient 1-D heat equation, let us define L , α^2 , and $f(x)$, compute and plot the solution.

```

1 clear
2 clc
3 close 'all'

4 %% Set parameters and define eigenfunctions
5 L = 1; % length of the domain
6 alpha_sq = 0.1; % thermal diffusivity ❶
7
8 N = 25; % number of terms to the series solution ❷
9
10 F = @(x,n) sin(n.*pi.*x./L); ❸
11 G = @(t,n) exp(-((n.*pi./L).^2)*alpha_sq.*t);
12
13 f(x) = @(x) x.*(1-x);
14

```

Note from Figure 24 that the initial condition is smooth and satisfies the boundary conditions.

To build the solution we combine the eigenfunctions along with the coefficients calculated using Equation 93.

```

16 %% Compute the solution
17 % initialize my series solution
18 u = @(x,t) 0;
19 for n = 1:N
20     % essentially doing the sine-series half-wave expansion
21     % compute the coefficient
22     cn = (2/L)*integral(@(x) f(x).*F(x,n),0,L);
23
24     % add the term to the series solution
25     u = @(x,t) u(x,t) + cn*F(x,n).*G(t,n);
26 end
27
28 %% plot the result
29 figure(1)
30 plot(X,u(X,0),'-b',...
31 X,u(X,0.1),'-g',...
32 X,u(X,0.5),'-r','linewidth',3);
33 title_str = sprintf('Heat Equation Example, N=%d',N);
34 title(title_str,'FontSize',16,'FontWeight','bold');
35 title('Lecture #23 Example','fontSize',16,'fontWeight','bold');
36 xlabel('X','fontSize',14,'fontWeight','bold');
37 ylabel('u(X,t)','fontSize',14,'fontWeight','bold');
38 grid on
39 set(gca,'fontSize',12,'fontWeight','bold');
40 legend('t = 0','t = 0.1','t = 0.5');

```

THE PLOT IS SHOWN in Figure 25. As is expected, the temperature is going down over time. As time goes to infinity, the solution will be zero everywhere. Recalling that the heat equation is simply a mathematical representation of conservation of energy, you should

❶ Strictly speaking we should include units for this quantity. For perspective, the thermal diffusivity of copper at room temperature is approximately $1.20 \text{ cm}^2/\text{s}$; for steel approximately $0.20 \text{ cm}^2/\text{s}$; and for adobe brick around $0.003 \text{ cm}^2/\text{s}$.

❷ We also need to choose the number of Fourier coefficients to calculate; we obviously cannot calculate them all.

❸ Be sure you understand how to define anonymous functions with multiple variables.

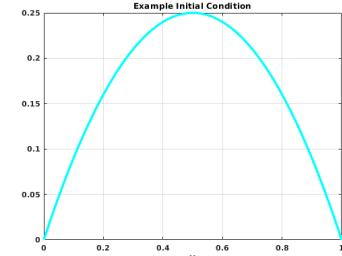


Figure 24: Smooth initial condition.

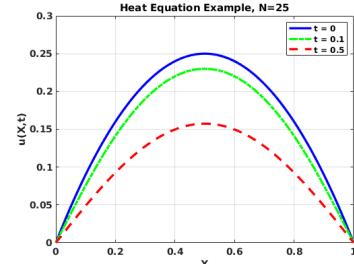


Figure 25: Solution for smooth initial condition.

ask yourself the question: where is the energy going? The answer is that the energy is flowing out of the left and right side of the bar and will continue to do so as long as the temperature of the bar is higher than the temperature at the boundary which, for this problem, is set to zero.

WHAT HAPPENS IF we increase the thermal diffusivity? Answer: the heat flows “faster.” Since the thermal diffusivity shows up in the equation $G(t)$, the answer should look like time “sped up.” Testing this hypothesis out on our MATLAB solution, we change thermal diffusivity to 1.2. The results are shown in Figure 26.

WHAT HAPPENS IF we have a much less smooth initial condition? Admittedly, it would be odd for the initial temperature distribution to be discontinuous, but given our experience with Fourier series expansions, we should have some idea as to what the Fourier series expansions of discontinuous functions should look like. To test this, suppose the initial temperature distribution were given by:

$$f(x) = \begin{cases} x, & 0 < x < \frac{L}{4} \\ 1, & \frac{L}{4} \leq x < \frac{L}{2} \\ 0, & \frac{L}{2} \leq x < \frac{3L}{4} \\ L - x, & \frac{3L}{4} \leq x < L \end{cases}$$

and shown in Figure 27. The solution (for $\alpha^2 = 0.1$) is shown in Figure 28.

Note the “wigginess” of the Fourier series representation of the initial condition; note also how that “wigginess” goes away almost immediately. This is due to the diffusive nature of the heat equation.

In much the same way as we could improve our resolution of functions represented by a Fourier series by computing more terms, we can do the same thing here. In Figure 29 we show the solution computed with $N = 100$ terms in the Fourier series.

Insulated Boundaries

As an exercise, let us consider what happens when we change the problem by insulating the boundary at $x = L$.

$$\text{Governing Equation : } \frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$$

$$\text{Boundary Conditions : } u(0, t) = 0, \quad \frac{du}{dx}(L, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = f(x), \quad 0 < x < L$$

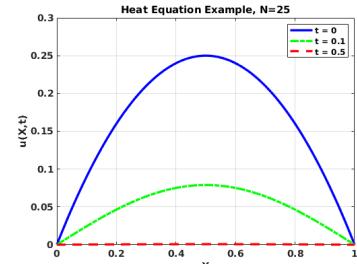


Figure 26: Solution with high thermal diffusivity.

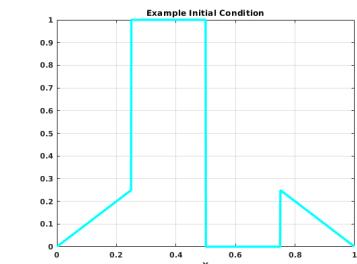


Figure 27: Example with discontinuous initial condition.

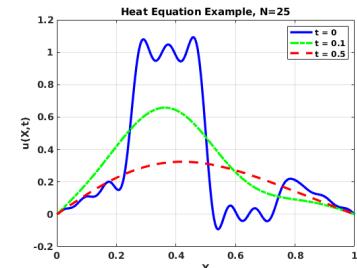


Figure 28: Solution with discontinuous initial condition, $N = 25$.

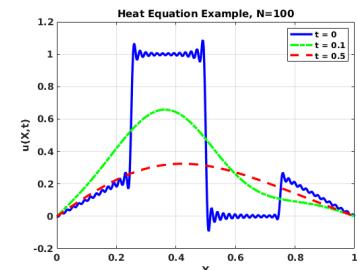


Figure 29: Solution with discontinuous initial condition, $N = 100$.

Before we do any analysis we should think about what we *expect* the solution to look like.³ Mathematically we implement the insulated boundary condition by setting the temperature gradient at that boundary equal to zero. Conceptually, we know this means that heat will no longer flow out of that boundary. Heat may or may not flow *towards* the right boundary depending on the temperature distribution within the domain but any heat reaching the right boundary will stay there until it can flow out towards the *left* boundary.

The details are left to the reader⁴ but application of separation of variables to the new boundary value problem yields the following solution:

$$\nu_n = \frac{(2n-1)\pi}{2L}, \quad n = 1, 2, 3, \dots$$

$$u(x, t) = \sum_{n=1}^{\infty} c_n \sin \nu_n x e^{-(\alpha \nu_n)^2 t}$$

$$c_n = \frac{\int_0^L f(x) \sin \nu_n x \, dx}{\int_0^L \sin (\nu_n x)^2 \, dx}$$

A plot of the solution when $N = 25$, $L = 1$, and $\alpha^2 = 1.5$ is shown in Figure 30.

WHAT HAPPENS IF both boundaries are insulated? Physically, when we insulate something, that means we want to keep heat from coming in or out of the domain. Does this mean heat will not diffuse *within* the domain? Of course not; heat will simply flow as it must while driven by temperature gradients in the domain. When will the diffusion stop? When there is no more temperature gradient to drive heat flow and that will happen when the temperature is uniform.

Mathematically, this means that we again change the boundary conditions so that the temperature gradient is zero at *both* boundaries. The details of this solution will be left to exercises but it is hoped by this point that you already know what the solution *must* look like; namely that heat will diffuse within the domain until a uniform temperature is reached that is equal to the *average* initial temperature.

³ This is important. Do not let yourself get stuck inside analytic blinders that prevent you from seeing the big picture. Eventually you will need to be prepared to critically scrutinize your answer and decide, on your own, whether or not it is reasonable and/or correct. To do this, you need to know *in advance* what you *expect* the solution to look like.

⁴ **Hint:** When we apply the insulated boundary condition, we will be left with $F_x(L) = \nu c_2 \cos \nu L = 0$ which we can satisfy if νL is an odd integer multiple of $\pi/2$. So $\nu_n L = \frac{(2n-1)\pi}{2}$, $n = 1, 2, 3, \dots$, and therefore $\nu_n = \frac{(2n-1)\pi}{2L}$.

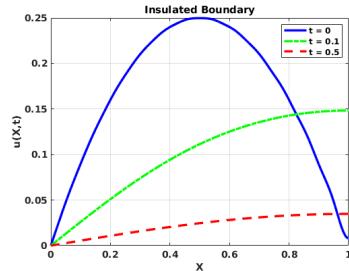


Figure 30: Solution with an insulated boundary at $x = L$.

Lecture 24 - The Wave Equation

Objectives

- Use separation of variables method to solve the Wave Equation.
- Illustrate the example solution with MATLAB.

Analytic Solution

Consider the following boundary value problem based on the wave equation:

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad a < x < b$$

$$\text{Boundary Conditions : } u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = f(x), \quad u_t(x, 0) = g(x), \quad 0 < x < L$$

This boundary value problem models a flexible string fixed on both ends with a specified initial displacement, $f(x)$, and initial velocity, $g(x)$.

We will follow the steps to find the solution using separation of variables.

Step #1: Assume a product solution:

$$u(x, t) = F(x)G(t)$$

Step #2: Insert proposed solution into the governing equation:

$$\begin{aligned} \frac{\partial^2}{\partial t^2} [F(x)G(t)] &= \alpha^2 \frac{\partial^2}{\partial x^2} [F(x)G(t)] \\ FG_{tt} &= \alpha^2 F_{xx}G \end{aligned}$$

Step #3: Separate variables:

$$\begin{aligned}\frac{FG_{tt}}{\alpha^2 FG} &= \frac{\alpha^2 F_{xx} G}{\alpha^2 FG} \\ \frac{G_{tt}}{\alpha^2 G} &= \frac{F_{xx}}{F} = -\lambda \\ G_{tt} + \alpha^2 \lambda G &= 0, \quad F_{xx} + \lambda F = 0\end{aligned}$$

We assume that $F(x)$ and $G(t)$ are not identically zero throughout the domain, thus dividing by $F(x)G(t)$ is mathematically acceptable.

On the middle line we see that $\frac{G_{tt}}{\alpha^2 G}$ is only a function of y ; $\frac{F_{xx}}{F}$ is only a function of x and yet they must be equal to each other for all values of x and y . The only way this makes sense is if they are both, in fact, constant. We will denote this constant $-\lambda$.

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

The boundary conditions must be applied to the separated equation for $F(x)$.¹

$$F_{xx} + \lambda F = 0, \quad F(0) = 0, \quad F(L) = 0, \quad 0 < x < L$$

We need to examine all possible values of λ .

$\lambda = 0$:

$$\begin{aligned}F_{xx} &= 0 \\ F(x) &= c_1 x + c_2 \\ F(0) &= c_1(0) + c_2 = 0 \\ \Rightarrow c_2 &= 0 \\ F(L) &= c_1(L) = 0 \\ \Rightarrow c_1 &= 0\end{aligned}$$

Thus we will disregard $\lambda = 0$ since, in that case, only the trivial solution satisfies the governing equation and boundary conditions.

$\lambda < 0$: Here we will set $\lambda = -\nu^2$, $\nu > 0$.

$$\begin{aligned}F_{xx} - \nu^2 F &= 0 \\ F(x) &= c_1 \cosh \nu x + c_2 \sinh \nu x \\ F(0) &= c_1 \cosh 0 + c_2 \sinh 0 \\ F(0) &= c_1 + 0 = 0 \Rightarrow c_1 = 0 \\ F(L) &= c_2 \sinh \nu L = 0\end{aligned}$$

¹ As with the heat equation in Lecture 23, the only way $G(t)$ can satisfy the homogeneous spatial boundary conditions would be for us to set $G(t) = 0$. Thus the product solution would be $u(x,t) = F(x)G(t) = F(x)(0) = 0$. Obviously a trivial solution $u(x,t) = 0$ is not what we are looking for.

This analysis is identical to what we carried out in the last lecture for the heat equation. It is worth doing this a few times just to make sure you know what you are doing. After that, you may decide that it is okay to skip to the answer. It can be risky to "skip to the answer" so do not let me tempt you away from the straight-and-narrow path of always thoroughly looking for valid eigenvalues.

Note again that we use the $\cosh()$ and $\sinh()$ form of the solution since the domain is bounded.

We once again recall that $\sinh x$ is strictly positive for $x > 0$. Therefore $c_2 = 0$ and, again, only the trivial solution satisfies the governing equation and boundary conditions for the case $\lambda < 0$. Therefore we will discard this possibility.

$\lambda > 0$: Here we will set $\lambda = \nu^2$, $\nu > 0$.

$$\begin{aligned} F_{xx} + \nu^2 F &= 0 \\ F(x) &= c_1 \cos \nu x + c_2 \sin \nu x \\ F(0) &= c_1 \cos 0 + c_2 \sin 0 = 0 \\ F(0) &= c_1 + 0 = 0 \Rightarrow c_1 = 0 \\ F(L) &= c_2 \sin \nu L = 0 \end{aligned}$$

Again we see that $\lambda > 0$ bears fruit; our eigenvalues are $\nu_n = \frac{n\pi}{L}$ and eigenfunctions are $F_n(x) = \sin \frac{n\pi x}{L}$. This should not be surprising. If we have the same separated equation and the same boundary conditions (at least in x -direction) we should expect the same eigenvalues and eigenfunctions.

In this case, the separated equation for $G(t)$ is now:

$$\begin{aligned} G_{tt} + \alpha^2 \nu^2 G &= 0 \\ G(t) &= c_1 \cos \alpha \nu t + c_2 \sin \alpha \nu t \end{aligned}$$

We combine these values of ν_n with $F(x)$ and $G(x)$ to get our product solution:

$$\begin{aligned} u(x, t) &= F(x)G(t) = \sum_{n=1}^{\infty} (a_n \cos \alpha \nu_n t + b_n \sin \alpha \nu_n t) \sin \nu_n t \\ u(x, t) &= \sum_{n=1}^{\infty} \left(a_n \cos \alpha \frac{n\pi t}{L} + b_n \sin \alpha \frac{n\pi t}{L} \right) \sin \alpha \frac{n\pi x}{L} \end{aligned}$$

As before, we are combining all of the constants that we can. Since each solution to $G(t)$ had two unknown constants, we the unknown constant in $F(x)$ into both of them.

Step #5: Satisfy the initial conditions.

We now have two infinite sets of unknowns: the coefficients a_n and b_n . We will resolve these constants through the initial conditions.

$$\begin{aligned} u(x, 0) &= \sum_{n=1}^{\infty} \left(a_n \cos 0 + b_n \sin 0 \right) \sin \alpha \frac{n\pi x}{L} \\ &= \sum_{n=1}^{\infty} a_n \sin \alpha \frac{n\pi x}{L} = f(x) \end{aligned}$$

Again we find ourselves with an infinite linear combination of orthogonal functions on the left and a function on the right. Our task is to determine the values of a_n such that they are actually equal. How do we do this? We multiply both sides by a member of the set of orthogonal functions and integrate. This time we will do this explicitly

for a_2 .

$$\underbrace{a_1 \int_0^L \sin \alpha \frac{\pi x}{L} \sin \alpha \frac{2\pi x}{L} dx}_{0} + a_2 \int_0^L \sin \left(\alpha \frac{2\pi x}{L} \right)^2 dx + \dots \text{ all zeros} = \int_0^L f(x) \sin \alpha_n \frac{2\pi x}{L} dx$$

So

$$\begin{aligned} a_n &= \frac{\int_0^L f(x) \sin \alpha_n \frac{n\pi x}{L} dx}{\int_0^L \sin \left(\alpha \frac{n\pi x}{L} \right)^2 dx} \\ &= \frac{2}{L} \int_0^L f(x) \sin \alpha_n \frac{n\pi x}{L} dx \end{aligned}$$

This defines the values for all a_n . We still need to deal with the b_n so we apply the other initial condition:

$$\begin{aligned} u_t(x, 0) &= \sum_{n=1}^{\infty} \left(-a_n \alpha \frac{n\pi}{L} \sin 0 + b_n \alpha \frac{n\pi}{L} \cos 0 \right) \sin \alpha \frac{n\pi x}{L} \\ &= \sum_{n=1}^{\infty} b_n \alpha \frac{n\pi}{L} \sin \alpha \frac{n\pi x}{L} = g(x) \end{aligned}$$

Alas we are in familiar territory now. To find the values of b_n we multiply both sides of the equation by $\sin \alpha_n \frac{n\pi x}{L}$ and integrate.

Do **not** forget to include the additional constants we gained through taking the derivative of the solution with respect to t .

$$\begin{aligned} b_n &= \frac{\int_0^L g(x) \sin \alpha \frac{n\pi x}{L} dx}{\alpha \frac{n\pi}{L} \int_0^L \sin \left(\alpha \frac{n\pi x}{L} \right)^2 dx} \\ &= \frac{\int_0^L g(x) \sin \alpha \frac{n\pi x}{L} dx}{\alpha \frac{n\pi}{L} \frac{L}{2}} \\ &= \frac{2}{\alpha n \pi} \int_0^L g(x) \sin \alpha \frac{n\pi x}{L} dx \end{aligned}$$

In summary, our solution to the wave equation is:

$$\begin{aligned} u(x, t) &= \sum_{n=1}^{\infty} \left(a_n \cos \alpha \frac{n\pi t}{L} + b_n \sin \alpha \frac{n\pi t}{L} \right) \sin \alpha \frac{n\pi x}{L} \\ a_n &= \frac{2}{L} \int_0^L f(x) \sin \alpha_n \frac{n\pi x}{L} dx \\ b_n &= \frac{2}{\alpha n \pi} \int_0^L g(x) \sin \alpha \frac{n\pi x}{L} dx \end{aligned}$$

MATLAB Implementation

As with the heat equation, we really cannot extract much insight by inspecting the solution formulas. We need to make a plot and to do so we will use MATLAB to represent an approximate solution. The MATLAB code is given below:

```

1 clear
2 clc
3 close 'all'

4 %% Example Problem
5 L = 3;
6 alpha_sq = 1;% T/rho
7 alpha = sqrt(alpha_sq);
8 N = 50;

9 f = @(x) ex1(x,L);
10 g = @(x) x.*0;

11 for n = 1:N
12 % compute an
13 an = (2/L)*integral(@(x) f(x).*sin(n*pi*x./L),0,L);
14 % compute bn
15 bn = ...
16 (2/(alpha*n*pi))*...
17 integral(@(x) g(x).*sin(n*pi*x./L),0,L);

18 % update the approximate solution
19 u = @(x,t) u(x,t) + ...
20 (an*cos(alpha.*n*pi*t./L) + ...
21 bn*sin(alpha.*n*pi*t./L)).*sin(n*pi*x./L);
22 end

23 %% Fixed Plot, single time step
24 ts = 3.0;
25 figure(3)
26 plot(X,u(X,ts),'-b','LineWidth',3);
27 title_str = ...
28 sprintf('Lecture 24 Example, t = %g',ts);
29 title(title_str,'fontsize',16,'fontWeight','bold');
30 xlabel('X','fontsize',14,'fontWeight','bold');
31 ylabel('u(X,T)','fontsize',14,'fontWeight','bold');
32 grid on
33 set(gca,'fontsize',12,'fontWeight','bold');
34 axis([0 L -2.0 2.0]);
35
36 %% Local functions
37 function y = ex1(x,L)
38 [m,n] = size(x);
39 y = nan(m,n);
40 for i = 1:length(x)
41 if (x(i)>0)&&(x(i) < L/2)
42 y(i) = (2/3).*x(i);
43 elseif(x(i) >= L/2) && (x(i)<L)
44 y(i) = (2/3)*(L - x(i));
45 end
46 end
47 end
48
49
50
51
52

```

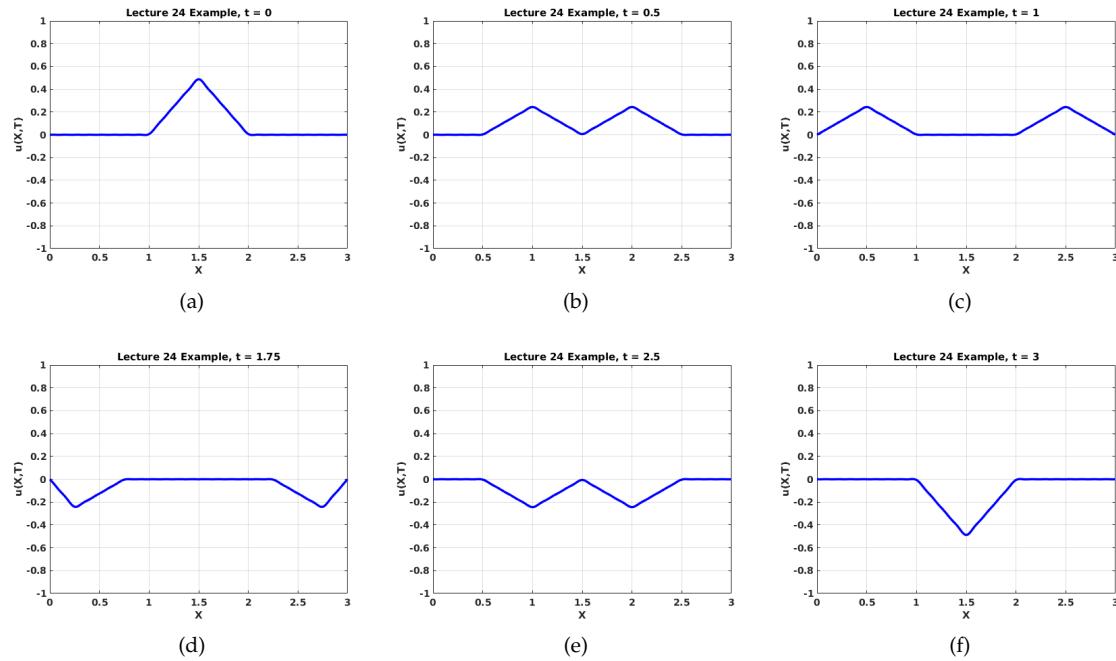
For this example we will set the wave speed $\alpha = 1$, the length $L = 3$ and the initial conditions as:

$$f(x) = \begin{cases} \frac{2}{3}x, & 0 < x < \frac{3}{2} \\ \frac{2}{3}(3-x), & \frac{3}{2} \leq x < 3 \end{cases}$$

$$g(x) = 0$$

The resulting solution is plotted in Figure 31.

Figure 31: Wave equation solution from $t = 0$ to $t = 3$.



Lecture 25 - Heat and Wave Equation with MATLAB

Objectives

- Describe the details of how to use MATLAB to construct approximate solutions to the heat and wave equations.
- Illustrate how to create an animation with MATLAB; and
- show an effective method for creating 2D plots with MATLAB.

Introduction

In the past two lectures we have invested considerable time in the solution of two boundary value problems using the method of separation of variables. When I teach this course I find that, during those lectures, there is little time available to spend going into the details of the MATLAB code created, essentially, to visualize the solution. Nonetheless it is often the case that a large fraction of the students have only modest proficiency in using MATLAB. In this lecture we focus on the MATLAB details. The hope is that students will come away with at least an introduction to the bare-minimum of MATLAB tools that will allow them to complete homework assignments.

Proficiency levels vary: students have as much as two years experience using MATLAB to as little as two or three weeks.

Example Heat Equation

Problem Statement: Find the temperature $u(x, t)$ of a bar of silver of length 10 cm. The density is 10.6 g/cm³, thermal conductivity is 1.04 cal/cm-s-°C, and the specific heat is 0.056 cal/g-°C. The bar is perfectly insulated laterally with ends kept at 0°C. The initial temperature is given by: $f(x) = 4 - 0.8|x - 5|$ °C. From this physical description of the problem, we formulate the following boundary value problem:

Take a moment to consider the units used in this boundary value problem. If you believe the units provided for density, thermal conductivity, and specific heat, simple “unit arithmetic” shows that the thermal diffusivity should have units of cm²/s. What are the units of $\partial^2 u / \partial x^2$? Answer: °C/cm². What are the units of $\partial u / \partial t$? Answer: °C/s. From this it should be clear that, indeed, the units are the same on the left and right side of the governing equation—as it *must* be in order to be correct. My point is that it does not take a tremendous amount of mathematical skill to check these things but you should *always* check them. If you do so, then you will definitely increase your confidence that you know what is going on; you may also save yourself from an embarrassing error.

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < 10, \quad t > 0$

Boundary Conditions : $u(0, t) = 0, \quad u(10, t) = 0, \quad t > 0$

Initial Conditions : $u(x, 0) = f(x) = 4 - 0.8|x - 5|, \quad 0 < x < 10$

Being able to translate this kind of a description of a problem into a properly formulated boundary value problem that you can solve is a key skill you should develop from this course.

Analytic Solution:

$$u(x, t) = \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{10} e^{-\left(\frac{an\pi}{10}\right)^2 t} \quad (94)$$

where $\alpha^2 = \frac{k}{\rho c_p} = \frac{1.04}{(10.6)(0.056)} \text{cm}^2/\text{s}$ and the coefficients c_n are given by:

$$c_n = \frac{2}{10} \int_0^{10} (4 - 0.8|x - 5|) \sin \frac{n\pi x}{10} dx$$

MATLAB tasks:

1. Construct an approximate representation of the solution in MATLAB including $N = 50$ terms of the infinite series.
2. Create a plot of the solution at $t = 0, 1$, and 10 seconds.
3. Create an animation of the time-dependent temperature profile that you can save and incorporate, for example, in a presentation.
4. Visualize the time-dependent temperature profile using a 2D surface plot.

We will tackle these tasks one at a time.

Task #1: Construct an approximate representation of the solution in MATLAB including $N = 50$ terms of the infinite series.

In this section of the code we clean out the workspace and provide basic given input data.

```

clear
clc
close 'all'

%% Heat Equation BVP & Solution
L = 10; % cm
alpha_sq = 1.752; % cm^2/s, thermal diffusivity of silver. ①
N = 50;

F = @(x,n) sin(n.*pi.*x./L); ②
G = @(t,n) exp(-((n.*pi./L).^2)*alpha_sq.*t);

f = @(x) 4 - 0.8*abs(x - 5);

```

① It is a good idea to include units and a brief statement indicating what a variable represents. It may be easy to remember that, for instance, $L=10$, % cm refers to the length but it is less easy to remember that α_{sq} refers to the thermal diffusivity.

② Make sure you fully understand how these anonymous functions work. The snippet: $F = @(x,n) \dots$ should be read: "F is a function of x and n..." The "L" that appears on the right hand side is the same "L" defined as a parameter. The fact that this works is one of the benefits of using anonymous functions. Remember to write these anonymous functions so that they can accept vector inputs for x and n . Built-in functions like `integral()` will fail if the function you supply to be integrated *cannot* accept vector inputs.

Next we will initialize and build—term by term—a truncated version of the infinite series.

```

16 u = @(x,t) o;❸
17 for n = 1:N
18     % essentially doing the sine-series half-wave expansion
19     % compute the coefficient
20     cn = (2/L)*integral(@(x) f(x).*F(x,n),0,L);
21
22     % add the term to the series solution
23     u = @(x,t) u(x,t) + cn*F(x,n).*G(t,n);
24 end

```

At this point, our first task is done. The variable u represents the truncated series solution and we can evaluate the function at any point x or t to get the solution.¹

Task #2: Create a plot of the solution at $t=0$, 1 , and 10 seconds.

Code to complete this is presented in the listing below.

```

%% Plot the result for fixed times ❶
% make a discrete X-axis
Nx = 1000;
X = linspace(0,L,Nx);
figure(1)
plot(X,u(X,0),'-ob',...    ❷
     X,u(X,1),'-g',...    ❸
     X,u(X,10),'--r','MarkerIndices',1:50:Nx,'linewidth',3); ❹
title('Lecture #25 Example','fontsize',16,'fontweight','bold');
xlabel('X [cm]','fontsize',14,'fontweight','bold');
ylabel('u(X,t) [^\circ C]',... ❺
      'fontsize',14,'fontweight','bold');
grid on
set(gca,'fontsize',12,'fontweight','bold');
legend('t = 0','t = 1','t = 10'); ❻

```

❶ The string snippet ' $^\circ \text{C}$ ' is \LaTeX mark-up and is rendered by MATLAB as $^\circ \text{C}$. While not strictly necessary, use of such mark-up can make a plot more attractive.

❷ Obviously use of a legend makes a plot easier to read. MATLAB also includes an optional argument named '`location`' that can be used with values such as: '`northwest`', '`southwest`', '`northeast`', '`southeast`', '`best`' ...etc—that allow you to place a legend such that it does not interfere with reading the plot. Consult the MATLAB documentation for more information about legends.

A plot created by the code snippet above is shown in Figure 32.

Task #3: Create an animation of the time-dependent temperature profile that you can save and incorporate, for example, in a presentation.

For this task we will first create a simple time-dependent plot that you might use for a homework assignment, independent research

❸ On the surface, this does not do much: it simply creates a variable u that is a handle to a function of two variables and sets its initial value to zero. If we did not have this, however, we would have to create a special case in the `for ... end` loop to create it on the first trip through.

¹ Sadly, there isn't anything you can do to prevent a user from evaluating the function at invalid/inappropriate values of x or t . For example, $u(1994, -25)$ is perfectly legal MATLAB code.

❶ The `%%` separates MATLAB code into sections that can be executed independently. Breaking scripts into sections like this can simplify debugging and helps improve code readability.

❷ Familiarize yourself with these "LineSpec" strings. For plots with multiple data series you should try to make it easy to tell the difference between different data series even if the plot is viewed in black and white.

❸ Using the '`MarkerIndices`' argument allows you to specify which data indices get annotated with a marker (when the line specification includes a marker). The value `1:50:Nx` in this case results in 1 out of every 50 data points having a marker applied. Experiment with this and see what the plot looks like if you omit this name-value pair.

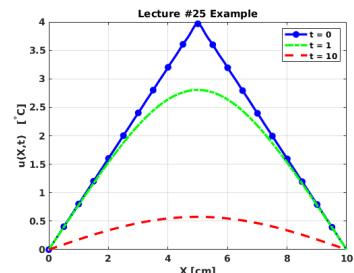


Figure 32: Plot of heat equation example at $t=0, 1$, and 10 seconds

project, or for your capstone. The goal is to gain understanding of the transient behavior of this physical system.

```
%>> Simple time-dependent plot
Tmax = 15; % s
NT = 30;
T = linspace(0,Tmax,NT); ❶
figure(2)
for n = 1:NT
    plot(X,u(X,T(n)),'-b','linewidth',2)
    title_str = sprintf('Lecture #25 Example, t = %5.3g',T(n)); ❷
    title(title_str,'fontsize',16,'fontweight','bold');
    xlabel('X','fontsize',14,'fontweight','bold');
    ylabel('u(X,T)','fontsize',14,'fontweight','bold');
    grid on
    set(gca,'fontsize',12,'fontweight','bold');
    axis([0 L -0.2 4.5]); ❸
    pause(Tmax/(NT-1)); ❹
end
```

❶ This command causes MATLAB to pause slightly before starting the next iteration in the for-loop. The argument to the pause() function is the duration (in seconds) of the pause. For some systems this pause also allows the graphics system a chance to update between iterations in the for-loop. (i.e. if you omit the pause, you may not see your plot update until the very end and miss all of the transient behavior.)

THIS PLOT IS GOOD for routine homework or analysis that you do not intend to present publicly. Sometimes you *do* want to show a time-dependent plot of a system you are analyzing but you do not want to run the MATLAB script that created the plot during the presentation. A good option is to create a movie that can be played from most computers and/or can be embedded in a presentation. The next code block accomplishes this task.

```
%>> Save the time-dependent plot as a Movie
FRAMES(NT) = struct('cdata',[],'colormap',[]); ❺
figure(3)
for n = 1:NT
    plot(X,u(X,T(n)),'-b','linewidth',3);
    title_str = ...
        sprintf('Lecture #25 Example, t = %g ',T(n));
    title(title_str,'fontsize',16,'fontweight','bold');
    xlabel('X','fontsize',14,'fontweight','bold');
    ylabel('u(X,T)','fontsize',14,'fontweight','bold');
    grid on
    set(gca,'fontsize',12,'fontweight','bold');
    axis([0 L -0.2 4.5]);
    drawnow % ensures graphics pipeline is complete/"flushed"
    FRAMES(n) = getframe(gcf); ❻
end

%>> play the movie
fig = figure(4);
```

❶ Discretize time as desired.

❷ The sprintf() allows you to create formatted strings for the title. In this case, we include the time, in seconds. The snippet '%5.3g' is a formatting string; in this case the value returned from T(n) is placed here in a field 5 digits wide with up to 3 digits to the right of the decimal point. The character 'g' tells MATLAB to render the number either in fixed-point notation (e.g. 3.14) or scientific notation (e.g. 1.6e9) whichever is more compact.

❸ This sets the axis limits axis ([xmin xmax ymin ymax]). The default behavior is to re-scale the plot to fit the max/min plotted values. For transient simulations this can make changes to the temperature profile harder to understand. Try running this script without this command to better understand the effect.

❺ This creates an array of structures; each structure has two fields, one for the color-data 'cdata', and one for the color-map 'colormap'. A structure is a data-type that we use infrequently for this course. If you wanted, for instance, to access the 3rd frame color-data you would use the command:

FRAMES(3).cdata

❻ the command gcf means: "get current frame." In this line, the nth frame of the animation is saved to the nth FRAMES structure.

```

movie(fig,FRAMES,10); % last argument is frames-per-second
%
%% Write frames to AVI file
v = VideoWriter('TransientHeat.avi'); ⑦
open(v);
for n = 1:NT
    writeVideo(v,FRAMES(n));
end
close(v); ⑧

```

⑦ This function creates and opens an AVI file to which the `writeVideo()` function can write a video frame-by-frame. See MATLAB documentation for other supported video file types.

⑧ Be a good leader and close any files you open for writing.

Task #4: Visualize the time-dependent temperature profile using a 2D surface plot.

Animations are nice but sometimes the splashy graphics are not needed and you just want to see how the temperature across the domain changes over time in a static image. A surface plot is an excellent way to do this; the MATLAB built-in function that does the job is cunningly named `surf()` and is shown in the listing below.

```

%% Plot the temperature vs time in a 2D plot using the surf
% function
[XX,TT] = meshgrid(X,T); ①
figure(5)
surf(XX,TT,u(XX,TT), 'edgecolor','none'); ②
title('Lecture 25 Surface Plot Example',...
    'fontsize',18,'fontweight','bold');
xlabel('X [cm]', 'fontsize',16,'fontweight','bold');
ylabel('T [s]', 'fontsize',16,'fontweight','bold');
zlabel('u(X,T) [^\circ C]', 'fontsize',16,...
    'fontweight','bold');

```

① The `meshgrid()` function outputs 2D grid coordinates corresponding to the vector inputs for each dimension. The output arrays `XX` and `TT` are suitable for use in the `surf()` function.

② The `surf(XX,YY,ZZ)` function takes at least three arguments. In this case the first two are used by the output of `meshgrid()` and the last is created by supplying the `xx` and `tt` arrays to our approximate solution—`u(x,y)`—which serves as the height (or z-coordinate) of the surface plot at each point. The name-value pair: `'edgecolor','none'` suppresses the (by default) black grid lines that delineate the mesh created with `xx` and `tt`. For high-resolution meshes these grid lines would obscure the color-map used to highlight the solution. (Try omitting this name-value pair and observe the effect.)

The resulting surface plot is shown in Figure 33.

Example Wave Equation

For this we will consider the wave equation example analyzed in Lecture 24. The code may be familiar by this point, but since we did not take the time to go through the MATLAB implementation in detail before, we will take the time here.

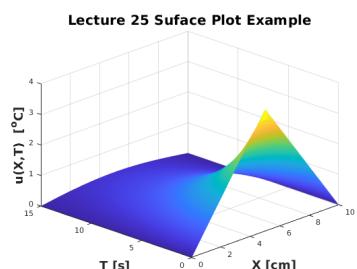


Figure 33: Surface plot of heat equation example.

The boundary value problem is given as: where the length is given

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad a < x < b$$

$$\text{Boundary Conditions : } u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = f(x), \quad u_t(x, 0) = g(x), \quad 0 < x < L$$

by $L = 3$, wave speed is 1, and the initial conditions are given by:

$$f(x) = \begin{cases} \frac{2}{3x}, & 0 < x < \frac{3}{2} \\ \frac{2}{3}(3-x), & \frac{3}{2} \leq x < 3 \end{cases}$$

$$g(x) = 0$$

The analytic solution is:

$$u(x, t) = \sum_{n=1}^{\infty} \left(a_n \cos \alpha \frac{n\pi t}{L} + b_n \sin \alpha \frac{n\pi t}{L} \right) \sin \alpha \frac{n\pi x}{L}$$

$$a_n = \frac{2}{L} \int_0^L f(x) \sin \alpha_n \frac{n\pi x}{L} dx$$

$$b_n = \frac{2}{\alpha n \pi} \int_0^L g(x) \sin \alpha \frac{n\pi x}{L} dx$$

We begin, again, by cleaning out the workspace and command window and closing all figure windows; then we declare basic problem parameters.

```
clear
clc
close 'all'

%% Example Problem
L = 3;
alpha_sq = 1;% T/rho
alpha = sqrt(alpha_sq); ①
N = 50;

f = @(x) ex1(x,L);②
g = @(x) x.*o; ③
```

❶ Recall that $\alpha^2 = T/\rho$ where T is the tension (force) and ρ is the density. Unit analysis shows that α^2 has units of: $mL/t^2/m/L^3 = L^2/s^2$ where m is mass, L is length, and t is time. Thus α has units of L/s and we call it the “wave speed”. Now is also a good time to examine the governing equation of the boundary value problem and confirm to yourself that the units make sense.

❷ Notice that the L from the parameter list is used for the second argument of $ex1(x,L)$.

❸ At first glance it would appear to be easier to simply make the assignment: $g = o;$. We do it this way so that the follow-on code can be written in a way that assumes that g is a function of x —i.e. the code $integral(@(x) g(x).*sin(n*pi*x./L,o,L))$ does not result in an error. In the future, if we replace $g(x) = 0$ with a non-trivial function of x , for example $g(x) = \sin x$, everything will work as expected.

Next we will initialize our approximate solution and built it up term-by-term.

```

u = @(x,t) o;
for n = 1:N
    % compute the coefficients
    an = (2/L)*integral(@(x) f(x).*sin(n*pi*x./L),o,L);
    bn = ...
        (2/(alpha*n*pi))*...
        integral(@(x) g(x).*sin(n*pi*x./L),o,L);
    % update the approximate solution
    u = @(x,t) u(x,t) + ...
        (an*cos(alpha.*n*pi*t./L) + ...
        bn*sin(alpha.*n*pi*t./L)).*sin(n*pi*x./L);
end

```

13
14
15
16
17
18
19
20
21
22
23
24

This equation has two expansion coefficients: a_n and b_n , unlike the heat equation which had only one but incorporation of that added complexity in MATLAB is straightforward.

Now that the approximate solution has been computed, we can plot the results. We may, if we wish, create a dynamic plot much like we did with the heat equation so that we can see the wave behavior in action.

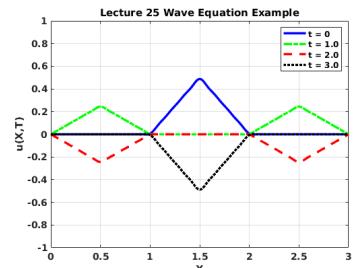
```

%% make discrete space and time space vectors
Tmax = 3;
NT = 50;
T = linspace(0,Tmax,NT);
Nx = 500;
X = linspace(0,L,Nx);

%% create time-dependent plot
figure(1)
for n = 1:NT
    plot(X,u(X,T(n)),'-b','linewidth',3);
    title_str = sprintf('Lecture 24 Example, t = %g ',T(n));
    title(title_str,'fontsize',16,'fontweight','bold');
    xlabel('X','fontsize',14,'fontweight','bold');
    ylabel('u(X,T)','fontsize',14,'fontweight','bold');
    grid on
    set(gca,'fontsize',12,'fontweight','bold');
    axis([0 L -1.5 1.5]);
    pause(Tmax/(NT-1));
end

```

25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44



Or we can make a single plot with multiple data-series as shown in Figure 34:

```

%% fixed plot, multiple data series
figure(2)
plot(X,u(X,0),'-b',...
    X,u(X,1.0),'-.g',...
    X,u(X,2.0),':-r',...
    X,u(X,3.0),':k','linewidth',3);
title('Lecture 25 Wave Equation Example',...
    'fontsize',16,'fontweight','bold');
xlabel('X','fontsize',14,'fontweight','bold');
ylabel('u(X,T)','fontsize',14,...
    'fontweight','bold');
grid on
set(gca,'fontsize',12,'fontweight','bold');

```

45
46
47
48
49
50
51
52
53
54
55
56
57

Figure 34: Plot of wave equation example problem at $t=0, 1.0, 2.0$, and 3.0 sec.

```

axis([0 L -1.0 1.0]);
legend('t = 0','t = 1.0','t = 2.0','t = 3.0',...
'location','best');

```

58
59
60

Lest we forget, we also need to define the function $\text{ex1}(x, L)$. We choose to do this as a *local function* which means that it has to be placed *at the end* of the script file.

```

%% Local functions
function y = ex1(x,L)
[m,n] = size(x);
y = nan(m,n);
for i = 1:length(x)
    if (x(i)>0) && (x(i) < L/2)
        y(i) = (2/3).*x(i);
    elseif(x(i) >= L/2) && (x(i)<L)
        y(i) = (2/3)*(L - x(i));
    end
end
end

```

61
62
63
64
65
66
67
68
69
70
71
72

Assignment #9

1. Consider the heat equation given in the following boundary value problem:

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$

Boundary Conditions : $u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0$

Initial Conditions : $u(x, 0) = f(x) = \begin{cases} 1, & 0 < x < L/2 \\ 0, & L/2 \leq x < L \end{cases}, \quad 0 < x < L$

Solve this boundary value problem using separation of variables. Use MATLAB to represent the solution of this equation for $\alpha^2 = 1$ and $L = 1$. Truncate the series solution at $N = 25$ terms. Plot the solution with separate lines for the solution at $t=0$, $t=0.1$, and $t=0.5$.

2. Solve the heat equation using separation of variables and find the temperature $u(x, t)$ in a rod of length L with thermal diffusivity α^2 , if the initial temperature is $f(x)$ throughout and if the ends $x = 0$ and $x = L$ are insulated.
3. Suppose heat is lost from the lateral surface of a thin rod of length L into a surrounding medium at temperature zero. If the linear law of heat transfer applies, then the heat equation takes on the form:

$$\alpha^2 \frac{\partial^2 u}{\partial x^2} - hu = \frac{\partial u}{\partial t}, \quad 0 < x < L, \quad t > 0$$

where h is a constant. Use separation of variables and find the temperature $u(x, t)$ if the initial temperature is $f(x)$ throughout and the ends $x = 0$ and $x = L$ are insulated. **Note:** when separating variables, keep h with the time-dependent part of the equation.

4. Solve the heat equation using separation of variables subject to the following boundary and initial conditions:

$$u(0, t) = 0, \quad u(100, t) = 0, \quad t > 0$$

$$u(x, 0) = \begin{cases} 0.8x, & 0 \leq x \leq 50 \\ 0.8(100 - x), & 50 < x \leq 100 \end{cases}$$

Use MATLAB to represent the solution of this equation for $\alpha^2 = 1.6352$ and $L = 100$. Truncate the series solution at $N = 25$ terms. Plot the solution with separate data series for the solution at $t=0$, $t=10$, and $t=50$ seconds. Create a surface plot (use the MATLAB built-in function `surf()` with $0 \leq x \leq 100$ as one dimension and $0 \leq t \leq 200$ as the other.

5. Solve the wave equation using separation of variables subject to the given conditions:

$$u(0, t) = 0, \quad u(\pi, t) = 0, \quad t > 0$$

$$u(x, 0) = 0, \quad u_t(x, 0) = \sin x, \quad 0 < x < \pi$$

6. Use separation of variables to solve the wave equation subject to the following conditions:

$$u(0, t) = 0, \quad u(1, t) = 0, \quad t > 0$$

$$u(x, 0) = x(1 - x), \quad u_t(x, 0) = x(1 - x), \quad 0 < x < 1$$

Use MATLAB to represent this solution. Truncate the series solution at $N = 25$ terms. Make a plot that shows the position $u(x, t)$ for $t = 0, 1, 5$, and 10 .

Review Problems #2

List of topics

1. Orthogonal Functions and Fourier Series
 - (a) Orthogonal Functions
 - (b) Fourier series
 - (c) Sturm-Liouville eigenvalue problems
 - (d) Fourier-Bessel and Fourier-Legendre expansions
2. Boundary Value Problems in Rectangular Coordinates
 - (a) Finding product solutions of separable partial differential equations.
 - (b) Classifying partial differential equations.
 - (c) Solving the heat equation and wave equation with various boundary/initial conditions.

Review Problems

1. Suppose the function $f(x) = x^2 + 1$, $0 < x < 3$, is expanded in a Fourier series, a cosine series, and a sine series. Draw a sketch of each expansion from $-3 < x < 3$ and indicate the value to which the expansion converges at $x = 0$ in each case.
2. The product of an odd function $f(x)$ with an odd function $g(x)$ is an _____ function.
3. To you were to expand $f(x) = |x| + 1$, $-\pi < x < \pi$, in a trigonometric series, the series that would converge most quickly would be a _____ series expansion.
4. Consider Chebyshev's differential equation:

$$(1 - x^2) u'' - xu' + n^2 u = 0, \quad -1 < x < 1$$

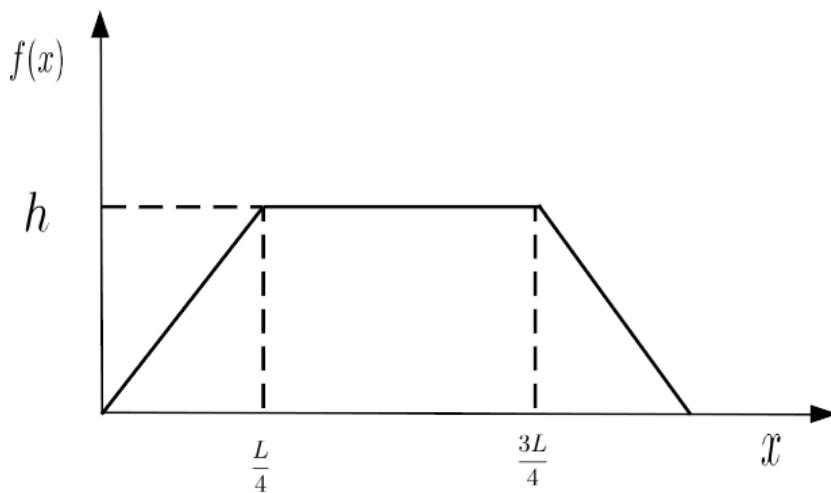
which for integer $n = 0, 1, 2, \dots$, have polynomial solutions called Chebyshev polynomials which are denoted $T_n(x)$. Express Chebyshev's equation in self-adjoint form and write the orthogonality relation for Chebyshev polynomials.

5. Consider a rod of length L coinciding with the interval $[0, L]$ on the x -axis. Set up the boundary-value problem for the temperature $u(x, t)$ where there is heat transfer from the left end into a surrounding medium which is maintained at a temperature of 20° , and the right end is insulated. The initial temperature throughout the rod is $f(x)$.

6. Solve the wave equation subject to the conditions:

$$\begin{aligned} u(0, t) &= 0, \quad u(\pi, t) = 0, \quad t > 0 \\ u(x, 0) &= 0.01 \sin 3x, \quad u_t(x, 0) = 0, \quad 0 < x < \pi \end{aligned}$$

7. Consider a string of length $L = 4$ and $h = 1$ fixed at both ends with initial displacement as shown in the sketch below.



Using MATLAB syntax, complete the local function shown in the space below:

```
%% Local Function to implement IC
function y = ICfun(x)
```

```
[m,n] = size(x);
```

```
y = nan(m,n);
```

```
for i = 1:length(x)
```

```
end
end
```

Lecture 26 - Laplace's Equation

Objectives

- Solve a boundary value problem based on Laplace's equation representing steady-state temperature in a rectangular domain.
- Show how to use the superposition principle to solve Laplace's equation with multiple non-homogeneous boundary conditions.

Laplace Equation Example

Consider the system depicted in Figure 35 and described by the following boundary value problem based on Laplace's equation.

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad 0 < x < a, \quad 0 < y < b$$

$$\begin{array}{ll} \text{Boundary Conditions : } & u(x,0)=0 \quad u_x(0,y)=0 \\ & u(x,b)=f(x) \quad u_x(a,y)=0 \end{array}$$

We will find the solution to this boundary value problem using separation of variables.

Step #1: Assume a product solution.

$$u(x,y) = F(x)G(y)$$

Step #2: Insert proposed solution into the governing equation.

$$\frac{\partial^2}{\partial x^2} [F(x)G(y)] + \frac{\partial^2}{\partial y^2} [F(x)G(y)] = 0$$

$$F_{xx}G + FG_{yy} = 0$$

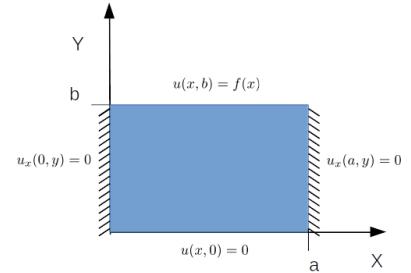


Figure 35: Schematic of example Laplace's equation problem.

Step #3: Separate variables by dividing by $F(x)G(y)$:

$$\begin{aligned} \frac{F_{xx}G}{FG} + \frac{FG_{yy}}{FG} &= 0 \\ \frac{F_{xx}}{F} + \frac{G_{yy}}{G} &= 0 \\ \frac{F_{xx}}{F} = -\frac{G_{yy}}{G} &= -\lambda \\ F_{xx} + \lambda F &= 0 \\ G_{yy} - \lambda G &= 0 \end{aligned}$$

This gives us two separated boundary value problems to solve.

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

In order to determine allowable values for λ , we find solutions to the separated boundary value problem that has *all homogeneous boundary conditions*.¹ So we will examine $F_{xx} + \lambda F = 0$. As usual, we will have to check for all possible values of λ .

$\lambda = 0$:

$$\begin{aligned} F_{xx} &= 0 \\ F(x) &= c_1 x + c_2 \\ \Rightarrow F_x &= c_1 \\ F_x(0) &= c_1 = 0 \\ F_x(a) &= 0 \text{ (satisfied)} \end{aligned}$$

We see that, while c_1 must be zero, $F(x) = c_2$ satisfies both boundary conditions for any value of c_2 . Thus $\lambda = 0$ is an acceptable eigenvalue and the corresponding eigenfunction is a constant.

$\lambda < 0$: To ensure λ is negative we will set $\lambda = -\alpha^2$, $\alpha > 0$.

$$\begin{aligned} F_{xx} - \alpha^2 F &= 0 \\ F(x) &= c_1 \cosh \alpha x + c_2 \sinh \alpha x \\ F_x(x) &= \alpha c_1 \sinh \alpha x + \alpha c_2 \cosh \alpha x \\ F_x(0) &= \alpha c_1(0) + \alpha c_2(1) = 0 \\ \Rightarrow c_2 &= 0 \\ F_x(a) &= \alpha c_1 \sinh a\alpha = 0 \\ \Rightarrow c_1 &= 0 \end{aligned}$$

Therefore only the trivial solution $F(x) = 0$ satisfies the separated equation and boundary conditions if $\lambda < 0$.

Once again we assume that neither $F(x)$ nor $G(y)$ are identically equal to zero throughout the domain, therefore it is mathematically acceptable for them to appear in the denominator.

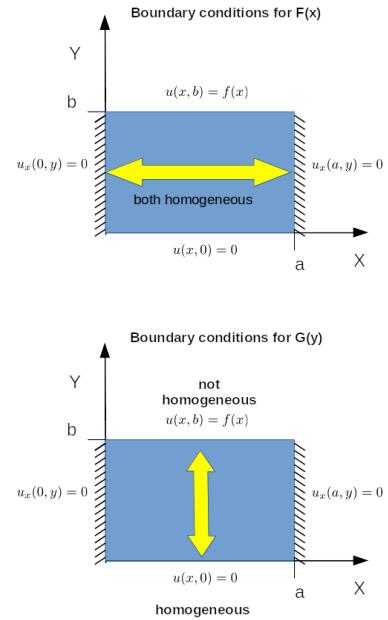


Figure 36: Pairs of boundary conditions for Laplace's equation.

¹ If neither separated boundary value problem has all homogeneous boundary conditions, you will need to solve the problem in multiple phases and use the superposition principle to obtain an answer. This is discussed in the last section of this lecture.

Since the domain is bounded between $0 < x < a$, we will use the \cosh and \sinh form of the solution.

For this last two lines, recall that $\sinh x$ is positive for all $x > 0$.

$\lambda > 0$: To insure λ is positive we will set $\lambda = \alpha^2$, $\alpha > 0$.

$$\begin{aligned} F_{xx} + \alpha^2 F &= 0 \\ F(x) &= c_1 \cos \alpha x + c_2 \sin \alpha x \\ F_x(x) &= -\alpha c_1 \sin \alpha x + \alpha c_2 \cos \alpha x \\ F_x(0) &= -\alpha c_1(0) + \alpha c_2(1) = 0 \\ \Rightarrow c_2 &= 0 \\ F_x(a) &= -\alpha c_1 \sin \alpha a = 0 \end{aligned}$$

This last condition, $\sin \alpha a = 0$, will be satisfied whenever αa is an integer multiple of π , therefore the eigenvalues are: $\alpha_n = n\pi/a$, $n \in \mathbb{Z}^+$. The corresponding eigenfunctions are: $F_n(x) = c_1 \cos \alpha_n x$.

Note that we do not include $n = 0$ since $\alpha > 0$.

In summary, non-trivial eigenfunctions exist when $\lambda = 0$ and when $\lambda > 0$. We need to find the corresponding solutions to $G(y)$ for these cases.

$\lambda = 0$:

$$\begin{aligned} G_{yy} &= 0 \\ G(y) &= c_3 y + c_4 \\ G(0) &= c_3(0) + c_4 = 0 \\ \Rightarrow c_4 &= 0 \end{aligned}$$

Apply the homogeneous boundary condition on $G(y)$.

So the product solution for the case $\lambda = 0$ is: $F(x)G(y) = c_1 c_3 y = c_0 y$

$\lambda = \alpha_n^2$, $\alpha_n = n\pi/a$.

$$\begin{aligned} G_{yy} - \alpha_n^2 G &= 0 \\ G(y) &= c_3 \cosh \alpha_n y + c_4 \sinh \alpha_n y \\ G(0) &= c_3(1) + c_4(0) \\ \Rightarrow c_3 &= 0 \\ \Rightarrow G_n(y) &= c_4 \sinh \alpha_n y \end{aligned}$$

Therefore the product solution for the case $\lambda > 0$ is: $F_n(x)G_n(y) = c_n \cos \alpha_n x \sinh \alpha_n y$.

We combine all previous constants for each eigenfunction into c_n .

The full product solution is, therefore:

$$u(x, y) = c_0 y + \sum_{n=1}^{\infty} c_n \cos \frac{n\pi x}{a} \sinh \frac{n\pi y}{a}$$

Step #5: Apply (remaining) boundary condition to determine unknown coefficients.

The boundary condition that we have not yet used is the non-homogeneous condition applied at $u(x, b)$. We must find suitable values for a_0 and $a_n, n = 1, 2, 3, \dots$ such that:

$$u(x, b) = a_0 b + \sum_{n=1}^{\infty} a_n \cos \frac{n\pi x}{a} \sinh \frac{n\pi b}{a} = f(x)$$

We will find the coefficients the same way we always have: multiply both sides by an orthogonal function and integrate. The eigenfunctions identified above comprise our set of orthogonal functions.

For $n = 0$:

$$\begin{aligned} c_0 b (1) \int_0^a dx + \sum_{n=1}^{\infty} a_n \left[\int_0^a (1) \cos \frac{n\pi x}{a} dx \right] \sinh \frac{n\pi b}{a} &= \int_0^a f(x) (1) dx \\ c_0 (b)(a) &= \int_0^a f(x) dx \\ c_0 &= \frac{1}{ab} \int_0^a f(x) dx \end{aligned}$$

For $n = 1, 2, 3, \dots$ the process is similar, we will show the calculation explicitly for $n = 1$:

$$\begin{aligned} c_0 b \int_0^a \cos \frac{n\pi x}{a} dx + c_1 \left[\int_0^a \cos \left(\frac{\pi x}{a} \right)^2 dx \right] \sinh \frac{\pi b}{a} &+ \\ \sum_{n=2}^{\infty} \left[\cos \frac{n\pi x}{a} \cos \frac{\pi x}{a} dx \right] \sinh \frac{n\pi b}{a} &= \int_0^a f(x) \cos \frac{\pi x}{a} dx \\ \Rightarrow c_1 \left(\frac{a}{2} \right) \sinh \frac{\pi b}{a} &= \int_0^a f(x) \cos \frac{\pi x}{a} dx \\ c_1 &= \frac{2 \int_0^a f(x) \cos \frac{\pi x}{a} dx}{a \sinh \frac{\pi b}{a}} \end{aligned}$$

and, in general:

$$c_n = \frac{2 \int_0^a f(x) \cos \frac{n\pi x}{a} dx}{a \sinh \frac{n\pi b}{a}}$$

In summary, the solution to this boundary value problem is:

$$u(x, y) = c_0 y + \sum_{n=1}^{\infty} c_n \cos \frac{n\pi x}{a} \sinh \frac{n\pi y}{a}$$

where

$$\begin{aligned} c_0 &= \frac{1}{ab} \int_0^a f(x) dx \\ c_n &= \frac{2 \int_0^a f(x) \cos \frac{n\pi x}{a} dx}{a \sinh \frac{n\pi b}{a}} \end{aligned}$$

Some notes on this process:

1. The eigenfunctions are solutions to the separated equation for $F(x)$ and they are orthogonal over the range $x \in [0, a]$; that is why the integrals are all from 0 to a .
2. $F_0(x) = 1$ is orthogonal to $F_n(x) = \cos \frac{n\pi x}{a}$ so all of the terms in the summation are equal to zero.

You should confirm that:
 $\int_0^a \cos \left(\frac{n\pi x}{a} \right)^2 dx = \frac{a}{2}$.

Implementation in MATLAB

To actually calculate and plot a solution we need to specify values for a , b , and $f(x)$ as well as choose a finite number of terms to the infinite series solution. For this example we will set $a = 3$, $b = 5$, and we will use $N = 25$ terms of the infinite series and we will define the type 1 boundary condition at $y = b$ to be:

$$f(x) = \begin{cases} x^2, & 0 < x < 3/2 \\ (3/2)^2, & 3/2 \leq x < 3 \end{cases}$$

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

clear
clc
close 'all'

%% Set Parameters
a = 5;
b = 3;
N = 15;

f = @(x) ex1(x,a);

%% define the eigenvalues and eigenfunctions
alpha = @(n) n.*pi./a;
F = @(x,n) cos(alpha(n).*x);
G = @(y,n) sinh(alpha(n).*y);

%% Compute coefficients
% compute Ao
co = (1/(a*b))*integral(@(x) f(x),0,a);

% initialize solution
u = @(x,y) co.*y;
for n = 1:N
    % compute An
    cn = (2./(a*G(b,n))).*...
        integral(@(x) f(x).*F(x,n),0,a);
    % update the approximate solution
    u = @(x,y) u(x,y) + cn*F(x,n).*G(y,n);
end

```

Since this is a two-dimensional geometry, a surface plot is appropriate for visualizing the solution.

```

30
31
32
33
34
35
36
37
38
39
40
41

%% Make discrete spatial coordinate axes
Nx = ceil(100*a); % "ceil" rounds up to next highest integer
Ny = ceil(100*b);
X = linspace(0,a,Nx);
Y = linspace(0,b,Ny);

[XX,YY] = meshgrid(X,Y);

%% Plot the solution in a 2D plot using surf
figure(1)
surf(XX,YY,u(XX,YY),'edgecolor','none');
title ("Lecture 26 Laplace's Equation Example",...❶

```

❶ Since the string used for the title includes an apostrophe, double-quotes must be used to enclose the string.

```

'fontsize',16,'fontweight','bold');
xlabel('X','fontsize',14,'fontweight','bold');
ylabel('Y','fontsize',14,'fontweight','bold');
zlabel('u(X,Y)','fontsize',14,'fontweight','bold');
set(gca,'fontsize',12,'fontweight','bold');

```

42
43
44
45
46

We have saved the definition for $\text{ex1}(x)$ until last since, as usual, we have implemented it as a local function and it must come at the end of the script file.

```

%% Local functions
function y = ex1(x,a)
[m,n] = size(x);
y = nan(m,n);
for i = 1:length(x)
    if(x(i)>= 0) && (x(i)<a/2)
        y(i) = x(i)^2;
    elseif(x(i) >= a/2) && (x(i)<a)
        y(i) = (a/2)^2;
    end
end
end

```

47
48
49
50
51
52
53
54
55
56
57
58

The resulting plot is shown in Figure 37. Readers are strongly encouraged to run this script in MATLAB and examine the output carefully and satisfy yourself that it, at a minimum, meets the specified boundary conditions.

Superposition Principle

In the last example problem, all of the boundary conditions were homogeneous except for along the top edge of the rectangular domain at $y = b$. We were able to use separation of variables and find a solution because there was one spatial dimension along which *both boundaries were homogeneous*. Specifically, the boundaries at $x = 0$ and $x = a$ had homogeneous type 2 boundary conditions.

What if, instead, we had boundary conditions as depicted by Figure 38. If these were boundary conditions for Laplace's equation, we would carry out separation of variables to derive the two separated boundary value problems just as before.

$$F_{xx} + \lambda F = 0$$

$$G_{yy} - \lambda G = 0$$

Suppose we assumed $\lambda = 0$ and tried to find solutions for $F(x)$, we would get:

$$\begin{aligned} F(x) &= c_1 x + c_2 \\ F(0) &= c_1(0) + c_2 = 0 \\ \Rightarrow c_2 &= 0 \\ F(a) &= c_1(a) = g(y) \leftarrow \text{Problem!!} \end{aligned}$$

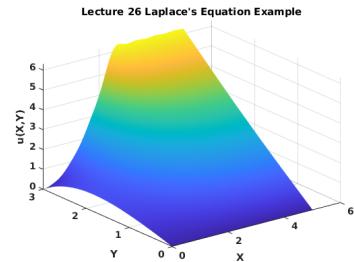


Figure 37: Surface plot of solution to example problem.

In all of the boundary value problems we have solved so far, this condition has always been met. In the heat equation and wave equation boundary value problems, there were always homogeneous boundary conditions in the separated boundary value problem for the spatial independent variable (x). The separated boundary value problem for the temporal independent variable (t) had non-homogeneous boundary (initial) conditions.

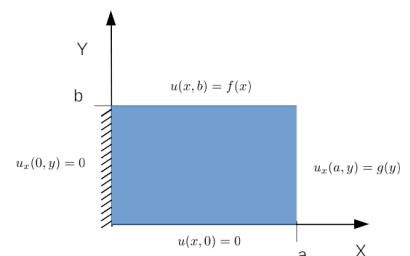


Figure 38: Neither spatial dimension has all homogeneous boundary conditions.

Everything is going fine until we have the condition $c_1(a) = g(y)$. Unless $g(y)$ is a constant: a) no value of c_1 will satisfy $g(y)$ for all values of $y \in [0, b]$; and b) there is nothing we can do about it. We are simply stuck. The same problem would occur if λ were non-zero or if we did the same analysis on $G(y)$. We need to do something different.

WHAT WE WILL DO is this: decompose the problem into two boundary value problems: Problem A and Problem B as is illustrated in Figure 39. Each boundary value problem in this decomposition will have one dimension for which there are homogeneous boundary conditions on both boundaries.

	Problem A	Problem B
Governing Equation	$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0$	$\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} = 0$
Boundary Conditions	$v_x(0,y)=0, v(x,0)=0$ $v_x(a,y)=0, v(x,b)=f(x)$	$w_x(0,y)=0, w(x,0)=0$ $w_x(a,y)=g(x), w(x,b)=0$

We then solve each boundary value problem to get $v(x, y)$ and $w(x, y)$. The solution to the original problem is the *superposition* (or just: sum) of the solutions to the sub-problems:

$$u(x, y) = v(x, y) + w(x, y)$$

Notes:

1. This superposition method *requires* that the governing equation and boundary conditions for the boundary value problem all be linear.
2. The method also depends on the fact that the governing equation is homogeneous.
3. You should break the problem up into as many sub-problems as is required so that, in each sub-problem, there is one dimension (spatial or temporal) that has homogeneous conditions at all boundaries.

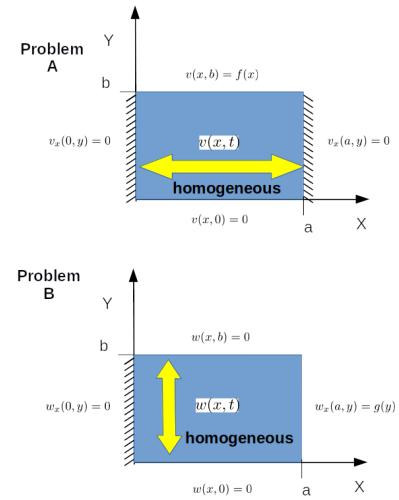


Figure 39: Superposition of two BVPs each with homogeneous boundary conditions in one dimension.

Lecture 27 - Non-homogeneous Problems

Objectives

- Demonstrate a method for solving some non-homogeneous BVPs of the following type:
 1. Non-homogeneous term in the PDE that is a function of no more than one independent variable; and/or
 2. constant non-homogeneous term in a boundary condition.
- Do an example problem.

Time-Independent PDEs and BCs

Consider the following BVP based on the heat equation:

$$\text{Governing Equation : } \frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + S(x), \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$$

$$\text{Boundary Conditions : } u(0, t) = u_0, \quad u(L, t) = u_1, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = f(x), \quad 0 < x < L$$

where u_0 and u_1 are constants. In this boundary value problem both the governing equation and the boundary conditions are non-homogeneous.

Since the governing equation is non-homogeneous, it is not separable. In order to see why, let us begin the separation process:

$$\begin{aligned} u(x, t) &= F(x)G(t) \\ \alpha^2 \frac{\partial^2}{\partial x^2} [F(x)G(t)] + S(x) &= \frac{\partial}{\partial t} [F(x)G(t)] \\ \alpha^2 F_{xx}G + S &= FG_t \\ \frac{\alpha^2 F_{xx}G}{\alpha^2 FG} + \frac{S}{\alpha^2 FG} &= \frac{FG_t}{\alpha^2 FG} \\ \frac{F_{xx}}{F} + \underbrace{\frac{S}{\alpha^2 FG}}_{\text{Problem!}} &= \frac{G_t}{\alpha^2 G} \end{aligned}$$

Here we carry out the steps of separation of variables just as we did in Lecture 23.

The term $S/\alpha^2 FG$ is a function of both x and t and cannot be separated.

The non-homogeneous boundary conditions are also a problem.

Suppose we were able to complete the separation process and derive separated boundary value problems for $F(x)$ and $G(t)$. We then try to apply the spatial boundary conditions to $F(x)$:

$$\begin{aligned} u(0, t) &= F(0)G(t) = u_0 \\ u(L, t) &= F(L)G(t) = u_1 \end{aligned}$$

If we had homogeneous boundary conditions and $u_0 = 0$ —like we have always had before—we could just require $F(0) = 0$ and be done with it.¹ But if $u_1 \neq 0$ we are stuck. We cannot simply say: $F(0) = u_0/G(t)$; even if we knew what $G(t)$ was it could only work in the case that $u_0/G(t)$ were a constant. The same problem applies to the boundary at $x = L$. If we are to use separation of variables, we need to find a way to make both the governing equation and the boundary conditions homogeneous.

Approach:

- Assume a solution of the form: $u(x, t) = v(x, t) + \psi(x)$. Inserting this solution into our governing equation gives us:

$$\begin{aligned} \frac{\partial}{\partial t} [v(x, t) + \psi(x)] &= \alpha^2 \frac{\partial^2}{\partial x^2} [v(x, t) + \psi(x)] + S(x) \\ v_t + \psi_t &\stackrel{0}{=} \alpha^2 v_{xx} + \alpha^2 \psi_{xx} + S(x) \end{aligned}$$

The boundary conditions become:

$$\begin{aligned} u(0, t) &= v(0, t) + \psi(0) = u_0 \\ u(L, t) &= v(L, t) + \psi(L) = u_1 \end{aligned}$$

and the initial condition becomes:

$$u(x, 0) = v(x, 0) + \psi(x) = f(x)$$

- Decompose the original boundary value problem into a new, homogeneous boundary value problem for $v(x, t)$ and non-homogeneous boundary value problem for $\psi(x)$:

Governing Equation : $\frac{\partial v}{\partial t} = \alpha^2 \frac{\partial^2 v}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$

Boundary Conditions : $v(0, t) = 0, \quad v(L, t) = 0, \quad t > 0$

Initial Conditions : $v(x, 0) = f(x) - \psi(x), \quad 0 < x < L$

¹ Zero is special because if $F(0) = 0$, it does not matter what $G(t)$ is, we still satisfy the condition.

$$\begin{aligned}\alpha^2 \psi_{xx} + S(x) &= 0 \\ \psi(0) = u_0, \quad \psi(L) &= u_1\end{aligned}$$

where the non-homogeneous terms from the governing equation and boundary conditions have been absorbed in the equation for $\psi(x)$. The initial condition for $v(x, t)$ remains non-homogeneous but is modified to account for $\psi(x)$.

We solve for $\psi(x)$, then solve for $v(x, t)$ and add the results together to get $u(x, t)$.

Example: Consider the following boundary value problem:

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + S, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$

Boundary Conditions : $u(0, t) = 0, \quad u(L, t) = u_1, \quad t > 0$

Initial Conditions : $u(x, 0) = f(x), \quad 0 < x < L$

where S and u_1 are constants, as are α^2 and L ; and $f(x)$ is a given function.

Step #1: Substitute $u(x, t) = v(x, t) + \psi(x)$.

Governing Equation : $\frac{\partial v}{\partial t} = \alpha^2 \frac{\partial^2 v}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < L, \quad t > 0$

Boundary Conditions : $v(0, t) = 0, \quad v(L, t) = 0, \quad t > 0$

Initial Conditions : $v(x, 0) = f(x) - \psi(x), \quad 0 < x < L$

$$\begin{aligned}\alpha^2 \psi_{xx} + r &= 0 \\ \psi(0) = 0, \quad \psi(L) &= u_1\end{aligned}$$

Step #2: Solve for $\psi(x)$.

$$\begin{aligned}\alpha^2 \psi_{xx} + r &= 0 \\ \psi_{xx}(x) &= -\frac{r}{\alpha^2} \\ \psi_x(x) &= -\frac{r}{\alpha^2} x + c_1 \\ \psi(x) &= -\frac{r}{2\alpha^2} x^2 + c_1 x + c_2\end{aligned}$$

Readers are strongly encouraged to look at the boundary value problems for $v(x, t)$ and $\psi(x)$ and see that if you "add them up" you recover the original boundary value problem for $u(x, t)$.

apply boundary conditions

$$\begin{aligned}\psi(0) &= c_2 = 0 \\ \Rightarrow c_2 &= 0 \\ \psi(L) &= -\frac{r}{2\alpha^2}L^2 + c_1L = u_1 \\ \Rightarrow c_1 &= \frac{1}{L} \left(u_1 + \frac{rL^2}{2\alpha^2} \right) \\ \Rightarrow \psi(x) &= -\frac{r}{2\alpha^2}x^2 + \frac{1}{L} \left(u_1 + \frac{rL^2}{2\alpha^2} \right) x\end{aligned}$$

Step #3: Solve for $v(x, t)$.

We have already solved this problem in Lecture 23 and we will not repeat the details here. The solution is:

$$\begin{aligned}v(x, t) &= \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} e^{-(\alpha \frac{n\pi}{L})^2 t} \\ c_n &= \frac{2}{L} \int_0^L [f(x) - \psi(x)] \sin \frac{n\pi x}{L} dx\end{aligned}$$

Step #4: Construct $u(x, t)$ from solutions for $v(x, t)$ and $\psi(x)$.

$$\begin{aligned}u(x, t) &= \psi(x) + v(x, t) \\ u(x, t) &= -\frac{r}{2\alpha^2}x^2 + \frac{1}{L} \left(u_1 + \frac{rL^2}{2\alpha^2} \right) x + \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} e^{-(\alpha \frac{n\pi}{L})^2 t}\end{aligned}$$

In this particular case, while we know from the boundary conditions, that $v(x, t)$ goes to zero as $t \rightarrow \infty$, the function $\psi(x)$ remains constant with time. Conceptually we can think of $\psi(x)$ as being the *steady state* part of the solution while $v(x, t)$ is the *transient* part.

$$u(x, t) = \underbrace{-\frac{r}{2\alpha^2}x^2 + \frac{1}{L} \left(u_1 + \frac{rL^2}{2\alpha^2} \right) x}_{\text{steady state}} + \underbrace{\sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} e^{-(\alpha \frac{n\pi}{L})^2 t}}_{\text{transient}}$$

A plot for the case: $\alpha^2 = 0.5$, $L = 1$, $r = 150$, $u_1 = 100$, and $f(x) = 300x(1-x)$ is shown in Figure 40. Think about the physics for a minute: we have a rod that is heated from a uniform source, r , which gives the roughly parabolic temperature profile throughout the rod except at the ends; the left-hand side is maintained at 0 degrees and the right-hand side is held steady at 100 degrees. Hopefully the answer you see in the plot matches, more-or-less, your expectations as to what the temperature profile *should* look like.

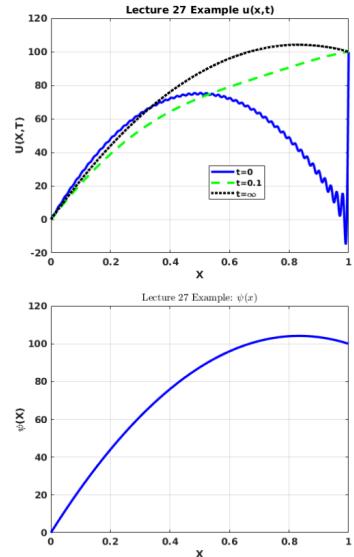


Figure 40: The solution, $u(x, t)$ at various times and the steady-state solution $\psi(x)$.

Take note of how, since the initial condition, $f(x)$, does not match the boundary condition at $x = 1$ there is "wigginess" in the Fourier series representation that "diffuses away" almost immediately. Note also how $u(x, \infty)$ matches $\psi(x)$ as expected.

Lecture 28 - Orthogonal Series Expansion

Objectives

- Apply separation of variables and solve boundary value problems that have solutions in the form of orthogonal series expansions (other than Fourier series).
- Show examples with the heat and wave equation.

For some sets of boundary conditions, the solution to the heat equation is an infinite series that is not a Fourier series. We will show that, apart from details regarding identifying eigenvalues, the implications of this are not large and will have little impact on how you go about computing the solution. All of this is most easily clarified with a couple of examples.

Example #1: Consider the boundary value problem below based on the heat equation.

$$\text{Governing Equation : } \frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}, \quad \alpha > 0, \quad 0 < x < 1, \quad t > 0$$

$$\text{Boundary Conditions : } u(0, t) = 0, \quad u_x(1, t) = -hu(1, t), \quad h > 0, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = 1, \quad 0 < x < 1$$

A schematic of this problem is shown in Figure 41. On the right end of the domain we have a type 3 boundary condition representing convective heat transfer to an environment maintained at a temperature of 0° .

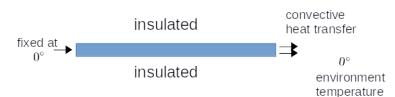


Figure 41: Schematic of Example #1.

Note that we have not established any physical units on this system. If it helps, consider all temperatures given to be in degrees Celsius.

We will solve this boundary value problem using separation of variables. Since we have already carried out the separation process for the heat equation numerous times, we will skip ahead and write down the separated equations:

$$\begin{aligned} u(x,t) &= F(x)G(t) \\ F_{xx} + \lambda F &= 0 \\ G_t + \alpha^2 \lambda G &= 0 \end{aligned}$$

We need to find values of λ for which non-trivial solutions to the equations can satisfy the boundary conditions. Since $F(x)$ is the spatial variable to which the boundary conditions apply, and both boundary conditions are homogeneous, we will focus our attention on the equation for $F(x)$.

$\lambda = 0$:

$$\begin{aligned} F_{xx} &= 0 \\ F(x) &= c_1 x + c_2 \\ F(0) = c_1(0) + c_2 &= 0 \\ \Rightarrow c_2 &= 0 \\ F_x(1) = c_1 &= -hc_1(1) \\ \Rightarrow c_1 &= 0 \end{aligned}$$

So we will rule out $\lambda = 0$ since only the trivial solution $F(x) = 0$ applies.

$\lambda < 0$: $\lambda = -\nu^2$, $\nu > 0$.

$$\begin{aligned} F_{xx} - \nu^2 F &= 0 \\ F(x) &= c_1 \cosh \nu x + c_2 \sinh \nu x \\ F(0) = c_1(1) + c_2(0) &= 0 \\ \Rightarrow c_1 &= 0 \\ F_x(1) = \nu c_2 \cosh \nu &= -hc_2 \sinh \nu \\ \frac{\nu}{h} c_2 &= -c_2 \tanh \nu \\ \Rightarrow c_2 &= 0 \end{aligned}$$

Since $\nu > 0$ and $h > 0$, and, as shown by Figure 42 $\tanh \nu$ is also positive, c_2 must also be equal to zero.¹

Since $h > 0$, the only way that $c_1 = -hc_1$ is if $c_1 = 0$. If c_1 were not zero, the last line would imply that $h = -1$ which, since $h > 0$, is a contradiction.

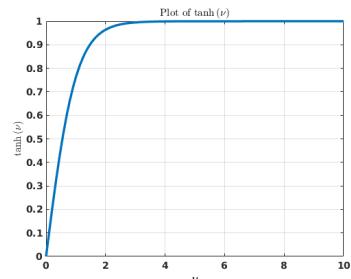


Figure 42: Plot of $\tanh(\nu)$.

¹ By this time, the reader should be getting pretty good at making these kinds of observations.

$\lambda > 0$: $\lambda = \nu^2$, $\nu > 0$.

$$\begin{aligned} F_{xx} + \nu^2 F &= 0 \\ F(x) &= c_1 \cos \nu x + c_2 \sin \nu x \\ F(0) &= c_1(1) + c_2(0) = 0 \\ \Rightarrow c_1 &= 0 \\ F_x(1) &= \nu c_2 \cos \nu = -h c_2 \sin \nu \\ c_2 \tan \nu &= -\frac{\nu}{h} \end{aligned}$$

Of course $c_2 = 0$ would satisfy this condition but we are looking for non-trivial solutions. We see from Figure 43 that the plot of $\tan \nu$ intersects the plot of $-\nu/h$ infinitely many times.² The values of ν where this happens will be used to find our eigenvalues, ν_n^2 , and the eigenfunctions will be $F_n(x) = c_n \sin \nu_n x$. Applying these eigenvalues to our separated equation $G(t)$ gives us:

$$\begin{aligned} G_t + \alpha^2 \nu^2 G &= 0 \\ G_n(t) &= c_3 e^{-(\alpha \nu_n)^2 t} \end{aligned}$$

and the product solution is:

$$u(x, t) = \sum_{n=1}^{\infty} c_n \sin(\nu_n x) e^{-(\alpha \nu_n)^2 t}$$

Now we must apply the initial condition:

$$u(x, 0) = \sum_{n=1}^{\infty} c_n \sin(\nu_n x) \overset{1}{\cancel{e^0}} = f(x)$$

On the left side of the equation above we have an infinite series with unknown coefficients c_n ; on the right side we have a given function $f(x)$. Our job is to find the values c_n so that the two sides are, in fact, equal. To do this we will multiply both sides of the equation by our orthogonal functions— $\sin(\nu_n x)$ —and integrate over the interval $x \in [0, 1]$.³ At this point we still have not nailed down the numeric values of ν_n but we *do* know that, since the separated boundary value problem for $F(x)$ from which we obtained ν_n and $\sin(\nu_n x)$ fall within the realm of Sturm-Liouville eigenvalue theory, the eigenfunctions are mutually orthogonal with respect to the weight function, $p(x)$ which, in this case, can be shown to be $p(x) = 1$.

The coefficients are given by:

$$c_n = \frac{(f(x), \sin(\nu_n x))}{(\sin(\nu_n x), \sin(\nu_n x))} = \frac{\int_0^1 f(x) \sin(\nu x) dx}{\int_0^2 \sin(\nu x)^2 dx}$$

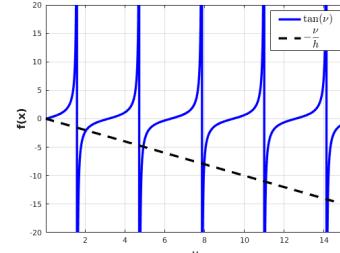


Figure 43: Plot of $\tan \nu$ and $-\nu/h$ vs ν for $h = 1$.

² This is the sort of claim for which a mathematician might reasonably expect a proof but, as engineers, we will simply not provide one. We know $\tan \nu$ is periodic and the pattern will continue indefinitely and the straight line $-\nu/h$ will intersect each period no matter the value of h and that will be good enough for us.

³ We should hasten to add—"and weight function"—to this phrase. It is an important detail that is easy to forget. When we move on to problems in polar and cylindrical coordinates we will need to be more strict about this.

WE WILL USE MATLAB to construct our solution to this boundary value problem. As usual, we will start by clearing out the MATLAB workspace, specify our constants and initial conditions and determine the number of terms to use in our infinite series for $u(x, t)$.

```

clear
clc
close 'all'

%% Parameters
alpha_sq = 0.1; % thermal diffusivity
h = 10; % convective heat transfer coefficient
L = 1;
f = @(x) 1; % initial condition

N = 25; % number of eigenfunctions

```

1
2
3
4
5
6
7
8
9
10
11

Next we must find numeric values for ν_n . To do this, we will observe from Figure 41 that there is exactly one intersection of $\tan \nu$ and $-\nu/h$ in the interval $\nu \in [\pi/2, 3\pi/2]$ and another intersection in the interval $\nu \in [3\pi/2, 5\pi/2]$ etcetera. We keep looking in intervals $[n\pi/2, (n+2)\pi/2]$, $n = 1, 3, 5, \dots$. This is done using the MATLAB built-in function `fzero()`.

```

%% Find the desired number of eigenvalues
ev_fun = @(x) tan(x) + x./h; ①

nu = nan(N,1);
delta = 1e-8; ②
xo = [pi/2+delta ,3*pi/2-delta];
for i = 1:N
    [x,fval,exit_flag] = fzero(ev_fun,xo); ③
    nu(i) = x;
    xo = xo + pi;
end
% do some crude error checking
assert(min(diff(nu))>0,...)
'Error! Something is wrong with your eigenvalues!';④

```

14
15
16
17
18
19
20
21
22
23
24
25
26
27

① `fzero()` is a root-finding function so we need to re-formulate $\tan \nu = -\nu/h$ so a root-finder can get the values of ν .

② We do not want to evaluate our function at the ends of the search interval since those correspond to asymptotes of $\tan x$. We move our search boundaries “in” a little bit (towards the interior of the interval) to avoid those asymptotes.

③ In this instance `fzero()` has two arguments and returns three values. The variable `ev_fun` is a handle⁴ to the function to which we are looking for roots and `xo` is the interval where we are searching for the roots. Of the return arguments: `x` is the root; `fval` is the numeric value of `ev_fun` evaluated at the root `x`; and `exit_flag` is a value returned by `fzero()` to indicate if the function was successful or not.⁵

④ This is a modest bit of error checking to ensure that, at a minimum, the roots we find are distinct.

The remainder of the MATLAB script is typical of what we have been doing so far.

```

%% Construct solution
Fn = @(x,n) sin(nu(n).*x);
Gn = @(t,n) exp(-(nu(n).^2).*alpha_sq.*t);

```

⁴ A variable that, when evaluated, returns a function is often referred to as a *handle* to the function. This is similar to the variable `gca` which can be thought of as a handle to the current axis.

⁵ See the documentation for a description of the possible values for `exit_flag`. Several built-in MATLAB functions use this output variable and a value of `exit_flag=1` normally indicates success. As you can see, we do not check the value of `exit_flag` in this script (and we also ignore `fval`) but readers are encouraged to make use of this kind of feedback from MATLAB functions when the reliability of their results is important.

```

u = @(x, t) o;
for n = 1:N
    ef_mag = integral(@(x) Fn(x, n) .* Fn(x, n), o, L); ⑤
    cn = integral(@(x) f(x) .* Fn(x, n), o, L) ./ ef_mag;
    u = @(x, t) u(x, t) + cn*Fn(x, n).*Gn(t, n);
end

```

33
34
35
36
37
38
39
40

⑤ We have not bothered to try and derive a closed-form solution for $\int_0^1 \sin(\nu_n x) dx$ so we do it numerically.

The solution for the parameters we have selected, plotted at various times, is shown in Figure 44.

Some questions that you should make sure that you can answer:

1. What is the final steady-state temperature? Answer: zero since all of the energy from the initial condition is transferred out via the left or right boundaries.
2. Why is the initial condition so “wiggly”? Answer: because the initial condition does not match the boundary condition at $x = 0$ and the discontinuity causes perturbations typical of Fourier series.
3. What happens if α^2 is increased or decreased?
4. What happens if h is increased or decreased?

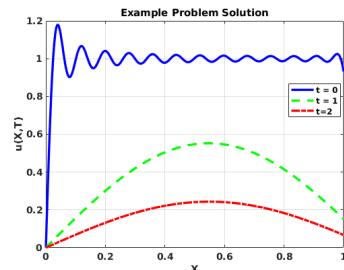


Figure 44: Solution for Example #1.

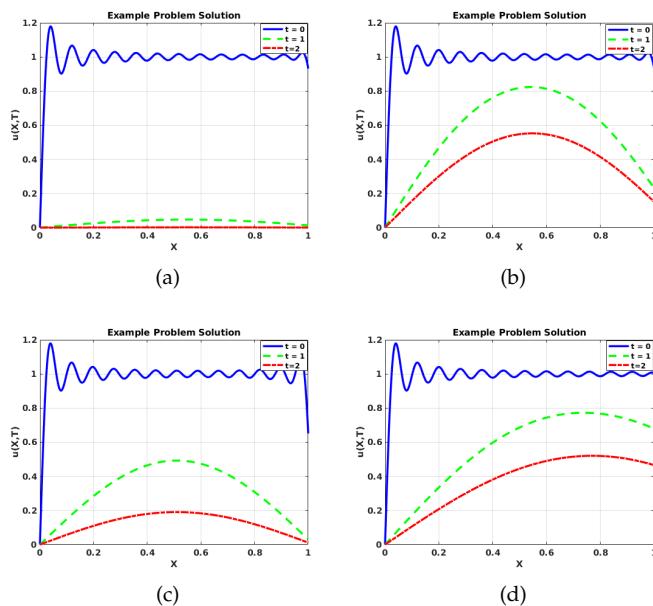


Figure 45: Variations in α and h for example problem.

Answers to the last two questions are illustrated in Figure 45. Images a) and b) illustrate how the solution changes when thermal diffusivity is increased or decreased, respectively. Images c) and d) illustrate how the solution changes when the convective heat transfer coefficient, h , is increased and decreased.

Example #2: The twist angle $\theta(x, t)$ of a torsionally vibrating shaft is given by the wave equation

$$\text{Governing Equation : } \alpha^2 \frac{\partial^2 \theta}{\partial x^2} = \frac{\partial^2 \theta}{\partial t^2}, \quad \alpha > 0, \quad 0 < x < 1, \quad t > 0$$

$$\text{Boundary Conditions : } \theta(0, t) = 0, \quad \theta_x(1, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } \theta(x, 0) = x, \quad \theta_t(x, 0) = 0, \quad 0 < x < 1$$

OF COURSE WE will also use separation of variables for this boundary value problem and, again, since we have already carried out the separation process for the wave equation numerous times, we will again skip ahead and write down the separated equations:

$$\begin{aligned} \theta(x, t) &= F(x)G(t) \\ F_{xx} + \lambda F &= 0 \\ G_{tt} + \alpha^2 \lambda G &= 0 \end{aligned}$$

We need to find values of λ for which non-trivial solutions to the equations can satisfy the boundary conditions. As with the first example, $F(x)$ is the spatial variable to which the boundary conditions apply and both boundary conditions are homogeneous. Thus we will again focus our attention on the equation for $F(x)$.

$\lambda = 0$:

$$\begin{aligned} F_{xx} &= 0 \\ F(x) &= c_1 x + c_2 \\ F(0) &= c_1(0) + c_2 = 0 \\ \Rightarrow c_2 &= 0 \\ F_x(0) &= c_1 = 0 \\ \Rightarrow c_1 &= 0 \end{aligned}$$

We see that for $\lambda = 0$ only the trivial solution can satisfy the boundary conditions.

$\lambda < 0$: $\lambda = \nu^2, \nu > 0$.

$$\begin{aligned} F_{xx} - \nu^2 F &= 0 \\ F(x) &= c_1 \cosh \nu x + c_2 \sinh \nu x \\ F(0) &= c_1(1) + c_2(0) = 0 \\ \Rightarrow c_1 &= 0 \\ F_x(1) &= \nu c_2 \cosh \nu = 0 \end{aligned}$$

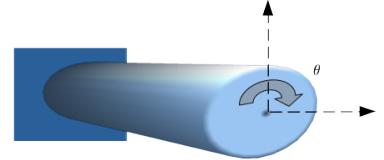


Figure 46: Schematic of Example #2.

For the last conditions, since $\cosh \nu$ is always positive for $\nu > 0$, $c_2 = 0$ and, again, only the trivial solution can satisfy the boundary conditions.

$$\lambda > 0: \lambda = \nu^2, \nu > 0.$$

$$\begin{aligned} F_{xx} + \nu^2 F &= 0 \\ F(x) &= c_1 \cos \nu x + c_2 \sin \nu x \\ F(0) &= c_1(1) + c_2(0) = 0 \\ \Rightarrow c_1 &= 0 \\ F_x(1) &= \nu c_2 \cos \nu = 0 \end{aligned}$$

By this point in time, this analysis may begin to feel quite repetitive. That is because it *is* repetitive. Nonetheless, resist the tendency to rush and/or become complacent. It is worth your while to patiently and carefully work through each of these cases every time.

We can satisfy this boundary condition for any value of c_2 provided that ν is a root of $\cos(\nu)$. This happens when ν is an odd-integer multiple of $\pi/2$. Thus:

$$\begin{aligned} \nu_n &= (2n-1)\frac{\pi}{2}, n = 1, 2, 3 \dots \\ F_n(x) &= c_n \sin \nu_n x \end{aligned}$$

The corresponding solution to $G(t)$ is:

$$\begin{aligned} G_{tt} + \alpha^2 \nu_n^2 G &= 0 \\ G(t) &= a_n \cos(\alpha \nu_n t) + b_n \sin(\alpha \nu_n t) \end{aligned}$$

and the product solution is:

$$u(x, t) = F(x)G(t) = \sum_{n=1}^{\infty} [a_n \cos(\alpha \nu_n t) + b_n \sin(\alpha \nu_n t)] \sin(\nu_n x)$$

To **SOLVE FOR** the remaining constants a_n and b_n we need to apply the initial conditions.

$$\begin{aligned} u(x, 0) &= \sum_{n=1}^{\infty} [a_n(1) + b_n(0)] \sin(\nu_n x) \\ &= \sum_{n=1}^{\infty} a_n \sin(\nu_n x) = x \end{aligned}$$

To solve for a_n , we now only need to multiply both sides by our orthogonal function— $\sin(\nu_n x)$ —and integrate. The resulting formula for a_n is:

$$a_n = \frac{(x, \sin(\nu_n x))}{(\sin(\nu_n x), \sin(\nu_n x))} = \underbrace{\frac{\int_0^1 x \sin(\nu_n x) dx}{\int_0^1 \sin(\nu_n x)^2 dx}}_{1/2}$$

This really *looks* like a Fourier series, but technically it is not since the expansion functions are not $\sin(n\pi x/L)$ or $\cos(n\pi x/L)$.

The other boundary condition is applied to get b_n :

$$u_t(x, 0) = \sum_{n=1}^{\infty} [-\alpha \nu_n a_n(0) + \alpha \nu_n b_n(1)] \sin(\nu_n x) = 0$$

$$\Rightarrow b_n = 0, \quad n = 1, 2, 3 \dots$$

In summary, the solution is:

$$u(x, t) = \sum_{n=1}^{\infty} a_n \cos(\alpha \nu_n t) \sin(\nu_n x)$$

$$a_n = 2 \int_0^1 x \sin(\nu_n x) dx$$

THE MATLAB CODE to construct and visualize the solution for this problem is shown in the listing below.

```

1 clear
2 clc
3 close 'all'

4 %% Parameters
5 alpha_sq = 5;
6 L = 1;
7 N = 30;

8 nu = @(n) (2*n-1)*pi/2;
9 Fn = @(x,n) sin(nu(n).*x);
10 Gn = @(t,n) cos(alpha_sq*nu(n).*t);
11 f = @(x) x; % initial displacement
12 u = @(x,t) 0; % initial velocity

13 for n = 1:N
14     ef_mag = integral(@(x) Fn(x,n).^2,0,L);
15     an = integral(@(x) f(x).*Fn(x,n),0,L)./ef_mag;
16     u = @(x,t) u(x,t) + an.*Fn(x,n).*Gn(t,n);
17 end

18 %% Plot the solution
19 Nx = 1000;
20 X = linspace(0,L,Nx);

21 Tmax = 5;
22 Nt = 50;
23 T = linspace(0,Tmax,Nt);

24 figure(2)
25 for t = 1:Nt
26     plot(X,u(X,T(t)),'-b','linewidth',3);
27     title_str = sprintf('Lecture 28 Example 2, t = %g ',T(t));
28     title(title_str,'fontsize',16,'fontweight','bold');
29     xlabel('X','fontsize',14,'fontweight','bold');
30     ylabel('U(X,T) Angular Displacement','fontsize',14,...);
31         'fontweight','bold');
32     grid on
33     set(gca,'fontsize',12,'fontweight','bold');
34     axis([0 L -L L]);
35     pause(Tmax/(Nt-1));
36 end
37
38
39
40
41
42

```

Assignment #10

1. Solve the boundary value problem below using separation of variables.

Governing Equation : $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad 0 < x < a, \quad 0 < y < b$

Boundary Conditions : $u(x,0)=0 \quad u(0,y)=0$
 $u(x,b)=f(x) \quad u(a,y)=0$

2. Solve the boundary value problem below using separation of variables.

Governing Equation : $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad 0 < x < 1, \quad 0 < y < 1$

Boundary Conditions : $u_y(x,0)=0 \quad u(0,y)=0$
 $u_y(x,1)=0 \quad u(1,y)=1-y$

Symbolically solve the problem and set up, but do not evaluate analytically, the Fourier coefficients.
Use MATLAB to numerically find the solution using $N = 15$ eigenmodes. Plot the solution using the MATLAB built-in function `surf()` and print out the value of the solution at $u(0.5, 0.5)$.

3. Solve the non-homogeneous boundary value problem below.

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + Ae^{-\beta x}, \quad \alpha > 0, \beta > 0, \quad 0 < x < 1, \quad t > 0$

Boundary Conditions : $u(0,t) = 0, \quad u(1,t) = 0, \quad t > 0$

Initial Conditions : $u(x,0) = f(x), \quad 0 < x < L$

4. Find a steady-state solution $\psi(x)$ of the non-homogeneous boundary value problem below.

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} - h(u - u_0), \quad \alpha > 0, h > 0, \quad 0 < x < 1, \quad t > 0$

Boundary Conditions : $u(0, t) = u_0, \quad u(1, t) = 0, \quad t > 0$

Initial Conditions : $u(x, 0) = f(x), \quad 0 < x < L$

5. Solve the boundary value problem below using separation of variables.

Governing Equation : $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad 0 < x < 1, \quad 0 < y < 1$

Boundary Conditions : $u(x, 0) = 0, \quad u_x(0, y) = 0$
 $u_y(x, 1) = 0, \quad u(1, y) = u_0$

Solve the equation symbolically. Use MATLAB to represent the solution for $N = 15$ eigenmodes with $u_0 = 0.25$. Plot the solution using MATLAB's built-in function `surf()`.

Lecture 29 - Fourier Series in Two Variables

Objectives

- Present Fourier series expansions in two (or more) independent variables.
- Show an example application in the solution to the heat equation in two spatial dimensions.

Fourier Series Expansion in Two Variables

Consider the function $f(x, y)$, $0 < x < a$, $0 < y < b$. Suppose we want to represent the function in the form of a *double* Fourier series as follows:

$$f(x, y) = \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} A_{mn} \sin \frac{m\pi y}{a} \sin \frac{n\pi x}{b} \quad (95)$$

1. Should we expect this to be possible? **Answer:** Of course! Fourier series have shown themselves to be a perfectly adequate tool for representing a variety of functions. There is no problem in adding another spatial dimension.
2. How will we determine the appropriate values for A_{mn} ? **Answer:** We will multiply both sides by orthogonal functions and integrate, as usual.

$$\int_0^b \int_0^a f(x, y) \sin \frac{m' \pi x}{a} \sin \frac{n' \pi y}{b} dx dy = \dots$$

$$\int_0^a \int_0^b A_{mn} \sin \frac{m \pi x}{a} \sin \frac{n \pi y}{b} \sin \frac{m' \pi x}{a} \sin \frac{n' \pi y}{b} dx dy$$

where, if $m' = m$ and $n' = n$ then:

$$\int_0^b \int_0^a f(x, y) \sin \frac{m' \pi x}{a} \sin \frac{n' \pi y}{b} dx dy = \dots$$

$$A_{mn} \underbrace{\left[\int_0^a \sin \left(\frac{m \pi x}{a} \right)^2 dx \right]}_{=\frac{a}{2}} \underbrace{\left[\int_0^b \sin \left(\frac{n \pi y}{b} \right)^2 dy \right]}_{=\frac{b}{2}}$$

Again, these sets of functions— $\sin \frac{m \pi x}{a}$ and $\sin \frac{n \pi y}{b}$ —are orthogonal with respect to a weight function $p(x) = 1$ and we should not forget it even if we leave it out of the equations and fail to mention it in the discussion.

Of course, if $m' \neq m$ or $n' \neq n$ then the respective integral is zero by orthogonality.

So that

$$A_{mn} = \frac{4}{ab} \int_0^b \int_0^a f(x,y) \sin \frac{m\pi x}{a} \sin \frac{n\pi y}{b} dx dy \quad (96)$$

MATLAB Implementation

Implementing this expansion in MATLAB is mostly straight-forward.
We will start by defining parameters:

```

clear
clc
close 'all'

%% Parameters
a = 4;
b = 3;

N = 20;

ex_select = 1;
% 1 smooth
% 2 not smooth

switch ex_select %❶
    case 1
        f = @(x,y) x.* (a-x).*y.* (b-y);

    case 2
        f = @(x,y) ex1(x,y,a,b);

    otherwise
        error('Invalid Example Choice!');
end

```

❶ Here we use a `switch ... case` selection structure. For the in-class demonstration I like to choose between different functions for which the double Fourier expansion will be used. The parameter `ex_select` is set to either 1 or 2; the `case` statements allow me to assign a function handle to `f` accordingly. The `otherwise` statement allows me to gracefully deal with having specified an invalid value for `ex_select`.

Next we will visualize the function that we hope to represent with a double Fourier series.

```

Nx = 250;
Ny = 250;

X = linspace(0,a,Nx);
Y = linspace(0,b,Ny);

[XX,YY] = meshgrid(X,Y);

figure(1)
surf(XX,YY,f(XX,YY),'edgecolor','none');
title('f(x,y)', 'fontsize',16,'fontweight','bold');
xlabel('X', 'fontsize',14,'fontweight','bold');
ylabel('Y', 'fontsize',14,'fontweight','bold');
zlabel('f(X,Y)', 'fontsize',14,'fontweight','bold');
set(gca, 'fontsize',12,'fontweight','bold');

```

Plots of the two example functions along with their double Fourier series expansions are shown in Figure 47.

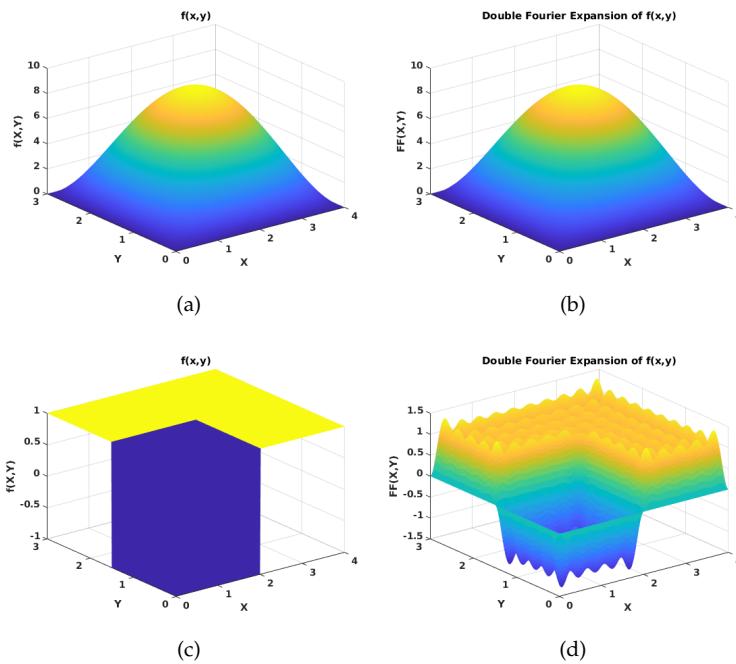


Figure 47: Plots of example functions and their double Fourier series expansions.

Note the “wigginess” present in the double Fourier expansion in the presence of discontinuities.

The MATLAB code needed to calculate the Fourier coefficients and plot the resulting expansion are provided below:

```
%>> %% Double Fourier Expansion
42 Xn = @(x,n) sin(n.* pi.* x./a);
43 Yn = @(y,n) sin(n.* pi.* y./b);
44
45 A = nan(N,N); %②
46 FF = @(x,y) 0;
47
48 for m = 1:N
49     for n = 1:N
50         A(m,n) = integral2(@(x,y) f(x,y).*Xn(x,m).*Yn(y,n), ...
51             0,a,0,b)./...
52             integral2(@(x,y) (Xn(x,m).^2).* (Yn(y,n).^2), ...
53             0,a,0,b); %(same as formula - 4/(a*b) above)
54         FF = @(x,y) FF(x,y)+A(m,n).*Xn(x,m).*Yn(y,n);
55     end
56 end
57
58 figure(2)
59 surf(XX,YY,FF(XX,YY), 'edgecolor', 'none');
60 title('Double Fourier Expansion of f(x,y)', 'fontsize',16, 'fontweight', 'bold');
61 xlabel('X', 'fontsize',14, 'fontweight', 'bold');
62 ylabel('Y', 'fontsize',14, 'fontweight', 'bold');
63 zlabel('FF(X,Y)', 'fontsize',14, 'fontweight', 'bold');
64 set(gca, 'fontsize',12, 'fontweight', 'bold');
65
```

Lastly we need to define the local function for $\text{ex1}(x)$. This example is presented to illustrate that the convergence behavior for a double Fourier series is similar to a regular (single) Fourier series. This

② The Fourier coefficients are conveniently stored in a two-dimensional matrix.

③ We use a double-nested loop to do the calculations. Note in this implementation we do not make use of the known value of the magnitude of the eigenfunctions.

function is piece-wise continuous and is defined as:

$$f(x) = \begin{cases} -1, & x < \frac{a}{2} \text{ and } y < \frac{b}{2} \\ 0, & \text{otherwise} \end{cases}$$

```
%// Local functions
function z = ex1(x,y,a,b)
[mx,nx] = size(x); % write your function to expect vector inputs
[my,ny] = size(y);
% for this implementation, I will expect x and y to have the
% same size
assert((mx==my) && (nx == ny), ...
    'error: x and y must have same size');
z = nan(mx,nx); % construct z to be the same as X and Y

% there are fancier ways to do this, but we will use a very
% simple
% implementation
for i = 1:mx
    for j = 1:nx
        if (x(i,j) < a/2) && (y(i,j) < b/2)
            z(i,j) = -1;
        else
            z(i,j) = 1;
        end
    end
end
end
```

66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86

④ Note the use of an assertion to enforce my expectation that the same number of Fourier modes will be used for the expansion in x as is used in the expansion in y .

Application to Solving a BVP

In this section we will carry out an example problem that calls for use of a double Fourier series expansion. Consider the boundary value problem below based on the heat equation.

Governing Equation : $\frac{\partial u}{\partial t} = \alpha^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \quad \alpha > 0, \quad 0 < x < a, \quad 0 < y < b, \quad t > 0$

Boundary Conditions : $u(x,0,t)=0 \quad u(0,y,t)=0 \quad t > 0$
 $u(x,b,t)=0 \quad u(a,y,t)=0$

Initial Conditions : $u(x,y,0) = f(x,y), \quad 0 < x < a, \quad 0 < y < b$

We will solve this problem using separation of variables.

Step #1: Assume a product solution.

$$u(x,y,t) = F(x)G(y)H(t)$$

Note: Think about the physics that this boundary value problem represents. It is the time-dependent heat equation on a rectangular domain. An initial temperature distribution is specified but all of the boundaries are held at a temperature of 0. Over time, we expect *all* of the energy to exit the domain via the boundaries so the final steady-state temperature is zero everywhere.

Step #2: Insert proposed solution into the governing equation.

$$\frac{\partial}{\partial t} [F(x)G(y)H(t)] = \alpha^2 \left\{ \frac{\partial^2}{\partial x^2} [F(x)G(y)H(t)] + \frac{\partial^2}{\partial y^2} [F(x)G(y)H(t)] \right\}$$

$$FGH_t = \alpha^2 (F_{xx}GH + FG_{yy}H)$$

Step #3a: Separate variables (partially: x from y and t).

$$\frac{FGH_t}{\alpha^2 FGH} = \frac{\alpha^2 F_{xx}GH}{\alpha^2 FGH} + \frac{\alpha^2 FG_{yy}H}{\alpha^2 FGH}$$

$$\frac{H_t}{\alpha^2 H} = \frac{F_{xx}}{F} + \frac{G_{yy}}{G}$$

$$\underbrace{\frac{F_{xx}}{F}}_{\text{function of } x} = \underbrace{-\frac{G_{yy}}{G} + \frac{H_t}{\alpha^2 H}}_{\text{function of } y,t} = -\lambda$$

$$F_{xx} + \lambda F = 0$$

$$\frac{G_{yy}}{G} = \frac{H_t}{\alpha^2 H} + \lambda$$

In this line, the function of x will, in general, only be equal to a function of y and t if they are both equal to some constant.

Step #3b: Separate remaining variables— t from y .

$$\frac{G_{yy}}{G} = \frac{H_t}{\alpha^2 H} + \lambda = -\mu$$

$$G_{yy} + \mu G = 0$$

$$H_t + \alpha^2 \gamma H = 0, \text{ where: } \gamma = \lambda + \mu$$

Step #4: Apply boundary conditions to determine non-trivial product solution(s). We have solved problems with these boundary conditions many times so, in the interest of brevity, we will simply state the following:

$$\lambda > 0, \quad \lambda = \beta_m^2, \quad \beta_m = \frac{m\pi}{a}$$

$$\mu > 0, \quad \mu = \beta_n^2, \quad \beta_n = \frac{n\pi}{b}$$

$$F_m(x) = \sin \frac{m\pi x}{a}$$

$$G_n(y) = \sin \frac{n\pi y}{b}$$

$$H_{mn}(t) = e^{-\alpha^2 \left[\left(\frac{m\pi}{a} \right)^2 + \left(\frac{n\pi}{b} \right)^2 \right] t}$$

Please do not let me tempt you off the virtuous path of analyzing these conditions carefully. You will find the analysis to be similar to what you have done before.

The product solution is given in Equation 97.

$$u(x, y, t) = \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right) e^{-\alpha^2 \left[\left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2\right]t} \quad (97)$$

Step #5: Satisfy the initial condition.

$$\begin{aligned} u(x, y, 0) &= \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right) e^{0 \cdot 1} = f(x, y) \\ &= \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right) = f(x, y) \end{aligned}$$

This is the double Fourier series that we started the lecture with. The coefficients are given by Equation 98.

$$A_{mn} = \frac{4}{ab} \int_0^b \int_0^a f(x, y) \sin \frac{m\pi x}{a} \sin \frac{n\pi y}{b} dx dy \quad (98)$$

A problem requiring use of the double Fourier series will be included in the homework.

Part V

Boundary Value Problems in Polar, Cylindrical and Spherical Coordinates

Lecture 30 - Laplacian in Polar Coordinates

Objectives

- Discuss the derivation of the Laplacian operator in polar coordinates.
- Do an example in which we solve Laplace's equation (steady-state heat equation) in polar coordinates.

Laplacian Operator in Polar Coordinates

In past lectures we have used the Laplacian operator (∇^2 or Δ) which is shown for rectangular coordinates below.

$$\begin{aligned}\nabla &= \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle \\ \nabla \cdot \nabla &= \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \\ &= \Delta\end{aligned}$$

The operator notation is convenient insofar as it is agnostic regarding coordinate system. We can use the Laplacian in a differential equation—e.g. $\nabla^2 u = 0$ —without immediate regard as to whether the domain will be described in a rectangular (Cartesian), polar, cylindrical or spherical coordinate system. Still, at some point in time before we can hope to solve such a differential equation, we need to select a coordinate system. Once we have done that, we need to have an expression of the Laplacian available with the appropriate independent variables.

WE WILL START with polar coordinates. The standard schematic of the cylindrical coordinate system is shown in Figure 48; polar coordinates are just cylindrical coordinates without the z -dimension. To derive an expression for the Laplacian operator in polar coordinates, we need to be able to express the independent variables for polar

Question: How do you pick a coordinate system? **Answer:** You select a coordinate system that makes *the boundary of the domain easy to describe*.

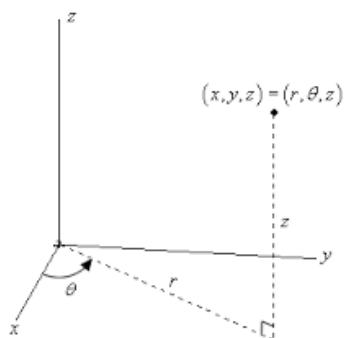


Figure 48: Cylindrical coordinate system.

coordinates— r and θ —in terms of the independent variables for rectangular coordinates— x and y . The relations between the respective coordinates are given in the margin.

Therefore, to express the Laplacian in polar coordinates, we need to express:

$$u_{xx} + u_{yy} = 0$$

in terms of r and θ . We have done changes of variables like this before; we will follow the same process.

Starting with the first derivatives, using the chain rule and product rule we have:

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial r} \frac{\partial r}{\partial x} + \frac{\partial u}{\partial \theta} \frac{\partial \theta}{\partial x}$$

or, using subscript notation:

$$u_x = u_r r_x + u_\theta \theta_x$$

We need to replace every occurrence of x with its equivalent in terms of r and θ .

$$\begin{aligned} r &= (x^2 + y^2)^{1/2} \\ r_x &= 2x \frac{(x^2 + y^2)^{-1/2}}{2} \\ &= \frac{2(r \cos \theta)}{2r} \\ &= \cos \theta \end{aligned}$$

$$\begin{array}{ll} r^2 = x^2 + y^2 & x = r \cos \theta \\ \theta = \tan^{-1}(y/x) & \text{or} \\ & y = r \sin \theta \end{array}$$

Here we use:

$$x = r \cos \theta$$

and

$$(x^2 + y^2)^{-1/2} = 1/r$$

$$\begin{aligned} \theta &= \tan^{-1}(y/x) \\ \theta_x &= -\frac{y}{x^2} \left(\frac{1}{1 + y^2/x^2} \right) \\ &= -\frac{y}{x^2 + y^2} \\ &= \frac{-r \sin \theta}{r^2} \\ &= -\frac{\sin \theta}{r} \end{aligned}$$

Here we use the product rule and the not-so-familiar fact that:

$$\frac{d}{dx} \tan^{-1}(u) = \frac{1}{1+u^2}$$

$$y = r \sin \theta$$

$$x^2 + y^2 = r^2$$

So:

$$\begin{aligned} u_x &= u_r r_x + u_\theta \theta_x \\ u_x &= u_r \cos \theta - u_\theta \frac{\sin \theta}{r} \end{aligned}$$

Repeating the process to find the equivalent of u_{xx} :

$$\begin{aligned} u_{xx} &= (u_x)_x \\ &= (u_x)_r r_x + (u_x)_\theta \theta_x \\ &= \left(u_r \cos \theta - u_\theta \frac{\sin \theta}{r} \right)_r \cos \theta - \left(u_r \cos \theta - u_\theta \frac{\sin \theta}{r} \right)_\theta \frac{\sin \theta}{r} \end{aligned}$$

We would then start all over again to find u_y and u_{yy} . Balancing mathematical tedium with conceptual understanding, we will omit these details. Interested readers are encouraged to finish the job.¹

AFTER MUCH TEDIOUS work and simplification you can arrive at the expression of the Laplacian in polar coordinates given in Equation 99.

$$\nabla^2 u = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} \quad (99)$$

We are now ready to solve Laplace's equation in polar coordinates.

Example: Solve the boundary value problem below based on the steady-state heat equation on a circular plate of radius c .

Governing Equation : $\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0, \quad 0 < r < c, \quad 0 < \theta < 2\pi$

Boundary Conditions : $u(c, \theta) = f(\theta), \quad 0 < \theta < 2\pi$

You can confirm that the equation is linear and homogeneous. As usual, we will use separation of variables to solve the problem.

Step #1: Assume a product solution.

$$u = F(r)G(\theta)$$

Step #2: Insert the product solution into the governing equation.

$$\begin{aligned} \frac{\partial^2}{\partial r^2} [F(r)G(\theta)] + \frac{1}{r} \frac{\partial}{\partial r} [F(r)G(\theta)] + \frac{1}{r^2} \frac{\partial^2}{\partial \theta^2} [F(r)G(\theta)] &= 0 \\ F_{rr}G + \frac{1}{r}F_rG + \frac{1}{r^2}FG_{\theta\theta} &= 0 \end{aligned}$$

Step #3: Separate variables.

$$\begin{aligned} \frac{F_{rr}G}{FG} + \frac{1}{r} \frac{F_rG}{FG} + \frac{1}{r^2} \frac{FG_{\theta\theta}}{FG} &= 0 \\ \frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} + \frac{1}{r^2} \frac{G_{\theta\theta}}{G} &= 0 \\ \underbrace{\frac{r^2 F_{rr} + r F_r}{F}}_{\text{function of } r} &= -\underbrace{\frac{G_{\theta\theta}}{G}}_{\text{function of } \theta} = \lambda \\ r^2 F_{rr} + r F_r - \lambda F &= 0 \\ G_{\theta\theta} + \lambda G &= 0 \end{aligned}$$

¹ Unlike most other hazing rituals, deriving the Laplacian operator for polar coordinates has some (small) redeeming benefits.

Step #4: Apply boundary conditions to determine non-trivial product solution(s). There is only one boundary condition explicitly given in this problem and it applies to the spatial variable r . In this problem, however, there are some important implicit boundary conditions:

1. The solution must be periodic in θ . This will be needed so that all the eigenfunctions in θ can be used; and
2. The solution must be finite everywhere. In particular, it will be important that $\lim_{r \rightarrow 0} F(r) < \infty$.

$\lambda = 0$:

We will start with $G(\theta)$:

$$\begin{aligned} G_{\theta\theta} &= 0 \\ G(\theta) &= c_1 + c_2\theta \end{aligned}$$

Since the solution must be periodic, with period 2π , $c_2 = 0$. Consequently $G(\theta) = c_1$ is a non-trivial eigenfunction.

Checking solutions for $F(r)$:

$$\begin{aligned} r^2 F_{rr} + rF_r &= 0 \\ F(r) &= c_3 + c_4 \ln r \end{aligned}$$

In order to satisfy our other implicit boundary condition, $c_4 = 0$. So $F(r) = c_3$ is also a valid eigenfunction. Looking ahead, our product solution will include a constant term for the eigenvalue $\lambda = 0$.

$\lambda < 0$: where we set $\lambda = -\nu^2$, $\nu > 0$.

For $G(\theta)$:

$$\begin{aligned} G_{\theta\theta} - \nu^2 G &= 0 \\ G(\theta) &= c_1 \cosh \nu\theta + c_2 \sinh \nu\theta \end{aligned}$$

Neither $\cosh()$ nor $\sinh()$ are periodic so we must conclude that $c_1 = c_2 = 0$ and that there are no non-trivial eigenfunctions for $\lambda < 0$.

$\lambda > 0$: where we set $\lambda = \nu^2$, $\nu > 0$.

For $G(\theta)$:

$$\begin{aligned} G_{\theta\theta} + \nu^2 G &= 0 \\ G(\theta) &= c_1 \cos \nu\theta + c_2 \sin \nu\theta \end{aligned}$$

which is periodic, with period 2π , if ν is an integer n .

For $F(r)$:

$$r^2 F_{rr} + rF_r - n^2 F = 0$$

Reminder: This is a Cauchy-Euler equation. Recall from the beginning of the course that we seek solutions of the form r^m . Inserting r^m into the equation gives us: $r^2[m(m-1)r^{m-2} + mr^m] = 0$ which can be simplified to: $r^m[m(m-1) + m] = 0$ which is only true if $m^2 = 0$. Recall how to deal with this double-root by applying a factor of $\ln r$ to the first solution.

This is a Cauchy-Euler equation. Inserting $F = r^m$ into the equation gives us:

$$\begin{aligned} r^2[m(m-1)]r^{m-2} + rmr^{m-1} - n^2r^m &= 0 \\ r^m[m(m-1) + m - n^2] &= 0 \\ m^2 - n^2 &= 0 \\ \Rightarrow m &= \pm n, \quad n = 1, 2, 3, \dots \end{aligned}$$

This yields the general solution: $F(r) = c_3r^n + c_4r^{-n}$. In order to satisfy the requirement that $\lim_{r \rightarrow 0} F(r) < \infty$, we must stipulate that $c_4 = 0$. Thus, for $\lambda > 0$, the product solution is of the form:

$$u_n(r, \theta) = F_n(r)G_n(\theta) = r^n(a_n \cos n\theta + b_n \sin n\theta)$$

Summarizing from all of the eigenvalues, the product solution is given in Equation 100.

$$u(r, \theta) = a_0 + \sum_{n=1}^{\infty} r^n(a_n \cos n\theta + b_n \sin n\theta) \quad (100)$$

Be careful not to forget the eigenfunction that you found for $\lambda = 0$ which was just a constant.

Step #5: Apply the boundary condition to solve for unknown constants. The explicit boundary condition that we have applies at $r = c$.

$$u(c, \theta) = a_0 + \sum_{n=1}^{\infty} c^n(a_n \cos n\theta + b_n \sin n\theta) = f(\theta)$$

On the left, we have an infinite series; on the right, we have a function that we want to represent with the infinite series. We need to determine how to set the unknown constants a_0 , a_n and b_n so that they are equal. How do we do this? Answer: we multiply both sides by an orthogonal function and integrate. Our orthogonal functions are:

$$\{1, \cos \theta, \cos 2\theta, \cos 3\theta, \dots, \sin \theta, \sin 2\theta, \sin 3\theta, \dots\}$$

This set of functions is orthogonal on the domain $\theta \in [0, 2\pi]$ with respect to weight function $p(\theta) = 1$. Carrying this out explicitly for the constant term:²

$$\begin{aligned} &\int_0^{2\pi} a_0(1) d\theta + \dots \\ &\sum_{n=1}^{\infty} c^n \left(\int_0^{2\pi} a_n \cos n\theta (1) d\theta + \int_0^{2\pi} b_n \sin n\theta (1) d\theta \right)^0 = \int_0^{2\pi} f(\theta)(1) d\theta \end{aligned}$$

so

$$a_0 = \frac{1}{2\pi} \int_0^{2\pi} f(\theta) d\theta$$

² i.e. We multiply both sides by a constant—1—and integrate. It does not matter what the value of the constant is, it is still orthogonal to all of the other eigenfunctions.

Carrying out the same process using $\cos n\theta$ and $\sin n\theta$ gives us:

$$a_n = \frac{1}{c^n \pi} \int_0^{2\pi} f(\theta) \cos n\theta \, d\theta \quad (101)$$

$$b_n = \frac{1}{c^n \pi} \int_0^{2\pi} f(\theta) \sin n\theta \, d\theta \quad (102)$$

Readers can confirm that:
 $\int_0^{2\pi} \cos(n\theta)^2 \, d\theta = \pi$ and
 $\int_0^{2\pi} \sin(n\theta)^2 \, d\theta = \pi$

MATLAB Implementation

As usual, it is helpful to implement a solution in MATLAB so you can visualize the results. We start by clearing out the MATLAB workspace and setting relevant parameters.

```

clear
clc
close 'all'

%% Parameters
c = 2;
N = 50;

fx_pick = 2;
%[1 | 2]
switch fx_pick
    case 1
        f = @(x) x;
    case 2
        f = @(x) ex1(x);
    otherwise
        error('Invalid case!');
end

```

Next we find the solution using the equations developed above.

```

%% solve the problem
Ao = (1./(2*pi))*integral(@(theta) f(theta),0,2*pi);

U = @(r,theta) Ao;

reltol = 1e-15;
for n = 1:N

    An = (1/((c.^n)*pi))*integral(@(theta) f(theta).*cos(n.*theta),...
        0,2*pi,'RelTol',reltol);    ❶
    Bn = (1/((c.^n)*pi))*integral(@(theta) f(theta).*sin(n.*theta),...
        0,2*pi,'RelTol',reltol);

    U = @(r,theta) U(r,theta)+ (r.^n).* (An*cos(n.*theta)+Bn*sin(n.*theta));

end

```

❶ The built-in function `integral()` has some name-value pairs to customize the function behavior. The name '`RelTol`' sets the *relative tolerance* parameter to the value provided. We will not go into great details here but, suffice it to say, a smaller value for '`RelTol`' gives a more precise result for the numeric integration.

Having constructed a representation of the solution, we make a plot so we can see if the solution makes sense.

```
%>> Make a Plot
NR = 100;
NT = 100;
R = linspace(0,c,NR);
THETA = linspace(0,2*pi,NT);
[RR,TT] = meshgrid(R,THETA);
UUp = U(RR,TT);

% plot in cartesian coordinates
XX = RR.*cos(TT); % get cartesian coordinate equivalents
YY = RR.* sin(TT);

figure(1)
surf(XX,YY,UUp, 'edgecolor', 'none');
colormap('jet');%<-- consider alternate colormaps
c = colorbar;%<-- add a colorbar
c.Label.String = 'Temperature'; %<-- give colorbar a label
title('Lecture 30 Example', 'fontsize',16,'fontweight','bold');
xlabel('X', 'fontsize',14, 'fontweight', 'bold');
ylabel('Y', 'fontsize',14, 'fontweight', 'bold');
zlabel('U', 'fontsize',14, 'fontweight', 'bold');
grid on
set(gca, 'fontsize',12, 'fontweight', 'bold');
```

The resulting plot is shown in Figure 49. The code for the local function $f(x) = \text{ex1}(x)$ is provided below.

```
%>> Local functions
function y = ex1(theta)
[m,n] = size(theta);
y = nan(m,n);
for i = 1:length(theta)
    if(theta(i)>= 0 && (theta(i)< pi/2)
        y(i) = 1;
    else
        y(i) = 0;
    end
end
end
```

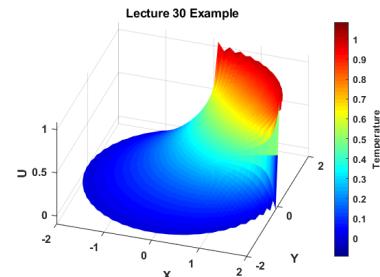


Figure 49: Solution for the case where $f(x) = \text{ex1}(x)$.

Note the “wigginess” in the solution at the points of discontinuity.

Lecture 31 - Polar Coordinates with Angular Symmetry

Objectives

- Carry out the separation of variables processes to solve the wave equation in polar coordinates.
- Show an example implementation in MATLAB.

Wave Equation in Polar Coordinates

In a few past lectures we have dealt with the wave equation on a line as shown in Equation 103.

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2} \quad (103)$$

This is actually a quite specific articulation of the wave equation, tailored only for a one-dimensional wave. Waves happen in more than one dimension, however, and we would like to describe those also. Equation 104 gives a more general expression of the wave equation using the Laplacian operator that is valid for any number of dimensions over finite or infinite domains.

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \nabla^2 u \quad (104)$$

If we specialize the Laplacian operator for 1D Cartesian coordinates then we recover Equation 103.

IN THIS LECTURE we will solve the wave equation in polar coordinates. Specializing the Laplacian for polar coordinates and inserting into the wave equation gives us:

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} \right) \quad (105)$$

We will use this equation to model radial vibrations in a circular membrane for which the membrane is fixed (no displacement) all around the periphery at $r = c$. We will also assume that the initial

displacement and velocity of the membrane are functions of radius only. Because the boundary condition and both initial conditions are constant for all angular positions, we should expect that *the solution* will also be constant for all angular positions; in particular, we expect $\partial u / \partial \theta$ and $\partial^2 u / \partial \theta^2$ to equal zero.¹ The boundary value problem for this is given below.

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial t^2} = \alpha^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} \right), \quad \alpha > 0, \quad 0 < r < c, \quad t > 0$$

$$\text{Boundary Condition : } u(c, t) = 0, \quad t > 0$$

$$\text{Initial Conditions : } u(r, 0) = f(r), \quad u_t(r, 0) = g(r), \quad 0 < r < c$$

¹ Conversely, if either the boundary or initial conditions were a function of angular position, θ , this assumption would not be true and we would need to use the full form of the wave equation in polar coordinates given in equation 105.

WE WILL SOLVE this boundary value problem using separation of variables.

Step #1: Assume a product solution.

$$u(r, t) = F(r)G(t)$$

Step #2: Insert the product solution into the governing equation.

$$\begin{aligned} \frac{\partial^2}{\partial t^2} [F(r)G(t)] &= \alpha^2 \left\{ \frac{\partial^2}{\partial r^2} [F(r)G(t)] + \frac{1}{r} \frac{\partial}{\partial r} [F(r)G(t)] \right\} \\ FG_{tt} &= \alpha^2 \left(F_{rr}G + \frac{1}{r} F_r G \right) \end{aligned}$$

Step #3: Separate variables.

$$\begin{aligned} \frac{FG_{tt}}{\alpha^2 FG} &= \frac{\alpha^2 \left(F_{rr}G + \frac{1}{r} F_r G \right)}{\alpha^2 FG} \\ \frac{G_{tt}}{\alpha^2 G} &= \frac{F_{rr} + \frac{1}{r} F_r}{F} = -\lambda \end{aligned}$$

So the separated equations are:

$$\begin{aligned} F_{rr} + \frac{1}{r} F_r + \lambda F &= 0 \\ G_{tt} + \alpha^2 \lambda G &= 0 \end{aligned}$$

Step #4: Apply boundary conditions to find non-trivial solutions.

We will take a short-cut here by assuming that the separated solution $G(\theta)$ must be periodic and, having done this before, we know that

this implies $\lambda > 0$. Setting $\lambda = \nu^2$, $\nu > 0$, we get:

$$\begin{aligned} G_{tt} + \alpha^2 \nu^2 G &= 0 \\ r^2 F_{rr} + r F_r + \alpha^2 \nu^2 F &= 0 \end{aligned}$$

where, to put the equation for $F(r)$ in a more familiar form, we multiplied through by r^2 . The general solution for both equations is:

$$\begin{aligned} F(r) &= c_1 J_0(\nu r) + c_2 Y_0(\nu r) \\ G(t) &= c_3 \cos(\nu \alpha t) + c_4 \sin(\nu \alpha t) \end{aligned}$$

IN THE r -COORDINATE we have only one explicit boundary condition. There is, however, another condition of which we must always be mindful, particularly in polar coordinates when $r = 0$ is part of the domain and that is: $\lim_{r \rightarrow 0} u(r, \theta) < \infty$. Once again, drawing upon our knowledge of Bessel functions, we know that $Y_0(r)$ diverges to negative infinity as r goes to zero. Thus we must set $c_2 = 0$ and the radial equation simplifies to:

$$F(r) = c_1 J_0(\nu r)$$

WE ALSO NEED to satisfy the boundary condition at $r = c$:

$$F(c) = c_1 J_0(\nu c) = 0$$

which implies that νc needs to be a positive root of $J_0(r)$. As we learned in lecture 19, $J_0(r)$ has infinitely many positive roots and we can find them with the assistance of the function `besselzero()` that we obtained from the MATLAB file exchange. If we denote the n^{th} root of J_0 as $k_{0,n}$, then our eigenvalues must be equal to:

$$\nu_n = \frac{k_{0,n}}{c}$$

so the full product solution will be:

$$u(r, t) = \sum_{n=1}^{\infty} J_0(\nu_n r) [a_n \cos \alpha \nu_n t + b_n \sin \alpha \nu_n t]$$

Step #5: Satisfy the initial conditions.

WE NOW HAVE two infinite sets of unknowns: a_n and b_n . We will resolve these constants through the initial conditions as applied below:

$$\begin{aligned} u(r, 0) &= \sum_{n=1}^{\infty} \left(a_n \cos 0 + b_n \sin 0 \right)^0 J_0(\nu_n r) = f(r) \\ &\quad \sum_{n=1}^{\infty} a_n J_0(\nu_n r) = f(r) \end{aligned}$$

Alert readers who remember the analysis from Lecture 30 will note that $\lambda = 0$ should also be included since, the corresponding eigenfunction in time $G(t) = c_1$, meets the periodicity requirement. If we followed that path we would find that, for $\lambda = 0$, the separated equation in the r -direction is the Cauchy-Euler equation. You will find that only the trivial solution, $F(r) = 0$, will satisfy the equation and the homogeneous Dirichlet boundary condition at $r = c$. In Lecture 31 the boundary condition at $r = c$ was not homogeneous so the $\lambda = 0$ eigenvalue had a non-trivial eigenfunction.

On the left, we have an infinite linear combination of eigenfunctions, $J_0(\nu_n r)$, on the right we have a function we are trying to represent, $f(r)$. We need to determine the values a_n so that the two are equal. How do we do this? We multiply both sides by an orthogonal function—and weight function—and integrate. As we learned previously, the weight function is $p(r) = r$. Doing this explicitly for a_1 gives us:

$$a_1 \int_0^c J_0(\nu_1 r)^2 r dr + a_2 \underbrace{\int_0^c J_0(\nu_2 r) J_0(\nu_1 r) r dr}_{\rightarrow 0 \text{ by orthogonality}} + \dots = \int_0^c f(r) J_0(\nu_1 r) r dr$$

and, in general, a_n is given by:

$$a_n = \frac{\int_0^c f(r) J_0(\nu_n r) r dr}{\int_0^c J_0(\nu_n r)^2 r dr} \quad (106)$$

WE USE THE OTHER initial condition to solve for b_n :

$$\begin{aligned} u_t(r, o) &= \sum_{n=1}^{\infty} (-a_n \alpha \nu_n \sin(0) + b_n \alpha \nu_n \cos 0) J_0(\nu_n r) = g(r) \\ &\quad \sum_{n=1}^{\infty} b_n \alpha \nu_n J_0(\nu_n r) = g(r) \end{aligned}$$

As before we will multiply by our orthogonal functions and weight function to find the coefficients b_n :

$$b_n = \frac{1}{\alpha \nu_n} \frac{\int_0^c g(r) J_0(\nu_n r) r dr}{\int_0^c J_0(\nu_n r)^2 r dr} \quad (107)$$

In summary, the solution is:

$$u(r, t) = \sum_{n=1}^{\infty} [a_n \cos \alpha \nu_n t + b_n \sin \alpha \nu_n t] J_0(\nu_n r) \quad (108)$$

where a_n and b_n are given by equation 106 and 107 respectively.

Implementation in MATLAB

The MATLAB that we will use should begin to look routine by now, but since this is the first time we used Fourier-Bessel expansions in solving a boundary value problem, the code will be listed here.

We start by clearing the workspace and defining problem parameters.

All terms other than the first one on the left hand side is zero due to orthogonality of $J_0(\nu r)$ on the interval $r \in [0, c]$ with respect to weight function $p(r) = r$.

```

clear
clc
close 'all'

%% Parameters
N = 50; % number of modes
c = 1; % radius of the circle
a_sq = 1.0; % "stiffness" parameter
a = sqrt(a_sq);

example = 2;
%example = [1 | 2]
switch example
    case 1
        f = @(r) ex1(r,c); % initial position
        g = @(r) o.*r; % initial velocity
    case 2
        b = 0.2;
        f = @(r) o.*r;
        g = @(r) ex2(r,b);
    otherwise
        error('Unexpected example number!!');
end

```

Next we use besselzero() to get the roots of J_0 , compute the coefficients a_n and b_n , and build the solution $u(r,t)$.

```

%% Get eigenvalues
k = besselzero(o,N,1); ❶
nu = k./c;

F = @(r,n) besselj(o,nu(n).*r);

U = @(r,t) o;

for n = 1:N
    % compute An
    an = integral(@(r) r.*f(r).*F(r,n),o,c) / ...
        integral(@(r) r.*(F(r,n).^2),o,c);
    % compute Bn
    bn = integral(@(r) r.*g(r).*F(r,n),o,c) / ...
        (a.*nu(n).*integral(@(r) r.*F(r,n).^2,o,c));
    % add the term to our solution
    U = @(r,t) U(r,t) + (an*cos(a*nu(n).*t) + ...
        bn*sin(a*nu(n).*t)).*F(r,n);
end

```

❶ Reminder that in order to use besselzero() in a script, you need to have a copy of besselzero.m in the current folder or otherwise on the MATLAB path.

We can make a dynamic plot if we like:

```

%% Plot the solution
NR = 20;
NTHETA = 20;
Tmax = 10;
NT = 50;
R = linspace(o,c, NR);
THETA = linspace(o, 2*pi, NTHETA);
[RR,TT] = meshgrid(R,THETA);
XX = RR.*cos(TT);
YY = RR.*sin(TT);

```

```

T = linspace(0,Tmax,NT);

for t = 1:NT
    UUp = U(RR,T(t));
    surf(XX,YY,UUp,'facecolor','none'); ❷
    title_str = sprintf('Lecture 31 example, t = %g \n',T(t));
    title(title_str,'fontsize',16,'fontweight','bold');
    xlabel('X','fontsize',14,'fontweight','bold');
    ylabel('Y','fontsize',14,'fontweight','bold');
    zlabel('U','fontsize',14,'fontweight','bold');
    set(gca,'fontsize',12,'fontweight','bold');
    axis([-c+1 c+1 -(c+1) c+1 -2 2]); ❸
    pause(0.5*Tmax/(NT-1));
end

```

54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69

❷ Use the argument 'facecolor' and value 'none' to get a plot resembling a wire mesh.

❸ Using the axis command here allows you to fix the axis size and prevent re-scaling with each time step which would make the wave-like motion harder to see and interpret.

And the local functions, as always, are placed at the end of the script.

```

%% Local functions
function y = ex1(r,c)
[m,n] = size(r);
y = nan(m,n);
for i = 1:length(r)
    if (r(i) < c/3)
        y(i) = 1;
    else
        y(i) = 0;
    end
end
end

function y = ex2(r,b)
[m,n] = size(r);
y = nan(m,n);
vo = 10;
for i = 1:length(r)
    if (r(i) < b)
        y(i) = -vo;
    else
        y(i) = 0;
    end
end
end

```

70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94

Plots of the solution at various times with boundary conditions selected for example 2 are shown in Figure 50.

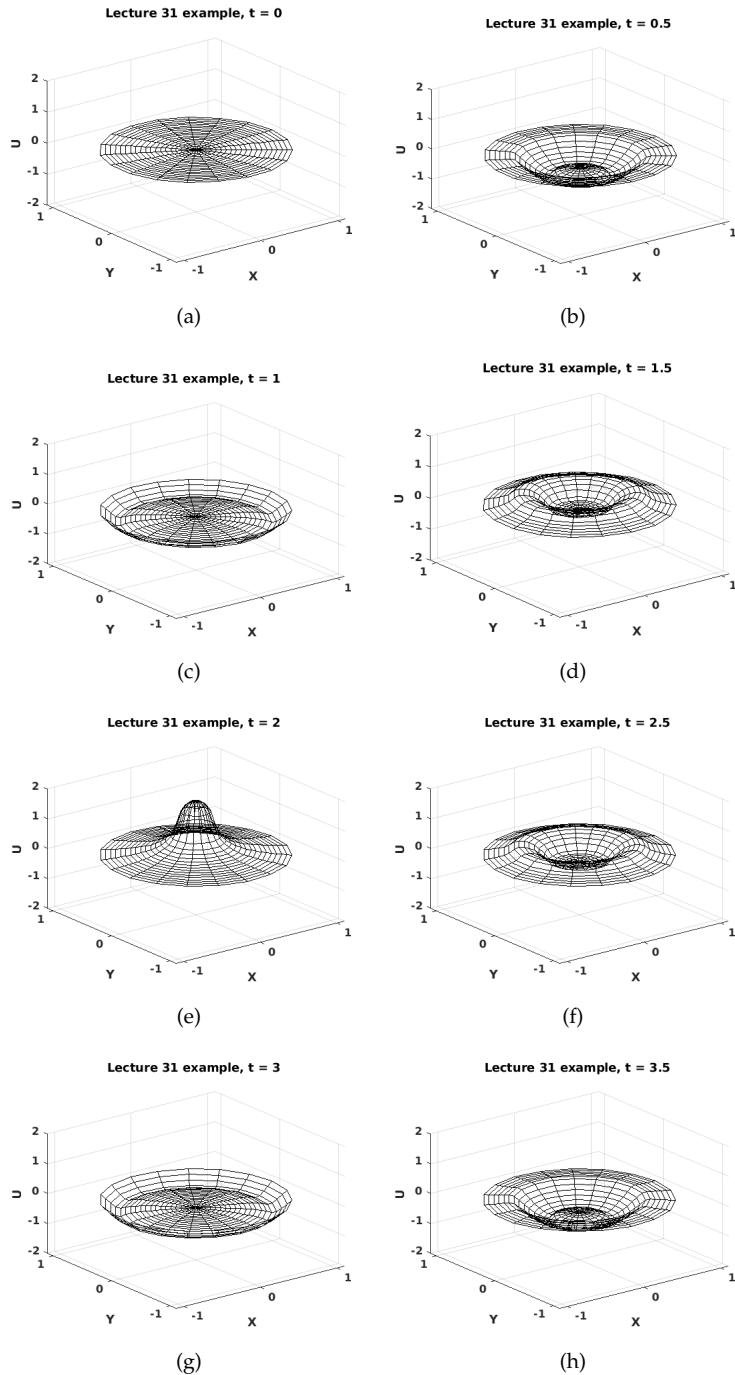


Figure 50: Plots solution with boundary conditions selected for example 2.

Assignment #11

1. Solve the following boundary value problem

Governing Equation : $\frac{\partial^2 u}{\partial t^2} = \alpha^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \quad \alpha > 0, \quad \begin{matrix} 0 < x < \pi \\ 0 < y < \pi \end{matrix}$

Boundary Conditions : $u(0,y,t)=0, \quad u(\pi,y,t)=0, \quad t > 0$
 $u(x,0,t)=0, \quad u(x,\pi,t)=0'$

Initial Conditions : $u(x,y,0)=xy(x-\pi)(y-\pi), \quad 0 < x < \pi, \quad 0 < y < \pi$
 $u_t(x,0)=0$

where $\alpha^2 = 1$. Compute the first 10 modes in x and y with MATLAB. Create a surface plot of the solution at $t = 10$ and save the plot in *.png format. Include the published version of your MATLAB code and the saved figure in your submission.

2. Solve the following boundary value problem:

Governing Equation : $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0, \quad \begin{matrix} 0 < x < a \\ 0 < y < b \\ 0 < z < c \end{matrix}$

Boundary Conditions : $u(0,y,z)=0, \quad u(a,y,z)=0$
 $u(x,0,z)=0, \quad u(x,b,z)=0$
 $u(x,y,0)=0, \quad u(x,y,c)=f(x,y)$

3. If the boundaries $\theta = 0$ and $\theta = \pi$ of a semicircular plate of radius 2 are insulated, we then have:

$$u_\theta(r, 0) = 0, \quad u_\theta(r, \pi) = 0, \quad 0 < r < 2$$

Find the steady-state temperature, $u(r, \theta)$, if:

$$u(2, \theta) = \begin{cases} u_0, & 0 < \theta < \pi/2 \\ 0, & \pi/2 < \theta < \pi \end{cases}$$

where u_0 is a constant. Solve the Laplace equation for this case. [Note: contrary to the typical case, the analytic evaluation of the integral for Fourier coefficients is straight-forward in this problem; you should actually do the integration analytically.]

4. Solve the following BVP to find the steady-state temperature in a quarter circular plate.

Governing Equation : $\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0, \quad 0 < r < c, \quad 0 < \theta < \pi/2$

Boundary Conditions : $u(r, 0) = 0, \quad u(r, \pi/2) = 0, \quad 0 < r < c$
 $u(c, \theta) = f(\theta), \quad 0 < \theta < \pi/2$

Lecture 32 - Laplace Equation in Cylindrical Coordinates

Objectives

- Solve the steady-state heat equation (Laplace equation) in cylindrical coordinates for two cases.
- Revisit modified Bessel functions of the first and second kind.

Steady-State Temperature in a Circular Cylinder - Case I

Consider the boundary value problem below based on the steady-state heat equation in cylindrical coordinates. A schematic of the problem is shown in Figure 51.

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{\partial^2 u}{\partial z^2} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0, \quad 0 < r < 2, \quad 0 < z < 4 \\ 0 < \theta < 2\pi$$

$$\text{Boundary Conditions : } u(r, 0) = 0, \quad u(r, 4) = u_0, \quad u(2, z) = 0$$

Based on the boundary conditions provided for the problem—none of which are dependent on θ —we can expect that the solution also will be independent of θ and so the governing equation can be simplified to:

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{\partial^2 u}{\partial z^2} = 0$$

As is becoming the usual, we will solve this problem using separation of variables.

Step #1: Assume a product solution:

$$u(r, z) = F(r)G(z)$$

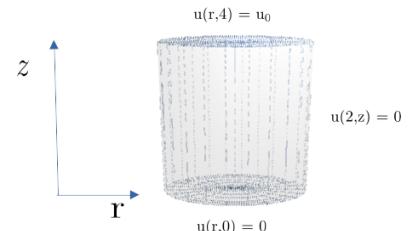


Figure 51: Schematic of Case I

Step #2: Insert the product solution into the governing equation.

$$\frac{\partial^2}{\partial r^2} [F(r)G(z)] + \frac{1}{r} \frac{\partial}{\partial r} [F(r)G(z)] + \frac{\partial^2}{\partial z^2} [F(r)G(z)] = 0$$

$$F_{rr}G + \frac{1}{r}F_rG + FG_{zz} = 0$$

Step #3: Separate the variables.

$$\frac{F_{rr}G}{FG} + \frac{1}{r} \frac{F_rG}{FG} + \frac{FG_{zz}}{FG} = 0$$

$$\frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} + \frac{G_{zz}}{G} = 0$$

$$\frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} = -\frac{G_{zz}}{G} = -\lambda$$

Thus the separated equations are:

$$r^2 F_{rr} + rF_r + \lambda r^2 F = 0, \quad \text{and} \quad G_{zz} - \lambda G = 0$$

Step #4: Apply boundary conditions to determine non-trivial product solution(s):

In the z -direction we have only one homogeneous boundary condition whereas in the r -direction, the only boundary condition we have is homogeneous. Therefore we should analyze the equation of $F(r)$ to determine values of λ that will admit non-trivial solutions.

$\lambda = 0$:

In this case the boundary value problem for $F(r)$ is a Cauchy-Euler equation:

$$r^2 F_{rr} + rF_r = 0$$

$$r^2[m(m-1)r^{m-2}] + rmr^{m-1} = 0$$

$$r^m[m(m-1) + m] = 0 \Rightarrow m^2 = 0$$

$$\Rightarrow F(r) = c_1 + c_2 \ln r$$

We must set $c_2 = 0$ so that $F(r)$ will remain bounded as $r \rightarrow 0$. Since $u(2, z) = 0$, we must stipulate that $F(2) = c_1 = 0$ so that only the trivial solution will satisfy the boundary conditions if $\lambda = 0$.

$\lambda < 0$: where $\lambda = -\alpha^2$, $\alpha > 0$.

In this case the boundary value problem for $F(r)$ is:

$$r^2 F_{rr} + rF_r - \alpha^2 r^2 F = 0$$

Which we recognize as the parametric modified Bessel's equation of order zero. The general solution is:

$$F(r) = c_1 I_0(\alpha r) + c_2 K_0(\alpha r)$$

Again, here we multiply the equation for $F(r)$ by r^2 to change the equation into a familiar form.

Reminder: For a Cauchy-Euler equation, we assume the solution is of the form $F(r) = r^m$. In this case, where the resulting auxiliary equation has a double-root— $m = 0, 0$ —the first solution is a constant (c_1) and the second solution is a constant times $\ln r$ so that it is linearly independent.

Reminder: The parametric modified Bessel's equation is of the form:

$$r^2 F_{rr} + rF_r - (\alpha^2 r^2 + v^2)F = 0$$

where v is the order of the equation.

In case one is not familiar with modified Bessel functions of the first and second kind of order zero— $I_0(r)$ and $K_0(r)$, respectively—the reader might be relieved to be informed that MATLAB has built-in functions available: `besseli()` and `besselk()`. A plot of these functions is shown in Figure 52.

The salient facts about these functions are that $K_0(\alpha r)$ diverges to infinity as $r \rightarrow 0$ and that $I_0(\alpha r)$ is strictly positive for $\alpha r > 0$. Thus $c_1 = c_2 = 0$ and only the trivial solution satisfies the boundary conditions in the case $\lambda < 0$.

$\lambda > 0$: where $\lambda = \alpha^2$, $\alpha > 0$.

In this case the boundary value problem for $F(r)$ is:

$$r^2 F_{rr} + r F_r + \alpha^2 r^2 F = 0$$

Which we recognize as the parametric Bessel's equation of order zero. The general solution is:

$$F(r) = c_1 J_0(\alpha r) + c_2 Y_0(\alpha r)$$

Since $Y_0(\alpha r)$ diverges to negative infinity as $r \rightarrow 0$, we must set $c_2 = 0$. In order to satisfy the boundary condition at $r = 2$:

$$\begin{aligned} F(2) &= c_1 J_0(2\alpha) = 0 \\ \Rightarrow \alpha_n &= \frac{k_{0,n}}{2} \end{aligned}$$

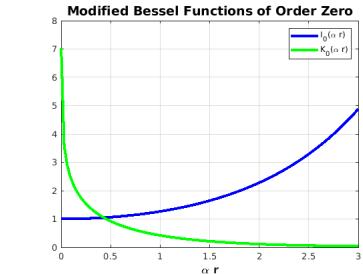


Figure 52: Plots of $I_0(\alpha r)$ and $K_0(\alpha r)$ for $\alpha r > 0$.

Use the MATLAB function `besselzero()` to get the roots to $J_0(\alpha r)$

where we have, on the fly, adopted the notation α_n —the n^{th} eigenvalue—and $k_{0,n}$ as the n^{th} root of the Bessel Function of the first kind of order zero.

For $\lambda = \alpha^2$, the boundary value problem in the z -direction is:

$$G_{zz} - \alpha^2 G = 0$$

Where the general solution is:

$$G(z) = c_3 \cosh \alpha z + c_4 \sinh \alpha z$$

Applying the homogeneous boundary condition at $z = 0$ we get:

$$\begin{aligned} G(0) &= c_3 \cosh 0 + c_4 \sinh 0 = 0 \\ &= c_3(1) + c_4(0) = 0 \\ \Rightarrow c_3 &= 0 \end{aligned}$$

The solution in the z -direction is, therefore:

$$G(z) = c_4 \sinh \alpha_n z$$

After combining constants, as usual, the full product solution is:

$$u(r, z) = \sum_{n=1}^{\infty} a_n J_0(\alpha_n r) \sinh \alpha_n z \quad (109)$$

where $\alpha_n = k_{0,n}/2$.

Step #5: Apply the remaining boundary condition to determine unknown coefficients.

The boundary condition that we have not yet used is the non-homogeneous condition applied at the top of the cylinder: $u(r, 4) = u_0$. We must find suitable values of a_n for $n = 1, 2, 3, \dots$ such that:

$$u(r, 4) = \sum_{n=1}^{\infty} a_n J_0(\alpha_n r) \sinh 4\alpha_n = u_0$$

Once again we have an infinite linear combination of eigenfunctions on the left and on the right we have a given function—in this case a constant, u_0 . We want them to be equal; our job is to determine a_n such that they are equal. How do we do this? We multiply both sides by our orthogonal function, and weight function, and integrate. The resulting equation for a_n is given in Equation 110.

$$a_n = \frac{u_0 \int_0^2 J_0(\alpha_n r) r dr}{\sinh 4\alpha_n \int_0^2 J_0(\alpha_n r)^2 r dr} \quad (110)$$

We can create a MATLAB script to construct an approximate solution for a specified value of u_0 and a finite number of eigenmodes. The code for $u_0 = 5$ and $N = 25$ is shown in the listing below.

```

clear
clc
close 'all'

%% Parameters
R = 2; Z = 4;
Uo = 5; % given temperature on top surface of the cylinder
N = 25;
k = besselzero(0,N,1); % get first n zeros of Jo    ①
alpha = k/R;

%% Construct Solution
A = nan(N,1);
u = @(r,z) 0; % initialize the series
for n = 1:N
    A(n) = (Uo/(sinh(Z*alpha(n)))).*...
        integral(@(r) besselj(0,alpha(n)*r).*r,o,R)./
        integral(@(r) besselj(0,alpha(n)*r).*...
            besselj(0,alpha(n)*r).*r,o,R);

    % update the series with the next term
    u = @(r,z) u(r,z) + ...
        A(n)*besselj(0,alpha(n)*r).*sinh(z*alpha(n));
end

```

Do not forget to include the constant $\sinh 4\alpha_n$; it is easy to leave off when you are in a hurry.

① Recall that the first argument to `besselzero()` is the order of the Bessel function, the second argument is the number of desired roots, and the third argument is the *kind* of Bessel function—first or second.

```

%% Plot results
Rv = linspace(0,R,100);
Zv = linspace(0,Z,200);

[RR,ZZ] = meshgrid(Rv,Zv);
UU = u(RR,ZZ);

figure(1)
surf(Rv,Zv,UU, 'edgecolor', 'none');
title('Laplacian in a Cylinder: Case 1',...
      'fontsize',18,'fontweight','bold');
xlabel('R', 'fontsize',16,'fontweight','bold');
ylabel('Z', 'fontsize',16,'fontweight','bold');
zlabel('U', 'fontsize',16,'fontweight','bold');
view([65 10]); ②

```

② The `view()` function allows you to rotate the plotted image to change the azimuth and elevation. It is best to experiment a bit with different arguments to find a view that best presents the results.

A cross-section of the temperature distribution is shown in Figure 53.

As we have come to expect, the approximate solution has a certain amount of “wigginess” at points of discontinuity. In this case, the issue is along the outer rim of the top of the cylinder; on the top of the cylinder, the solution is constant at $u_0 = 5$; on the outer surface of the cylinder the solution is fixed at $u = 0$. Along the outer rim of the top is where these conflicting solutions come together and the “wigginess” is the result. If we increase the number of eigenmodes a more sharp solution can be obtained. A plot with $N = 100$ eigenmodes is shown in Figure 54.

Steady-State Temperature in a Circular Cylinder - Case II

Consider another boundary value problem similar to the first except, in this case, there are homogeneous boundary conditions on the top and bottom while the side of the cylinder is maintained at a prescribed temperature. A schematic is shown in Figure 55.

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{\partial^2 u}{\partial z^2} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0, \quad 0 < r < 1, \quad 0 < z < 1 \\ 0 < \theta < 2\pi$$

$$\text{Boundary Conditions : } u(r,0) = 0, \quad u(r,1) = 0, \quad u(1,z) = 1 - z$$

As with Case I, the boundary conditions are all independent of θ so we expect the solution to be independent of θ also, and we can simplify the governing equation to:

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{\partial^2 u}{\partial z^2} = 0$$

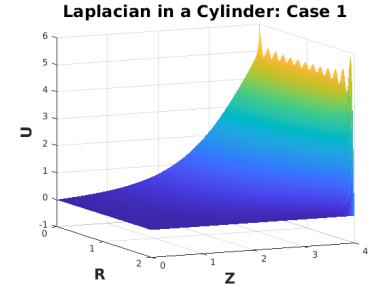


Figure 53: Cross section of Case I solution for $N=25$.

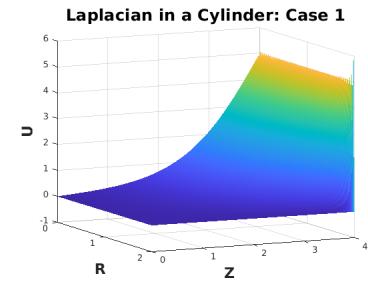


Figure 54: Cross section of Case I solution for $N=100$.

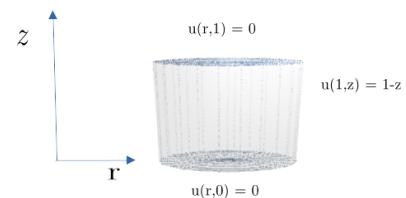


Figure 55: Schematic of domain and boundary conditions for Case I.

THE DETAILS OF Steps #1 through #3 of separation of variables are exactly the same for this case. The separated equations are, again:

$$\begin{aligned} r^2 F_{rr} + rF_r + \lambda r^2 F &= 0 \\ G_{zz} - \lambda G &= 0 \end{aligned}$$

What makes this case different is the boundary conditions. Specifically, in the r -direction we no longer have a homogeneous boundary condition but in the z -direction, we do. Therefore we will use the equation for $G(z)$ to determine values of λ that admit non-trivial solutions.

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

We have seen this combination of boundary value problem and boundary conditions numerous times before. $G_{zz} - \lambda G = 0$ and $G(0) = G(1) = 0$. If $\lambda = 0$ we know the solution is linear, but the only line that is zero at both ends is $G(z) = 0$ so we can discard that case quickly. If $\lambda > 0$, or $\lambda = \alpha^2$, $\alpha > 0$, then we have

$$G_{zz} - \alpha^2 G = 0$$

with general solution: $G(z) = c_1 \cosh \alpha z + c_2 \sinh \alpha z$. The condition $G(0) = 0$ implies that $c_1 = 0$. The condition $G(1) = 0$ means that $c_2 \sinh \alpha = 0$ but, as we have seen before, $\sinh \alpha$ is never zero for $\alpha > 0$. Thus we have no choice but to also set $c_2 = 0$.

Consequently, our only remaining option is $\lambda < 0$, where we set $\lambda = -\alpha^2$, $\alpha > 0$. This gives us:

$$G_{zz} + \alpha^2 G = 0$$

with general solution: $G(z) = c_1 \cos \alpha z + c_2 \sin \alpha z$.

When we apply the boundary conditions we get:

$$\begin{aligned} G(z) &= c_1 \cos \alpha z + c_2 \sin \alpha z \\ G(0) &= c_1(1) + c_2(0) = 0 \\ \Rightarrow c_1 &= 0 \\ G(1) &= c_2 \sin \alpha = 0 \end{aligned}$$

The last condition is satisfied if $\alpha_n = n\pi$ with n a positive integer.

With $\lambda = -\alpha^2$, the equation for $F(r)$ is:

$$r^2 F_{rr} + rF_r - \alpha^2 r^2 F = 0$$

We recognize this as the parametric modified Bessel's equation of order zero. The general solution is: $F(r) = c_3 I_0(\alpha r) + c_4 K_0(\alpha r)$. This

is fresh in our mind, so we immediately conclude that $c_4 = 0$ since $K_0(\alpha r)$ diverges as $r \rightarrow 0$. The product solutions for this problem are, therefore:

$$u(r, z) = \sum_{n=1}^{\infty} c_n I_0(n\pi r) \sin n\pi z \quad (111)$$

Step #5: Apply the remaining boundary condition to determine the unknown coefficients.

The boundary condition that we have not yet used is the non-homogeneous condition applied on the outer surface of the cylinder: $u(1, z) = 1 - z$.

We must find suitable values of c_n such that:

$$u(1, z) = \sum_{n=1}^{\infty} c_n I_0(n\pi) \sin n\pi z = 1 - z$$

Here again, of course, we need to multiply both sides by an orthogonal function (and weight function) and integrate. In this case, however, the orthogonal set of functions is $\sin n\pi z$ and the weight function is $p(z) = 1$. Carrying out this (by now routine) task, we obtain the expression for c_n as:

$$c_n = \frac{\int_0^1 (1-z) \sin n\pi z \, dz}{I_0(n\pi) \int_0^1 \sin(n\pi z)^2 \, dz} \quad (112)$$

In summary, for Case II, the solution is given by Equation 111 with the coefficients given by Equation 112. A plot of the solution for $N=150$ is given in Figure 56.

THE MATLAB CODE needed to construct this solution is shown in the listing below.

```

clear
clc
close 'all'

%% Case 2
R = 1; Z = 1;
g = @(z) 1-z; % temperature boundary condition
N = 150;

c = nan(N,1);
u = @(r,z) 0; % initialize the series
for n = 1:N
    c(n) = (1./besselj(0,n*pi)).*...
        integral(@(z) g(z).*sin(n*pi*z),0,Z)./...
        integral(@(z) sin(n*pi*z).*sin(n*pi*z),0,Z);

    % update the series with the next term
    u = @(r,z) u(r,z) + ...
        c(n)*besselj(0,n*pi*r).*sin(n*pi*z);
end

```

Students sometimes struggle with determining which term is the orthogonal function. The key is to realize that $I_0(n\pi)$ is not really a function at all but a constant—albeit a tricky one to evaluate.

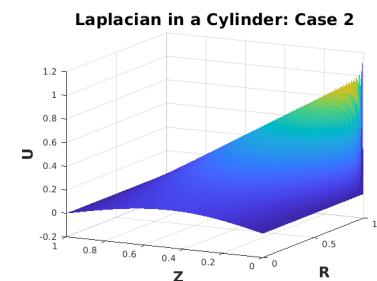


Figure 56: Solution for Case II with $N=150$.

```
%% make surface plot  
Rv = linspace(0,R,100);  
Zv = linspace(0,Z,200);  
  
[RR,ZZ] = meshgrid(Rv,Zv);  
UU = u(RR,ZZ);  
  
figure(1)  
surf(Rv,Zv,UU,'edgecolor','none');  
title('Laplacian in a Cylinder: Case 2',...  
    'fontsize',18,'fontweight','bold');  
xlabel('R','fontsize',16,'fontweight','bold');  
ylabel('Z','fontsize',16,'fontweight','bold');  
zlabel('U','fontsize',16,'fontweight','bold');  
view([-63 15]);
```

Lecture 33 - Introduction to the Neutron Diffusion Equation in Cylindrical Coordinates

Objectives

- Describe the neutron diffusion equation.
- Solve the neutron diffusion equation for a bare, homogeneous, finite, cylindrical reactor.

Background and Introduction

In a nuclear reactor, power is produced by neutron-induced fission in fuel materials in the core. To analyze this phenomena, we need to have a model of neutron transport and interaction. The dependent variable in this analysis is the *neutron flux* (ϕ) which is the product of neutron density—number of neutrons per unit volume—and the average speed of the neutrons.¹ Since almost all operating reactor cores are arranged as a nearly cylindrical array of fuel assemblies, we will idealize the geometry of a reactor core as a smooth finite cylinder. To make the calculations amenable to analytical methods, we will make the following additional assumptions:

1. The neutron flux is at steady state and equal to zero at the core boundary.² The exterior of the core is assumed to be *bare* with no reflector materials present.
2. Since the flux is assumed to be zero at the core boundary it will be natural to also assume that the neutron flux is constant with respect to variations in angular position within the cylinder.
3. The fuel, cladding, coolant, and surrounding structural materials are assumed to be a homogeneous medium.
4. Variation in neutron energy will be ignored and their individual direction of travel will also be ignored.³ Thus neutron flux will be a function of radial and axial position only— $\phi(r, z)$.

¹ Equivalently, the neutron flux can be understood as the total distance traveled by all neutrons per unit volume and per unit time. The units for flux are: $\frac{\text{neutrons}}{\text{cm}^2 \cdot \text{s}}$ although “neutrons” are not technically a unit.

² This is referred to as a *vacuum boundary condition*. The flux is not actually zero at the boundary but is *assumed* to be zero a certain distance outside the boundary, referred to as the *extrapolation distance* (d). For students studying reactor physics you will learn that the extrapolation distance is assumed to be proportional to the diffusion coefficient: $d = 2.13D$. If D is small, as is the case for some reactors, $d \approx 0$ is not a terrible approximation.

³ Instead of accounting for neutron direction of travel we will assume that the population of neutrons *diffuse*, like heat or perfume, from high concentration to low.

Neutron Diffusion Equation

Collectively, the above assumptions lead us to use the *Diffusion Theory approximation* for neutron transport. The general form of the neutron diffusion equation is given in Equation 113.

$$D\nabla^2\phi - \Sigma_a\phi + \nu\Sigma_f = 0 \quad (113)$$

where D is the diffusion coefficient,⁴ Σ_a and Σ_f are the macroscopic absorption and fission cross sections for the core material,⁵ and ν is the average number of neutrons released per fission event.

If we specify the form of the Laplacian operator for cylindrical coordinates with angular symmetry and define $B^2 = \frac{\nu\Sigma_f - \Sigma_a}{D}$,⁶ the boundary value problem can be expressed as follows:

$$\text{Governing Equation : } \frac{\partial^2\phi}{\partial r^2} + \frac{1}{r}\frac{\partial\phi}{\partial r} + \frac{\partial^2\phi}{\partial z^2} + B^2\phi = 0, \quad 0 < r < R, \quad -H/2 < z < H/2$$

$$\text{Boundary Conditions : } \phi(r, -H/2) = \phi(r, H/2) = 0, \quad \phi(R, z) = 0$$

In addition to angular symmetry, this problem is symmetric in the axial direction; the top-half and bottom-half are the same. Thus we will only model the top-half of the domain and impose a *symmetry* boundary condition at $z = 0$; specifically we will assert: $\phi_z(r, 0) = 0$. Consequently, we will no longer need to enforce the condition $\phi(r, -H/2)$ for this problem.

There is another condition that $\phi(r, z)$ needs to satisfy, and that is it must be *non-negative*. Since flux is the product of neutron speed times the neutron density—two quantities that must be positive to be physically realistic—the flux itself must be positive everywhere in the domain. We will see the impact of this condition as we solve the problem.

LET US NOW turn ourselves to the task of solving this boundary value problem using, as usual, separation of variables.

Step #1: Assume a product solution:

$$\phi(r, z) = F(r)G(z)$$

Step #2: Insert the product solution into the governing equation.

$$\begin{aligned} \frac{\partial^2}{\partial r^2}[F(r)G(z)] + \frac{1}{r}\frac{\partial}{\partial r}[F(r)G(z)] + \frac{\partial^2}{\partial z^2}[F(r)G(z)] + B^2F(r)G(z) &= 0 \\ F_{rr}G + \frac{1}{r}F_rG + FG_{zz} + B^2FG &= 0 \end{aligned}$$

⁴ The diffusion coefficient is a material property that is proportional to the average distance a neutron can travel in a medium without interaction with the atoms of a material. If a neutron is expected to travel only a short distance before interaction then D is small; if a long distance, D is large.

⁵ Like D , Σ_a and Σ_f are related to the distance a neutron is expected to travel before an interaction (absorption for Σ_a and fission for Σ_f) except in this case it is an inverse proportionality. For example, if a neutron is likely to be absorbed after traveling only a short distance, that means Σ_a is large.

⁶ B^2 is generically referred to as “buckling” or, since it is defined in terms of material properties, it is sometimes called “material buckling.”

Step #3: Separate the variables.

$$\frac{F_{rr}G}{FG} + \frac{1}{r} \frac{F_r G}{FG} + \frac{F G_{zz}}{FG} + B^2 \frac{F G}{FG} = 0$$

$$\underbrace{\frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} + \frac{G_{zz}}{G}}_{\text{function of } r} = -B^2 = -\lambda_1 - \lambda_2$$

$$\underbrace{\frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} + \lambda_1}_{\text{function of } r} = -\underbrace{\frac{G_{zz}}{G} - \lambda_2}_{\text{function of } z} = \lambda_3$$

Here we split up B^2 so part of it can be used in each equation.

The separated equations are:

$$\frac{F_{rr}}{F} + \frac{1}{r} \frac{F_r}{F} + \underbrace{\lambda_1 - \lambda_3}_{\nu^2} = 0$$

$$\frac{G_{zz}}{G} + \underbrace{\lambda_2 + \lambda_3}_{\kappa^2} = 0$$

Note that $\nu^2 + \kappa^2 = (\lambda_1 - \lambda_3) + (\lambda_2 + \lambda_3) = \lambda_1 + \lambda_2 = B^2$.

or, equivalently:

$$r^2 F_{rr} + r F_r + \nu^2 r^2 F = 0$$

$$G_{zz} + \kappa^2 G = 0$$

Step #4: Apply boundary conditions to determine non-trivial product solution(s):

We actually have homogeneous conditions on *every* boundary for this problem, so it does not matter which equation we start with. Also, some readers may have noticed, by the way we defined the separation constants— ν^2 and κ^2 —we have also quietly implied that they will both be *positive*. With these boundary conditions, this is indeed what we would have found to be the case anyway.

IN THE z -DIRECTION the general solution is:

$$G(z) = c_1 \cos \kappa z + c_2 \sin \kappa z$$

Applying the boundary condition at $z = 0$ gives us:

$$G_z(0) = -\kappa c_1 \sin 0 + \kappa c_2 \cos 0 = 0$$

$$\Rightarrow c_2 = 0$$

Applying the boundary condition at $z = H/2$ gives us:

$$G(H/2) = c_1 \cos \kappa \frac{H}{2} = 0$$

$$\Rightarrow \kappa \frac{H}{2} = \frac{n\pi}{2}, \quad n = 1, 3, 5, \dots$$

$$\Rightarrow \kappa = \frac{n\pi}{H}, \quad n = 1, 3, 5, \dots$$

There are an infinite number of values of κ that allow non-trivial solutions to $G(z)$ but, it turns out, *only one* of them—corresponding to $n = 1$ and $\kappa = \pi/H$ —is admissible. This is because the neutron flux must be non-negative. If the higher eigenmodes were allowed, then $G(z)$ would become negative in portions of the domain and thus $\phi(r, z)$ would become negative.⁷ Thus $G(z) = c_1 \cos \frac{\pi z}{H}$.

IN THE r -DIRECTION the general solution is:

$$F(r) = c_3 J_0(\nu r) + c_4 Y_0(\nu r)$$

Since $Y_0(\nu r)$ diverges to negative infinity as $r \rightarrow 0$, we must set $c_4 = 0$. The boundary condition at $r = R$ gives us:

$$\begin{aligned} F(c) &= c_3 J_0(\nu R) = 0 \\ \Rightarrow \nu &= \frac{k_{0,n}}{R} \end{aligned}$$

where $k_{0,n}$ is the n^{th} root of J_0 . Recalling from previous experience with Bessel functions of the first kind of order zero, J_0 oscillates infinitely many times so there are infinitely many roots. As was the case in the z -direction, we really are only interested in the first eigenmode $\nu_1 \approx 2.405/R$. This is because all higher eigenmodes would result in $F(r)$ being negative in some parts of the domain, which owing to the nature of $\phi(r, z)$, we cannot allow.

PUTTING TOGETHER THE results from the z - and r -directions gives us the product solution as shown in Equation 114.

$$\phi(r, z) = A J_0\left(\frac{2.405r}{R}\right) \cos\left(\frac{\pi z}{H}\right) \quad (114)$$

Step #5: Apply the remaining boundary conditions to determine the unknown coefficients.

On the plus-side, we only have one unknown constant remaining; on the minus-side, we actually do not have any more boundary conditions to apply to determine the unknown constant. It turns out that a problem of this type—it is not a wave equation, heat equation, or Laplace equation but a so-called *eigenvalue equation*⁸—was destined to have this problem from the start. We can solve for the flux shape, but we cannot nail down its magnitude. Luckily, we have a different way of specifying the unknown constant.

Let us assume that the purpose of this nuclear reactor is to create power. Each fission event releases a tiny amount of heat—approximately 3.2×10^{-11} Joules,⁹ a quantity we denote E_R . The *rate* of fissions (R_f) occurring in the reactor is proportional to the flux—specifically

⁷ We will dismiss as unworkable the assumption that $F(r)$ and $G(z)$ could somehow be contrived to be of the same sign everywhere in the domain and thereby get around this conclusion.

We combine both constants from $G(z)$ and $F(r)$ into A .

⁸ So-called because one may re-arrange the governing equation:

$$\nabla^2 \phi + \frac{B^2}{k} \phi = 0$$

to

$$\nabla^2 \phi = -\frac{B^2}{k} \phi$$

where, in this form, the term $1/k$ has been added and represents the *eigenvalue*. A more complete description is relegated to a course in reactor physics.

⁹ This number is specific for fission induced by a thermal neutron incident upon Uranium-235 but the recoverable energy released for other fission reactions is similar.

$R_f = \Sigma_f \phi(r, z)$. If we integrate the fission rate over the volume of the core, we get the total fission rate; multiplying by the energy released per fission, we get the total core power. Let us take the total reactor power to be a known parameter: P . Converting the words of this paragraph into math gives us:

$$P = E_R \Sigma_f 2\pi \underbrace{\int_{-H/2}^{H/2} \int_0^R A J_0\left(\frac{2.405r}{R}\right) \cos \frac{\pi z}{H} r dr dz}_{\int \int_V \phi dV} \quad (115)$$

Therefore

$$A = \frac{P}{E_R \Sigma_f 2\pi \int \int_V \phi dV} \quad (116)$$

In summary, the solution of the neutron diffusion equation in a finite, homogeneous, bare cylindrical reactor is given by Equation 114 where the constant is determined by specifying the reactor thermal power and evaluating Equation 116.

A Note on Buckling

In order to wrap up all of the details about solving the neutron diffusion equation for a finite, bare, homogeneous cylinder, we should take a moment to consider what happened to B^2 . As you may recall, during the separation of variables process, we parsed out pieces of B^2 to the equation both for the z -component and the r -component. It turned out that B^2 , which you can easily verify is equal to $v^2 + \kappa^2$, was found to be numerically equal to $\left(\frac{2.405}{R}\right)^2 + \left(\frac{\pi}{H}\right)^2$. Even though the initial definition of B^2 in the boundary value problem statement was based on Σ_a , Σ_f , and D —all of which are *material properties*—we determined in the separation of variables process that B^2 must be a numerical value related to the *geometric properties* of the problem. We sometimes denote B^2 as B_g^2 and call it *geometric buckling* whereas the quantity $\frac{v\Sigma_f - \Sigma_a}{D}$ is more often called the “material buckling.” For a critical—i.e. steady state—reactor, the two are equal.

What does this observation tell you? For one thing, it means that if either R or H is very small—if the reactor is like a thin rod or if it is flat like a pancake—the geometric buckling will be large. Thus for a reactor of this geometry to become critical, materials must be loaded in the core that increase Σ_f relative to Σ_a . The main way this can be done is to increase the concentration of fissile isotopes like ^{235}U or ^{239}Pu —an undertaking that is possible but expensive and, for sufficiently high fuel enrichment, subject to regulatory hurdles. On the other hand, a reactor with R and H both larger has lower

buckling. These considerations should be kept in mind when you are specifying the geometry of a reactor that you are designing.

Assignment #12

1. Find the steady-state temperature $u(r, z)$ in a cylinder if the boundary conditions are:

$$\begin{aligned} u(2, z) &= 0, \quad 0 < z < 4 \\ u(r, 0) &= u_0, \quad 0 < r < 2 \end{aligned}$$

Hint: For the differential equation $G_{zz} - \alpha^2 G = 0$, the solution for $\alpha > 0$ can be expressed as $G(z) = c_1 \cosh \alpha z + c_2 \sinh \alpha z$, but it can also be expressed as: $G(z) = c_1 \cosh [\alpha(k - z)] + c_2 \sinh [\alpha(k - z)]$ for any constant k —a fact that you should confirm for yourself. This is equivalent to shifting the location of the z -origin by k units. For this problem, it is very useful to use the latter form with $k = 4$.

2. Find the steady-state temperature $u(r, z)$ in a finite cylinder defined by $0 \leq r \leq 1$, $0 \leq z \leq 1$ if the boundary conditions are given as:

$$\begin{aligned} u(1, z) &= z, \quad 0 < z < 1 \\ u_z(r, 0) &= 0, \quad 0 < r < 1 \\ u_z(r, 1) &= 0, \quad 0 < r < 1 \end{aligned}$$

3. The temperature in a circular plate of radius c is determined from the boundary value problem:

$$\alpha^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} \right) = \frac{\partial u}{\partial t}, \quad 0 < r < c, \quad t > 0$$

$$u(c, t) = 0, \quad t > 0$$

$$u(r, 0) = f(r), \quad 0 < r < c$$

(a) Solve for $u(r, t)$.

(b) Implement your solution in MATLAB using the following parameter values: $\alpha^2 = 0.1$, $c = 2$, and $f(r)$ defined as follows:

$$f(r) = \begin{cases} 1+r, & 0 < r \leq 1 \\ 0, & 1 < r \leq 2 \end{cases}$$

and carry out the following tasks:

- i. Plot $u(r, 0)$ using $N = 15$ modes.
- ii. State the value to which $u(1, 0)$ converges and explain why.
- iii. Briefly describe what the final steady-state solution looks like.

4. A circular plate is a composite of two different materials in the form of concentric circles. The temperature $u(r, t)$ in the plate is determined from the boundary value problem:

$$\text{Governing Equation : } \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} = \frac{\partial u}{\partial t}, \quad 0 < r < 2, \quad t > 0$$

$$\text{Boundary Condition : } u(2, t) = 100, \quad t > 0$$

$$\text{Initial Condition : } u(r, 0) = \begin{cases} 200, & 0 < r < 1 \\ 100, & 1 < r < 2 \end{cases}$$

Use the substitution: $u(r, t) = v(r, t) + \psi(r)$ to solve the boundary value problem.

Lecture 34 - Laplace's Equation in Spherical Coordinates

Objectives

- Solve Laplace's equation in spherical coordinates.
- Introduce spherical harmonics.

Boundary Value Problem

In this lecture we will consider the problem of steady-state temperature in a sphere. We will define our spherical coordinates as shown in Figure 57. The relationship between x , y , z and r , θ , and ϕ are shown in the margin.

Laplace's equation could be stated generically enough as $\nabla^2 u = 0$, of course, but we need to adapt the definition of the Laplacian operator for spherical coordinates. This is shown in Equation 117.

$$\frac{\partial^2 u}{\partial r^2} + \frac{2}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2 \sin \theta^2} \frac{\partial^2 u}{\partial \phi^2} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} + \frac{\cot \theta}{r^2} \frac{\partial u}{\partial \theta} = 0 \quad (117)$$

In order to make the complexity a little more manageable, we will assume a boundary condition that is only a function of θ :

$$u(c, \theta) = f(\theta)$$

Thus the solution will only be a function of r and θ and all derivatives of u with respect to ϕ can be eliminated from the Laplacian. The governing equation will therefore be:

$$\frac{\partial^2 u}{\partial r^2} + \frac{2}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} + \frac{\cot \theta}{r^2} \frac{\partial u}{\partial \theta} = 0 \quad (118)$$

Despite all appearances, Equation 118 is a linear, homogeneous, second-order boundary value problem so we will use separation of variables as usual.

Step #1: Assume a product solution.

$$u(r, \theta) = F(r)G(\theta)$$

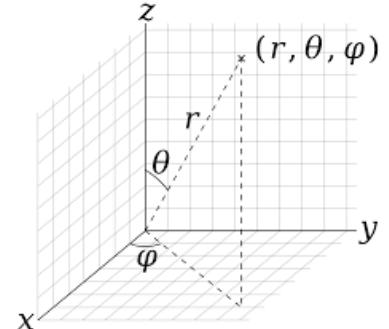


Figure 57: Spherical coordinate system.

$$\begin{aligned} x &= r \sin \theta \cos \phi & 0 < r < c \\ y &= r \sin \theta \sin \phi & 0 < \phi < 2\pi \\ z &= r \cos \theta & 0 < \theta < \pi \end{aligned}$$

Step #2: Insert the product solution into the governing equation.

$$\begin{aligned} \frac{\partial^2}{\partial r^2} [F(r)G(\theta)] + \frac{2}{r} \frac{\partial}{\partial r} [F(r)G(\theta)] + \dots \\ \frac{1}{r^2} \frac{\partial^2}{\partial \theta^2} [F(r)G(\theta)] + \frac{\cot \theta}{r^2} \frac{\partial}{\partial \theta} [F(r)G(\theta)] = 0 \end{aligned}$$

$$\begin{aligned} F_{rr}G + \frac{2}{r}F_rG + \frac{1}{r^2}FG_{\theta\theta} + \frac{\cot \theta}{r^2}FG_\theta = 0 \\ r^2F_{rr}G + 2rF_rG + FG_{\theta\theta} + \cot \theta FG_\theta = 0 \end{aligned}$$

Here we multiply through by r^2 to simplify the upcoming separation process.

Step #3: Separate the variables.

$$\begin{aligned} \frac{r^2F_{rr}G}{FG} + \frac{2rF_rG}{FG} + \frac{FG_{\theta\theta}}{FG} + \frac{\cot \theta FG_\theta}{FG} = 0 \\ \frac{r^2F_{rr}}{F} + \frac{2rF_r}{F} + \frac{G_{\theta\theta}}{G} + \frac{\cot \theta G_\theta}{G} = 0 \\ \frac{r^2F_{rr}}{F} + \frac{2rF_r}{F} = -\frac{G_{\theta\theta}}{G} - \frac{\cot \theta G_\theta}{G} = \lambda \end{aligned}$$

So the separated equations are:

$$\begin{aligned} r^2F_{rr} + 2rF_r - \lambda F = 0 \\ G_{\theta\theta} + \cot \theta G_\theta + \lambda G = 0 \end{aligned}$$

We can readily recognize the equation for $F(r)$ as a Cauchy-Euler equation. The equation for $G(\theta)$, on the other hand, looks like nothing we have ever seen.

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

We WILL FIND, eventually, that we can change the equation for $G(\theta)$ into something recognizable if we make the following transformation to the independent variable: $\cos \theta = x$. Since θ goes from 0 to π and $x = \cos \theta$, x will go from -1 to 1. Let us set out on the process to replace all derivatives of G with respect to θ to derivatives with respect to x . Using the chain rule:

$$\begin{aligned} G_\theta &= \frac{d}{dx}(G) \frac{dx}{d\theta} \\ &= -(1-x^2)^{1/2}G_x \end{aligned}$$

Here we use:

$$\begin{aligned} \frac{dx}{d\theta} &= -\sin \theta \\ &= -(1-x^2)^{1/2} \end{aligned}$$

An important detail here is that:
 $-\sin \theta = -(1-x^2)^{1/2}$ since
 $\cos^2 \theta + \sin^2 \theta = 1$ and therefore
 $\sin \theta = (1-\cos^2 \theta)^{1/2}$.

Repeating to find an expression for $G_{\theta\theta}$:

$$\begin{aligned} G_{\theta\theta} &= \frac{d}{dx}(G_\theta) \frac{dx}{d\theta} \\ &= \frac{d}{dx} \left[-(1-x^2)^{1/2} G_x \right] \left[-(1-x^2)^{1/2} \right] \\ &= \left[-\frac{1}{2}(1-x^2)^{-1/2}(-2x)G_x - (1-x^2)^{1/2}G_{xx} \right] \left[-(1-x^2)^{1/2} \right] \\ &= \frac{1}{2}(-2x)G_x + (1+x^2)G_{xx} \end{aligned}$$

Inserting these expressions into our equation for $G(\theta)$ gives us:

$$\overbrace{-xG_x + (1-x^2)G_{xx}}^{G_{\theta\theta}} + \underbrace{\frac{x}{1-x^2}^{1/2} \overbrace{[-1(1-x^2)^{1/2}G_x]}^{G_\theta}}_{\cot\theta} + \lambda G = 0$$

$$= \frac{\cos\theta}{\sin\theta}$$

which, upon simplification is:

$$(1-x^2)G_{xx} - 2xG_x + \lambda G = 0$$

Which is, at long last, Legendre's equation. Recall that Legendre's equation has polynomial solutions for $\lambda = n(n+1)$, $n = 0, 1, 2, \dots$ and these solutions are called Legendre polynomials, $P_n(x)$. Substituting now $x = \cos\theta$, we get the solution for $G(\theta)$:

$$G(\theta) = c_1 P_n(\cos\theta)$$

The functions $P_n(\cos\theta)$ are sometimes called *spherical harmonics*.

NOW THAT WE have solutions for $G(\theta)$ as well as our allowed eigenvalues, we circle back to solve $F(r)$. Following the usual procedure for Cauchy-Euler equations, we assume a solution of the form $F(r) = r^m$ and find values of m that satisfy the equation:

$$\begin{aligned} r^2 F_{rr} + 2rF_r - n(n+1)F &= 0 \\ r^2 \left[m(m-1)r^{m-2} \right] + 2rmr^{m-1} - n(n+1)r^m &= 0 \\ r^m \left[m^2 - m + 2m - n(n+1) \right] &= 0 \\ m^2 + m - n(n+1) &= 0 \\ (m-n)(m+(n+1)) &= 0 \\ \Rightarrow m &= n, -(n+1) \end{aligned}$$

Therefore the general solution for $F(r)$ is:

$$F(r) = c_2 r^n + c_3 r^{-(n+1)}$$

Functions that are solutions to Laplace's equation are called *harmonics*. Spherical harmonics solve Laplace's equation in spherical coordinates.

In order to ensure that $\lim_{r \rightarrow 0} F(r) < \infty$, we must set $c_3 = 0$. The product solution is given in Equation 119.

$$u(r, \theta) = \sum_{n=0}^{\infty} c_n r^n P_n(\cos \theta) \quad (119)$$

Step #5: Apply the remaining boundary conditions to determine the unknown coefficients.

The last boundary condition we have to apply is on the outside surface of the sphere:

$$u(R, \theta) = \sum_{n=0}^{\infty} c_n R^n P_n(\cos \theta) = f(\theta)$$

On the left of the last equality we have an infinite linear combination of orthogonal functions; on the right we have a function. We want to know the values of c_n so that they will be equal. How do we do this? We multiply both sides by an orthogonal function, and weight function, and integrate.

THIS CASE is a bit different than the last time we met Legendre polynomials, however, insofar as the argument for P_n is now $\cos \theta$ and not x . Suppose we pretended, temporarily, that we were dealing with $P_n(x)$; what would we do? The equation for c_n would look something like:

$$\begin{aligned} \int_{-1}^1 c_n R^n P_n(x)^2 dx &= \int_{-1}^1 f(x) P_n(x) dx \\ \Rightarrow c_n &= \frac{1}{R^n} \frac{\int_{-1}^1 f(x) P_n(x) dx}{\int_{-1}^1 P_n(x)^2 dx} \end{aligned}$$

But in our case, we have $P_n(\cos \theta)$; we need to change variables, again, to reflect $x = \cos \theta$, and $dx = -\sin \theta d\theta$. We must also make substitutions in the limits of integration: if $x = \cos \theta$, when $x = -1$, $\theta = \pi$; also $x = \cos \theta$ when $x = 1$, corresponds to $\theta = 0$. Making these substitutions into our expression gives us the formula for our coefficients in Equation 120.

$$c_n = \frac{\int_0^\pi f(\theta) P_n(\cos \theta) \sin \theta d\theta}{R^n \int_0^\pi P_n(\cos \theta)^2 \sin \theta d\theta} \quad (120)$$

MATLAB Implementation

In the code listings below we will construct the solution for $R = 2$ and $f(\theta)$ given by:

$$f(\theta) = \begin{cases} 10(\pi/2 - \theta), & 0 \leq \theta < \pi/2 \\ 0, & \pi/2 \leq \theta \leq \pi \end{cases}$$

Admittedly, this is the only boundary condition that we have applied so far for this problem. We latched on to the eigenvalues $\lambda = n(n + 1)$, for non-negative integer n , and did not explore what would happen if n is negative. In the interest of time and space in this lecture, I ask that we leave well-enough alone and defer that exploration for another day.

With the given substitutions, the formula for c_n becomes:

$$c_n = \frac{\int_\pi^0 f(\theta) P_n(\cos \theta) (-\sin \theta) d\theta}{R^n \int_\pi^0 P_n(\cos \theta)^2 (-\sin \theta) d\theta}$$

In Equation 120 we flipped the bounds of integration and removed the minus sign from $(-\sin \theta)$.

We start, as usual, by clearing out the workspace and setting problem parameters.

```
clear
clc
close 'all'
 $\% \%$  Set Parameters
R = 2;
N = 4;
f = @(theta) ex1(theta);
```

Next we construct the solution for the specified number of eigenfunctions.

```
 $\% \%$  Construct the Solution
c = nan(N,1);
u = @(r,theta) o; % initialize the series

% start for n = 0
n = 0;
co = integral(@(th) f(th).*...
    legendreP(n,cos(th)).*sin(th),0,pi)./...
    integral(@(th) (R^n).*...
    (legendreP(n,cos(th)).^2).*sin(th),0,pi);

u = @(r,theta) u(r,theta) + co*(r.^o).*legendreP(n,cos(theta));

for n = 1:N
    % get the next coefficient
    c(n) = integral(@(th) f(th).*...
        legendreP(n,cos(th)).*sin(th),0,pi)./...
        integral(@(th) (R^n).*...
        (legendreP(n,cos(th)).^2).*sin(th),0,pi);

    % update the approximation
    u = @(r,theta) u(r,theta) + ...
        c(n)*(r.^n).*legendreP(n,cos(theta));
end
```

Next we would like to visualize the results. At the time of this writing, MATLAB has limited capability for visualizing three-dimensional data. For the example, we will tabulate the solution on a regular mesh comprising a cube that contains the sphere of interest. The tabulated solution will be written to a VTK¹ data file that can be visualized with software tools such as ParaView.²

```
 $\% \%$  Process Result for Plotting
Nx = 50;
Xv = linspace(-R,R,Nx);
Yv = linspace(-R,R,Nx);
Zv = linspace(-R,R,Nx);
dx = Xv(2)-Xv(1); % need this for VTK file
[XX,YY,ZZ] = meshgrid(Xv,Yv,Zv);

RR = sqrt(XX.^2+YY.^2+ZZ.^2);
PP = acos(ZZ./RR);
UU = u(RR,PP);
% set region outside the sphere to nan
UU(RR>R) = nan;
```

¹ VTK stands for “Visualization Toolkit” and a VTK data file is one of several standard data formats used for storing scientific data as it is prepared for visualization.

² A. Henderson. ParaView Guide, A Parallel Visualization Application. Technical report, Kitware Inc., 2007

```

%% Write the data to a file
filename = 'solution.vtk';
dataname = 'U';
origin = [-R -R -R];
spacing = [dx dx dx];
save_scalarStructuredPoints3D_VTK_binary(filename ,...
    dataname,UU,origin ,spacing );

```

45
46
47
48
49
50
51
52

Lastly, let us show the local functions used to represent the boundary condition and to write the resulting data into a properly formatted VTK file.

```

%% Local functions
function u = ex1(theta)
[n,m] = size(theta);
u = nan(n,m);
ind_a = theta <=(pi/2);
ind_b = theta >(pi/2);
u(ind_a) = 10*((pi/2)-theta(ind_a));
u(ind_b) = 0;
end

function save_scalarStructuredPoints3D_VTK_binary(filename ,... ①
    dataname,data_set,origin,spacing)

[nx,ny,nz]=size(data_set);

% open the file
fid = fopen(filename , 'w');

% ASCII file header
fprintf(fid , '# vtk DataFile Version 3.0\n');
fprintf(fid , 'VTK from Matlab\n');
fprintf(fid , 'BINARY\n\n');
fprintf(fid , 'DATASET STRUCTURED_POINTS\n');
fprintf(fid , 'DIMENSIONS %d %d %d \n' ,nx,ny,nz);
fprintf(fid , 'ORIGIN %4.3f %4.3f %4.3f \n' ,...
    origin(1),origin(2),origin(3));
fprintf(fid , 'SPACING %4.3f %4.3f %4.3f \n' ,...
    spacing(1),spacing(2),spacing(3));
fprintf(fid , '\n');
fprintf(fid , 'POINT_DATA %d \n' ,nx*ny*nz);
fprintf(fid , strcat('SCALARS','\t',dataname,' float ','\n'));
fprintf(fid , 'LOOKUP_TABLE default \n');

% write the data
fwrite(fid , reshape(data_set,1,nx*ny*nz) , 'float' , 'b' );
% close the file
fclose(fid);

end

```

53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90

❶ This local function, as the name suggests, saves the scalar data—which is organized in a three-dimensional regular grid—into a binary data file. The ASCII file header is a specified format and is used to describe the structure of the data so that software like ParaView can be used to visualize the data. We refer to this as a “binary data file” since the actual data is stored in binary form. Alternatively the data could be stored in plain text but that would make the file much larger.

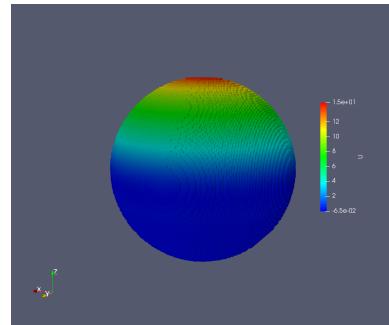


Figure 58: Plot of solution using ParaView.

The solution for this case is shown in Figure 58.

Lecture 35 - Non-homogeneous Problem in Spherical Coordinates

Objectives

- Solve the time-dependent heat equation in spherical coordinates.
- Provide another example illustrating how non-homogeneous boundary conditions can be treated.
- Show another MATLAB solution.

Non-homogeneous Heat Equation on a Sphere

Consider the time-dependent temperature within a unit sphere as described by the following boundary value problem:

$$\text{Governing Equation : } \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{2}{r} \frac{\partial u}{\partial r}, \quad 0 < r < 1, \quad t > 0$$

$$\text{Boundary Conditions : } u(1, t) = 100, \quad t > 0$$

$$\text{Initial Conditions : } u(x, 0) = 0, \quad 0 < r < 1$$

Notice that we have omitted portions of the Laplacian, in spherical coordinates, containing derivatives of ϕ and θ . Besides the usual uniformity of material properties, this is owing to the boundary conditions and initial conditions that are constants.¹ Thus we have considerably simplified the equation. Having said that you should also notice that the boundary condition at $r = 1$ is non-homogeneous. We will not be able to solve this problem using separation of variables without dealing with that boundary condition first.

READERS MAY RECALL from Lecture 27 that we were able to deal with (some types of) non-homogeneous terms in the governing equation and boundary conditions by assuming a solution of the form:
 $u(x, t) = v(x, t) + \psi(x)$. The boundary value problem that we derived

¹ We have omitted the thermal diffusivity, so you should assume $\alpha^2 = 1$.

for $\psi(x)$ absorbed all of the non-homogeneous terms leaving a homogeneous boundary value problem for $v(x, t)$ that we could solve using separation of variables. We will pursue a similar strategy in this lecture. What we will do is:

- Verify that $\frac{\partial^2 u}{\partial r^2} + \frac{2}{r} \frac{\partial u}{\partial r}$ from the governing equation, can be expressed as: $\frac{1}{r} \frac{\partial^2}{\partial r^2}(ru)$. This is easily verified:

$$\begin{aligned}\frac{1}{r} \frac{\partial^2}{\partial r^2}(ru) &= \frac{1}{r} \frac{\partial}{\partial r} \left[\frac{\partial}{\partial r}(ru) \right] \\ &= \frac{1}{r} \frac{\partial}{\partial r} [u + ru_r] \\ &= \frac{1}{r} [u_r + u_r + ru_{rr}] \\ &= u_{rr} + \frac{2}{r} u_r\end{aligned}$$

and

- Let $ru(r, t) = v(r, t) + \psi(r)$ where $\psi(r)$ will once again be used to absorb the nonhomogeneities. In this case we will also need to take care to only use solutions $u(r, t) = \frac{1}{r}v(r, t) + \frac{1}{r}\psi(r)$ that remain bounded as $r \rightarrow 0$.

LET US NOW restate the boundary value problem in terms of $ru(r, t) = v(r, t) + \psi(r)$. The governing equation becomes:

$$\begin{aligned}\frac{1}{r} \frac{\partial^2}{\partial r^2} [v(r, t) + \psi(r)] &= \frac{\partial}{\partial t} \left[\frac{1}{r}v(r, t) + \frac{1}{r}\psi(r) \right] \\ \frac{1}{r} [v_{rr} + \psi_{rr}] &= \frac{1}{r}v_t\end{aligned}$$

The boundary condition becomes:

$$\begin{aligned}u(1, t) &= 100 \\ \frac{1}{1}v(1, t) + \frac{1}{1}\psi(1) &= 100 \\ v(1, t) + \psi(1) &= 100\end{aligned}$$

The boundary condition in the new form is $\frac{1}{r}v(r, t)\Big|_{r=1} + \frac{1}{r}\psi(r)\Big|_{r=1}$

And the initial condition is:

$$\begin{aligned}u(r, 0) &= 0 \\ \frac{1}{r}v(r, 0) + \frac{1}{r}\psi(r) &= 0\end{aligned}$$

We will take the boundary value problem for $\psi(r)$ therefore to be:

$$\begin{aligned}\frac{1}{r}\psi_{rr} &= 0 \\ \Rightarrow \psi_{rr} &= 0 \\ \psi(1) &= 100\end{aligned}$$

and the boundary value problem for $v(r, t)$ to be:

$$\begin{array}{l} \text{Governing} \\ \text{Equation} \end{array} : \quad v_{rr} = v_t, \quad 0 < r < 1, \quad t > 0$$

$$\begin{array}{l} \text{Boundary} \\ \text{Conditions} \end{array} : \quad v(1, t) = 0, \quad t > 0$$

$$\begin{array}{l} \text{Initial} \\ \text{Conditions} \end{array} : \quad v(r, 0) = -\psi(r), \quad 0 < r < 1$$

LET US FIRST solve for $\psi(r)$:

$$\begin{aligned} \psi(r) &= 0 \\ \psi(1) &= 100 \end{aligned}$$

The general solution is $\psi(r) = c_1 r + c_2$. Applying the boundary condition we get: $\psi(1) = c_1(1) + c_2 = 100$. There are infinitely many values for c_1 and c_2 that could satisfy this condition but we need to remember that $\frac{1}{r}\psi(r)$ must remain bounded as $r \rightarrow 0$. Thus we will choose to set $c_2 = 0$ and $c_1 = 100$ so that:

$$\psi(r) = 100r \quad (121)$$

NOW WE WILL TURN to the solution of the boundary value problem for $v(r, t)$ using separation of variables.

Step #1: Assume a product solution.

$$v(r, t) = F(r)G(t)$$

Step #2: Insert the product solution into the governing equation.

$$\begin{aligned} \frac{\partial^2}{\partial r^2} [F(r)G(t)] &= \frac{\partial}{\partial t} [F(r)G(t)] \\ F_{rr}G &= FG_t \end{aligned}$$

Step #3: Separate variables.

$$\begin{aligned} \frac{F_{rr}G}{FG} &= \frac{FG_t}{FG} \\ \frac{F_{rr}}{F} &= \frac{G_t}{G} = -\lambda \end{aligned}$$

So the separated equations are:

$$\begin{aligned} F_{rr} + \lambda F &= 0 \\ G_t + \lambda G &= 0 \end{aligned}$$

Step #4: Apply boundary conditions to determine non-trivial product solution(s).

For this problem, our boundary conditions are that $v(1, t) = 0$ and that $\frac{1}{r}v(r, t)$ should remain finite as $r \rightarrow 0$. Rather than go through the analysis exhaustively, for this lecture we will claim that $\lambda > 0$ and set $\lambda = \alpha^2$, $\alpha > 0$. This means that:

$$\begin{aligned} F(r) &= c_3 \cos \alpha r + c_4 \sin \alpha r \\ G(t) &= c_5 e^{-\alpha^2 t} \end{aligned}$$

In order to satisfy the requirement as $r \rightarrow 0$, we will set $c_3 = 0$ since $\frac{1}{r} \cos \alpha r$ diverges as $r \rightarrow 0$. As for the rest of $F(r)$:

$$\begin{aligned} F(1) &= c_4 \sin \alpha = 0 \\ \Rightarrow \sin \alpha &= 0 \\ \Rightarrow \alpha &= n\pi, \quad n = 1, 2, 3, \dots \end{aligned}$$

So the product solution is:

$$v(r, t) = \sum_{n=1}^{\infty} c_n \sin(\alpha_n r) e^{-\alpha_n^2 t} \quad (122)$$

Step #5: Satisfy the initial condition.

$$\begin{aligned} v(r, 0) &= \sum_{n=1}^{\infty} c_n \sin(\alpha_n r)(1) = -\psi(r) \\ \sum_{n=1}^{\infty} \sin(\alpha_n r) &= -100r \end{aligned}$$

On the left we have an infinite linear combination of eigenfunctions, on the right we have a function. We need to determine the values of the constants c_n so that the two are equal. How do we do this? We will multiply both sides by an orthogonal function and integrate. The result is:

$$c_n = \frac{\int_0^1 -100r \sin(\alpha_n r) dr}{\int_0^1 \sin(\alpha_n r)^2 dr} \quad (123)$$

THE LAST STEP will be to combine the solutions for $\psi(r)$ and $v(r, t)$ to form $u(r, t)$:

$$\begin{aligned} ru(r, t) &= v(r, t) + \psi(r) \\ u(r, t) &= \frac{1}{r} [v(r, t) + \psi(r)] \\ u(r, t) &= \underbrace{\frac{1}{r} \sum_{n=1}^{\infty} c_n \sin(\alpha_n r) e^{-\alpha_n^2 t}}_{\text{transient}} + \underbrace{100}_{\text{steady state}} \end{aligned}$$

where the formula for the coefficients is given in Equation 123.

Interested readers should show that for $\lambda < 0$ or $\lambda = 0$ there are no non-trivial solutions that will both satisfy the boundary condition and not diverge as $r \rightarrow 0$.

As in Lecture 27, we find that $v(x, t)$ corresponds to the transient solution, and $\psi(r)$ is the steady-state solution.

MATLAB Implementation

The code to construct this solution is straight-forward and presented in its entirety in the listing below.

```

1 clear
2 clc
3 close 'all'

4 %>> Parameters
5 N = 50;

6 c = 1; % radius of sphere
7 Psi = @(r) 100.*r;
8 R = @(r,n) sin(n.*pi.*r);
9 T = @(t,n) exp(-((n.*pi).^2).*t);

10 V = @(r,t) 0; % initialize my approximation for V
11
12 for n = 1:N
13     % compute coefficient
14     cn = 2*integral(@(r) -Psi(r).*R(n,r),0,c);
15
16     % update solution to V
17     V = @(r,t) V(r,t) + cn.*R(r,n).*T(t,n);
18 end
19
20 % construct solution U
21 U = @(r,t) (1./r).*(V(r,t)+Psi(r));
22
23
24

```

Plots of the solution at various time, t , are presented below.

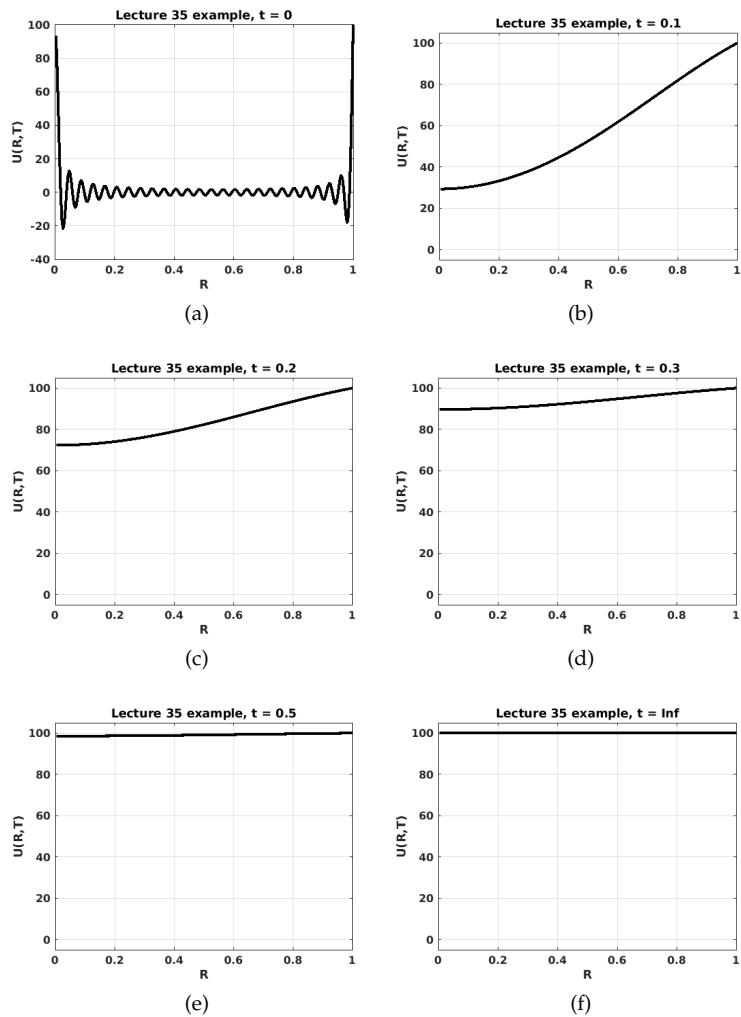


Figure 59: Plots solution at various times.

Assignment #13

- Find the steady-state temperature $u(r, \theta)$ in a sphere described by the boundary value problem below.

Governing Equation : $u_{rr} + \frac{2}{r}u_r + \frac{1}{r^2}u_{\theta\theta} + \frac{\cot\theta}{r^2}u_\theta = 0, \quad 0 < r < c, \quad 0 < \theta < \pi$

Boundary Condition : $u(c, \theta) = \begin{cases} 50, & 0 < \theta < \pi/2 \\ 0, & \pi/2 < \theta < \pi \end{cases}$

This problem was solved in Lecture 34. You need not repeat the analysis except to the extent that you want to remind yourself of the details. Implement the solution in MATLAB where the radius of the sphere is $c = 1$. Use MATLAB to print out enough coefficients of the first four non-zero terms to the series solution.

- Solve the boundary value problem involving spherical vibrations:

Governing Equation : $\alpha^2 \left(\frac{\partial^2 u}{\partial r^2} + \frac{2}{r} \frac{\partial u}{\partial r} \right) = \frac{\partial^2 u}{\partial t^2}, \quad 0 < r < c, \quad t > 0$

Boundary Conditions : $u(c, t) = 0, \quad t > 0$

Initial Conditions : $u(r, 0) = f(r), \quad u_t(r, 0) = g(r), \quad 0 < r < c$

Hint: Write the left side of the partial differential equation as $\alpha^2 \frac{1}{r} \frac{\partial^2}{\partial r^2}(ru)$. Let $v(r, t) = ru(r, t)$.

Review Problems #3

List of topics

1. Boundary Value Problems in Rectangular Coordinates.
 - (a) Laplace's Equation
 - (b) Non-homogeneous Boundary Value Problem
 - (c) Orthogonal Series Expansions
 - (d) Fourier Series in Two Variables
2. Boundary Value Problems in Polar, Cylindrical and Spherical Coordinates.
 - (a) Problems in Polar Coordinates
 - (b) Problems in Cylindrical Coordinates
 - (c) The Neutron Diffusion Equation in Cylindrical Coordinates
 - (d) Problems in Spherical Coordinates

Review Problems

1. Solve the following boundary value problem:

Governing Equation : $\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}, \quad 0 < x < 1, \quad t > 0$

Boundary Condition : $u(0, t) = u_0, \quad u_x(1, t) = -u(1, t) + u_1, \quad t > 0$

Initial Condition : $u(x, 0) = u_0, \quad 0 < x < 1$

where u_0 and u_1 are constants.

2. Solve the boundary value problem below based on Laplace's equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0, \quad 0 < x < a, \quad 0 < y < b, \quad 0 < z < c$$

for the unit cube ($a = b = c = 1$) with top ($z = 1$) and bottom ($z = 0$) kept at constant temperatures u_0 and $-u_0$ respectively, and the remaining temperatures kept at $u = 0$.

3. Find the steady-state temperature $u(r, \theta)$ in a semicircular plate of radius 1 if the boundary conditions are given as:

$$\begin{aligned} u(1, \theta) &= u_0(\pi\theta - \theta^2), \quad 0 < \theta < \pi \\ u(r, 0) &= 0, \quad u(r, \pi) = 0, \quad 0 < r < 1 \end{aligned}$$

4. Find the steady-state temperature $u(r, z)$ in a cylinder if the lateral side ($r = 2$) is kept at temperature $u = 0$; the top ($z = 4$) is kept at $u = 50$, and the base, ($z = 0$) is insulated.

Part VI

Numerical Methods

Introduction

Lecture 1 - Course Introduction and Numeric Preliminaries

Objectives

The objectives of this lecture are:

- Introduce the course objectives.
- Describe how numbers are represented on computers.
- Outline sources of errors in numerical methods relative to analytical methods.

Course Introduction

This course is intended to be an introduction and overview of numerical methods. The target audience comprises undergraduate students of engineering who have already taken a full sequence of calculus and differential equations courses. Some students may also have taken the analytical methods course described earlier in this book. All students are expected to have some proficiency in the MATLAB programming environment.

Question: What are numerical methods?

Answer: Numerical methods (or numerical analysis) is the study of *algorithms* for the problems of *continuous* mathematics.

Objectives

THE OBJECTIVES for this course are as follows:

1. Students will understand the fundamentals of numerical methods with an emphasis on the most essential algorithms and methods.
2. Students will have enhanced their scientific computing skills and further developed their proficiency in using the MATLAB environment to implement algorithms and have learned to critically evaluate their results.

3. Students will understand the fundamentals of the finite element method (FEM) and have attained an introductory-level proficiency in using the COMSOL software package to carry out a multi-physics analysis of a relevant physical model.
4. Students will have developed the ability to formulate a problem of interest, apply numerical methods and computational tools to analyze the problem, and communicate pertinent results to others.

Course Topics

The specific topics we will cover include:

1. Methods for solving non-linear equations
 - (a) Bisection method
 - (b) Newton's method
 - (c) Secant method
2. Methods for solving linear equations
 - (a) Gauss elimination and related methods like LU- and Cholesky factorization
 - (b) Iterative solution methods for sparse linear systems of equations¹
3. Curve fitting
 - (a) Least squares algorithms
 - (b) Curve fitting with Lagrange polynomials
4. Numeric differentiation
 - (a) Finite difference methods
 - (b) Lagrange polynomials
5. Numeric integration
 - (a) A variety of Newton-Cotes methods
 - (b) Gauss Quadrature
6. Initial- and Boundary-value problems ²
 - (a) Runge-Kutta methods for initial value problems³
 - (b) Shooting method for boundary value problems
 - (c) Finite element method⁴

Note: While this manuscript includes a two-lecture introduction to the Finite Element Method, the COMSOL workshops that are conducted as part of this course will not be presented.

¹ We will also discuss some preconditioners.

² This section will prominently include MATLAB built-in methods; particularly for boundary value problems.

³ We will not extensively cover either explicit or implicit multi-step methods, giving preference to the wide variety of very effective RK methods.

⁴ There will be a simple MATLAB demonstration with application to one-dimensional boundary value problems. The majority of FEM coverage will be related to the use of COMSOL.

Representation of Numbers on a Computer

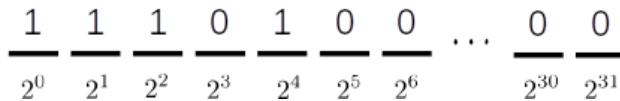
Every engineer who uses computational tools in their work should have a basic understanding of how numbers are represented on a computer. The essential take-aways from this section are:

1. Since the computer is a finite machine, only a finite set of numbers can be exactly represented on the computer. All other numbers are approximated.
2. Computers store integers and non-integers differently and the limits pertaining to what numbers can be represented or how exactly they are represented are also different.
3. The fact that numbers, in general, are represented inexactly on the computer has an impact on how algorithms are developed.

Integers

Integers are represented exactly on a computer, but only a finite subset of all integers can be represented. There are a number of integer types that are supported by common computer architectures and language compilers. For the purposes of this class we will focus on two types: *unsigned integers* and *signed integers*. To further focus the discussion we will only consider 32-bit representations of signed and unsigned integers.

PERHAPS THE EASIEST integer representation to understand is the 32-bit unsigned integer.⁵ In this format 32 binary digits corresponding to 2^0 to 2^{31} are stored in memory. Numbers in binary work the same way our normal decimal number work: just base 2 instead of base 10. For example, the 32-bit unsigned integer representation of the number 23 is shown in Figure 6o.



With 32-bit unsigned integers the computer can exactly represent all integers between 0 and $2^{32} - 1$. Negative integers and integers equal to or greater than 2^{31} are not represented at all.

THERE ARE FEW applications for which *signed integers* will be used for this course, but they are, naturally, important. Perhaps the most obvious way that a computer might represent signed integers would be to just use the same format as unsigned integers except to reserve

Some of the integer data types specified by the C++ language standard include:

1. `signed char` (8 bits)
2. `short int` (16 bits)
3. `int` (at least 16 bits – usually 32 bits)
4. `long int` (at least 32 bits)
5. `long long int` (at least 64 bits)

Each of these categories includes a signed and unsigned variant. Even within these categories there is fuzziness—e.g., “*at least 16 bits*”—that allows for variations between different compiler implementations and computer hardware (e.g. Intel vs. AMD CPU).

Note: The word *bit* is taken to be synonymous with the words *binary digit*.

⁵ You can create a 32-bit unsigned integer in MATLAB by typing:

`x = uint32(1994)`

Variables with other data types can be constructed using similar syntax; see the MATLAB documentation for details.

Figure 6o: The number 23 in unsigned integer format.

one bit for the sign. This is not, however, the way it is normally done. For one thing, this approach results in there being two different bit-patterns for zero. This might seem like a trivial inconvenience but it is not the sort of thing that passes without notice in computer engineering circles. Another, perhaps more significant, issue with this approach is that special logic would be needed when adding or subtracting a mixture of positive and negative integers. (i.e. you would not be able to use the same circuitry on the microprocessor for adding positive and negative numbers together as you would for adding two positive numbers.)

The format that is normally used today for representing signed integers is called *two's complement*. If you want to write a positive number in two's complement format, you do the same thing as you would normally do for an unsigned integer.⁶ If you need to write a negative number, you start by expressing the positive number, then you *take the complement of each binary digit* and, when you are done with that, you *add one*. A demonstration of this number representation, shortened to 5-bits to make it more compact, is shown in Figure 61.

This representation has the advantages that: a) there is only a single representation for zero; and b) the same hardware is used for both addition and subtraction (i.e. to subtract you simply add a negative number.)

Floating Point Numbers

In general, binary floating point numbers are given in the form shown in Equation 124:

$$1. \underbrace{\text{fff...f}}_{\text{mantissa}} \times 2^{\underbrace{\text{eee...e}}_{\text{exponent}}} \quad (124)$$

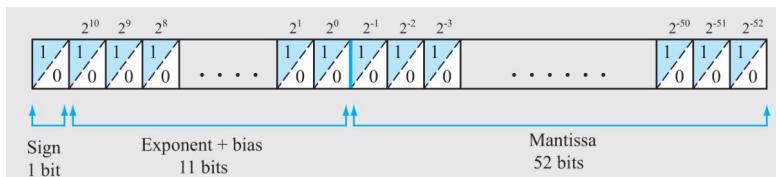
where the number of digits available are divided between the fraction, or *mantissa*, and *exponent*. In any finite machine, there will be a limit to the number of total digits available. Thus an engineering decision needs to be made to determine how many digits will be allocated to the mantissa and how many for the exponent. If you have more digits in the mantissa, you will be able to represent numbers that are closer together—the *precision* of your representation will increase. If you allocate more digits to the exponent, you will be able to represent numbers that are larger (for large positive exponents) or smaller (large negative exponents)—the *range* of your representation will be wider. The implications of these decisions should be clear: if you devote fewer bits to the mantissa, there will be more rounding errors in your calculations as real numbers are mapped, in some way,

⁶ Except, as it turns out, the largest positive 32-bit signed integer will only half as large as the corresponding largest signed integer.

$\begin{array}{r} 1 & 1 & 1 & 0 & 1 \\ \hline \end{array}$	Start with 23 in binary.
$\begin{array}{r} 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$	Flip the digits.
$\begin{array}{r} 1 & 0 & 0 & 1 & 0 \\ \hline \end{array}$	Add one to get -23.
$\begin{array}{r} & & & & \\ \hline & & & & \end{array}$	"carried" ones
$\begin{array}{r} 1 & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline & & & & \end{array}$	23
$\begin{array}{r} 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$	+ -23
$\begin{array}{r} & & & & \\ \hline & & & & \end{array}$	0

Figure 61: Demonstration of two's-complement: adding 23 and -23 in 5-bit representation.

to the best available floating point representation; if you devote fewer bits to the exponent, really large numbers and really small numbers will not be represented at all. In earlier days of computing, different computer vendors made different decisions as to how floating point numbers would be represented.⁷ This caused problems when scientists tried to run the same code on different computers and got different results. In 1985, the IEEE-754 standard was approved and, since then, has been adopted by essentially all computer manufacturers. As a result, the menagerie of floating point formats and implementations have been tamed. Scientists and engineers could run their codes on different machines and expect to get the same results.



The IEEE-754 standard provides specification of several floating point number formats. For this class, we will be concerned primarily with the *double precision* floating point format which is illustrated in Figure 62. This format uses a single sign bit (1 for negative, 0 for positive), 11 bits for the exponent which is encoded as an unsigned integer with bias,⁸ and 52 bits for the mantissa.⁹ In the double precision format, the smallest positive number that can be represented is 2^{-1022} which is approximately equal to 2.2×10^{-308} . The smallest interval between numbers that can be represented is called *machine epsilon*. The size of this limit is driven by the number of bits in the mantissa. For IEEE-754 double precision floating point numbers this is approximately 2.22×10^{-16} .¹⁰

THE PROCEDURE TO encode real numbers in the double precision floating point format will be illustrated with an example.

Example: Write the number -10.5 using the IEEE-754 double precision format.

Step #1: Determine the mantissa and the exponent. To do this, we normalize the number by dividing by 2^e where e is the largest power of 2 that is *less* than the magnitude of the number you are encoding.

In this case, $e = 3$ since 10.5 is greater than 2^3 but less than 2^4 .

$$\begin{aligned} \frac{-10.5}{2^3} &= -1.3125 \\ \Rightarrow -10.5 &= -1.3125 \times 2^3 \end{aligned}$$

⁷ Cleve Moler. Floating Point Numbers, 2014. URL <https://blogs.mathworks.com/cleve/2014/07/07/floating-point-numbers/>. Accessed on July 12, 2023

Figure 62: IEEE-754 double precision number format.

⁸ For double precision numbers, the bias is 1023.

⁹ **Note:** The number 1 shown in Equation 124 is not stored but is implicitly included in all *normalized* floating point numbers. This is done so that all numbers will have a *unique* floating point representation.

MATLAB includes built-in functions to report the smallest and largest representable floating point numbers. The functions `realmin(precision)` and `realmax(precision)` will return the smallest or largest positive floating point number for "single" or "double" precision floating point precision.

In MATLAB, machine epsilon is provided with the function: `eps(precision)` for "single" and "double" precision.

¹⁰ Sometimes computational scientists refer to this as "16 digits of precision."

Therefore the mantissa will need to encode 0.3215, and the exponent will need to encode 3 plus the bias of 1023, or 1026.

Step #2: Set the sign bit. As the number is negative, the sign bit is: 1.

Step #3: Calculate the 11 exponent bits.

$$\begin{aligned} 1026 &= 1024 + 2 \\ &= 1 \times 2^{10} + 1 \times 2^1 \end{aligned}$$

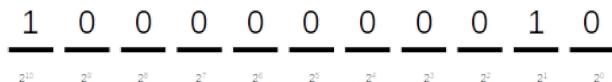


Figure 63: Exponent: $3+1023 = 1026$ in 11-bit binary.

Step #4: Calculate the 52 mantissa bits to represent 0.3125. The result is shown in Figure 64.

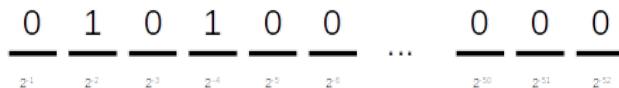


Figure 64: Mantissa: $0.25 + 0.0625 = 0.3125$.

Where the calculations might be done as shown below:

$$0.3125 - 0 \times 2^{-1} = 0.3125$$

$$0.3125 - 1 \times 2^{-2} = 0.0625$$

$$0.0625 - 0 \times 2^{-3} = 0.0625$$

$$0.0625 - 1 \times 2^{-4} = 0$$

and all other entries are, of course, zero.

Sources of Error

When using numerical methods there are several sources of error that you, as an engineer, should be aware of. We have just discussed the details of double precision numbers so *round-off error* is at the top of our mind. Real numbers are not represented exactly on a computer but instead are represented as floating point numbers. The floating point number *closest* to the real number gets used in its place. Error due to this round-off accumulates with mathematical operations. Quantitative analysis of the accumulation of round-off-error is a tedious (albeit important) business that we will avoid in this class. Nonetheless, engineers need to be aware that it is happening and, in a few instances, we will take steps to mitigate this build-up of error.

The second source of error that we will mention here is something that should be familiar to students who have taken the course in analytical methods: *truncation error*. In the analytical methods course we suffered from truncation error every time we shortened our infinite series solution to a finite number of terms. We reduced this error by increasing the number of terms we retained, but there was no way to make it go away completely. In this course we will see further instances of truncation error in most every algorithm we implement. Derivatives and integrals that could, in principle, be done exactly, will be done approximately as we truncate the Taylor series expansion or limit the order of polynomials used to represent the exact derivative or integral. We can reduce the magnitude of these errors, but we cannot make them go away entirely.

The third source of error mentioned here is *modeling error*. Modeling errors are reflected in the differences between observed physical phenomena and the corresponding mathematical model of the phenomena. There are a couple of reasons for these differences and these include:

1. Linearization of non-linear processes. Instances where you may already have done this, in your heat transfer or fluid dynamics class, include:
 - (a) Assumption of constant thermal conductivity for heat conduction problems. This assumption linearizes the heat equation and, for simple geometry, renders the problem amenable to analytical methods. This assumption is retained for many numerical methods when solving the heat equation on more complex geometry. The modeling error resulting from linearization remains.
 - (b) Assumption of constant drag coefficient for external flow calculations. Once again, this assumption simplifies the calculations at the cost of model fidelity.
2. De-coupling of physical processes. Nature does not specialize its behavior by academic subjects. Consequently the air flowing over the control surfaces of a fighter jet is both a fluid dynamics and structural dynamics problem when the wing responds to forces imposed by the air. Water flowing through a pressurized water reactor core is heated by forced convection and radiation that you might calculate in heat transfer class, but the resulting change in water properties also affect the neutron economy and power production rate that you might calculate in reactor physics which, in turn, will have further effect on the heat transfer problem. We de-couple these processes so the analysis is more manageable but as a result the solution we arrive at is in error.

Scientists and engineers need to be aware of all of these sources of errors and take steps to mitigate them where possible.

Lecture 2 - Linear Algebra Background

Objective

The objective of this lecture is to:

- Provide an overview of essential linear algebra concepts.

Definitions

In this course we will regularly make use of vectors and matrices. These mathematical objects are widely used and probably familiar to readers from previous courses. Nonetheless we will review some concepts so that we can all go forward with a common understanding or, at least, a common vocabulary. We begin with the following definitions:

Definition 16 (Vector)

A vector is row or column of numbers. From our perspective, the significance of such a row vector or column vector is that the numbers so presented are a discrete representation of a function.

Definition 17 (Matrix)

In many contexts, it is suitable to view a matrix as: any rectangular array of numbers.¹ For this class, and as a computational engineer, a more specific definition of a matrix that is often appropriate would be: a discrete representation of a differential operator.

These are intended to be operational definitions; not technically complete or correct but adequate for our present purposes. More than that, the definitions are intended to highlight why, as an engineer, *you should care* about vectors, matrices, and mathematical operations done with them.

In this course, when we numerically solve the heat equation to find temperature, say, in a cylindrical rod, the function that we find as the solution will be represented as a vector. If we take the derivative of the temperature so that we can calculate heat flux, for instance, the function that is the derivative of the temperature will also

¹ **Note:** Unless otherwise indicated we will assume any matrix to be two-dimensional. Higher-dimensional generalizations of a matrix exist, but they are not within the scope of this class.

be stored as a vector. Representing functions is the main thing that vectors are *used for* in this class.

There are instances in this class where we will store a rectangular array of numbers and call it a matrix. An example would be the “Runge-Kutta matrix” that stores coefficients used in various Runge-Kutta algorithms for solving initial value problems. But a more representative example of what we use matrices for applies when we are solving boundary value problems using the finite difference or finite element method. Consider, as an example, the case of a numeric solution of Poisson’s equation:

$$\left(\frac{\partial^2}{\partial x^2} \right) u(x) = f(x)$$

$$Au = f$$

In this expression the matrix A in the second row is a discrete representation of the $\partial^2/\partial x^2$ operator and the vectors u and f are discrete representations of the functions $u(x)$ and $f(x)$. We solve this discrete form of the differential equation by solving the matrix-vector equation using linear algebra.

Operations with Matrices

Matrix Equality

Consider two matrices, A and B . We say A and B are *equal* and write:

$$A = B$$

if $a_{i,j} = b_{i,j}$ for all i, j and if A and B are the same size.² Where, in this context, we use the word *size* to refer to the number of rows and columns in a matrix.

Multiplication by a Scalar

Any matrix, A , can be multiplied by a scalar, α :

$$B = \alpha A$$

where $b_{i,j} = \alpha a_{i,j}$ for all values i, j and the output matrix B is the same size as A .

Addition of Two Matrices

Matrices of the same size can be added:

$$C = A + B$$

² Here we use the notation $a_{i,j}$ and $b_{i,j}$ to refer to the element of A or B in the i -th row and j -th column. For a two-dimensional matrix, the MATLAB built-in function [m,n]=size(A) returns the number of rows, m, and columns, n, of A .

Note: In MATLAB this operation would be encoded exactly as written: $C = A + B$.

The output matrix C is the same size as A and B . This operation is not defined if A and B are not the same size.

Matrix Multiplication

Multiplication of two matrices is denoted:

$$C = AB$$

The output matrix C has the same number of rows as A and the same number of columns as B . This operation is only defined if A has the same number of columns as B has rows.³ Each element, $c_{i,j}$, of C is equal to the *dot product* of row i of A with column j of B . More explicitly, suppose A has m rows and n columns, B has n rows and p columns, The $c_{i,j}$ element of C is calculated:

$$c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}$$

where $i = 1, \dots, m$ and $j = 1, \dots, p$.⁴

Note that matrix multiplication, in general, is not commutative. The operations: $C = AB$ and $D = BA$ may or may not both be defined but they are only equal under very specific circumstances.⁵

Matrix Transpose

The matrix transpose is an operation where the rows and columns of a matrix are interchanged. For example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

This operation is usually only needed to support the semantics of linear algebraic operations which we will discuss later in the text.

Special Matrix Structure

There are several matrices that are special owing to very specific structural properties. These include:

1. *Square matrix*. A matrix is square if it has the same number of rows as it has columns.
2. *Symmetric matrix*. A matrix is symmetric if it is equal to its own transpose: $A = A^T$.
3. *Zero matrix*. This is a matrix where all of the elements are zero. This special matrix serves as the additive identity in matrix algebra.

³ Matrices A and B with this property are said to be *conformable*.

⁴ **Important Note:** In MATLAB this operation would be written: $C = A*B$. Notice that we do *not* use the $.*$ notation in this case because that would imply element-by-element multiplication which: a) may not be possible since A and B are not necessarily of the same size; and b) is not the same mathematical operation as matrix multiplication.

⁵ If A is of size $[m, n]$ and B is $[p, q]$:

1. $C = AB$ is valid if $n = p$. C will be of size $[m, q]$.
2. $D = BA$ is valid if $q = m$. D will be of size $[p, n]$.
3. If A and B are *square*—same number of rows as columns—and both are the same size, then C and D are both valid and are both the same size as A and B .
4. If C and D are both valid and the same size, $C \neq D$ unless both A and B are *diagonal*.

The following definitions of structural properties apply only to *square matrices*.

4. *Diagonal matrix.* A diagonal matrix is one that is zero except along the main diagonal. In mathematical notation: $a_{i,i} \neq 0$ and $a_{i,j} = 0$ where $i \neq j$.
5. *Identity matrix.* The identity matrix, denoted I , is a diagonal matrix with ones along the main diagonal. This matrix serves as the multiplicative identity in matrix algebra. For example, for $n = 3$, the identity matrix is:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and, for any 3×3 matrix A :

$$AI = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} = A$$

Note: It is also true that $IA = A$.

6. *Lower/Upper triangular matrix.* A lower triangular matrix is non-zero only for entries at or below the main diagonal. If the main diagonal is also zero, the matrix is referred to as *strictly lower triangular*. An upper triangular matrix is defined similarly: non-zero only on the main diagonal or above; if the main diagonal is zero, then the matrix is called *strictly upper triangular*.

Matrix Properties

We have already mentioned some of the structural properties of a matrix. There are other important properties not solely related to its size, zero-structure, or symmetry of entries.

1. *Rank.* The matrix rank is the number of rows or columns of a matrix that are *linearly independent*. If a_i denotes the i^{th} row or column of matrix A , then the rows/columns are linearly independent if:

$$\alpha_1 a_1 + \alpha_2 a_2 + \cdots + \alpha_n a_n = 0, \iff \alpha_i = 0, \forall i \in \{1, \dots, n\}$$

Which means that a linear combinations of the rows/columns can only be equal to zero (a vector of all zeros) *if, and only if, all* of the coefficients in the linear combination are equal to zero.⁶

Example: The matrix:

$$A = \begin{bmatrix} 1 & 0 & 1 \\ -2 & -3 & 1 \\ 3 & 3 & 0 \end{bmatrix}$$

Note: The mathematical notation \iff should be read "if and only if", and \forall means "for all." Consequently, $\iff \alpha_i = 0, \forall i \in \{1, \dots, n\}$ means: "if and only if α_i equals zero for all i in the set 1 to n."

⁶ This is completely analogous to the concept of linear independence of functions described in Lecture 3 of the analytical methods part of this text.

is of rank 2 since column 3 is equal to the difference between column 1 and column 2.

Example: The diagonal matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

is full rank since no linear combination of the rows or columns is equal to the zero vector.

If A is an $[m, n]$ matrix, the matrix rank is *at most* the minimum of m , and n . If A is a square matrix and $\text{rank} = m$, we say the matrix is *full rank*. If A is a rectangular $[m, n]$ matrix with independent columns, then $A^T A$ is full rank.

2. *Invertibility.* If the square matrix A is invertible, then there exists a square matrix of the same size, denoted A^{-1} , such that:

$$AA^{-1} = A^{-1}A = I$$

where I is the identity matrix of the same size as A . If A is *square* and is *full rank* then A is invertible.

A conceptual use of the matrix inverse is in solving a linear system of equations. If A is an invertible matrix with n rows, and b is a given vector of length n , then we can solve the system of equations:

$$Ax = b$$

by applying the inverse of A :

$$\begin{aligned} A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b \\ \Rightarrow x &= A^{-1}b \end{aligned}$$

While we will not use the matrix inverse in this way, it is still conceptually important to know when a matrix inverse exists.

3. *Matrix eigenvalues.* Consider the equation below:

$$Au = \lambda u$$

for the matrix A and vector u where A and u are conformable in multiplication and λ is a constant. This is called an *eigenvalue equation*. The vector u is called an *eigenvector* and the constant λ is called an *eigenvalue*.

If A is an invertible matrix with m rows and columns then it has a full set of m linearly independent eigenvectors; if A is symmetric then the eigenvectors are orthogonal.⁷ In either case, all of the

Note: The identity matrix works with vectors also, so long as the vector is conformable for multiplication. If I is conformable with the vector x , then $Ix = x$.

⁷ Students who have taken the analytical methods class should find this language to be eerily familiar: exchange *eigenvector* with *eigenfunction* and replace the matrix A with “*a linear differential operator*” and you would be forgiven for thinking we have accidentally skipped back to lecture 18 from the analytical methods class.

eigenvalues are non-zero. Most square matrices can be “*diagonalized*” in this way:

$$A = U\Lambda U^{-1}$$

in which U is an $n \times n$ matrix where each column is an eigenvector and Λ is a diagonal matrix with the eigenvalues along the diagonal. Square matrices that cannot be diagonalized in this way are said to be *defective*.⁸

There is at least one matrix property related to the eigenvalues that students should be aware of. A symmetric matrix is said to be *positive definite* if the following equation holds:

$$x^T Ax \geq 0, \text{ and } x^T Ax = 0 \Rightarrow x = 0$$

If this equation is true, it implies that all of the eigenvalues of A are positive. Apart from implying that A is thus invertible (all eigenvalues non-zero), and square (only square matrices are invertible), if a matrix is positive definite there are significant impacts in which methods one should use to solve a system of equations involving that matrix. We will revisit this property when discussing solution methods for linear systems of equations.

4. *Matrix determinant.* The determinant is a useful property of a matrix.⁹ The operation of taking the determinant is denoted:

$$\det(A) = |A|$$

There is also a relationship between the determinant of a matrix and the eigenvalues of a matrix:

$$|A| = \prod_{i=1}^n \lambda_i$$

where λ_i are the eigenvalues of A . Since, as previously mentioned, all of the eigenvalues of an invertible matrix are non-zero, the determinant of an invertible matrix is non-zero.

⁸ The language used in this should indicate to you how important it is for a matrix to have a full set of eigenvectors and able to be diagonalized in this way.

⁹ **Personal aside:** As a young ensign in the navy nuclear power training program I remember being taught an operational definition of (the thermodynamic property) entropy. “*entropy: a useful thermodynamic property.*” A determinant is like that: it is hard to have a *short* conversation about all of its implications so nobody wants to get into exactly what it means, but it is useful and you may need to calculate it.

Part VII

Solving Non-Linear Equations

Lecture 3 - The Bisection Method

Objectives

The objectives of this lecture are to:

- Introduce the problem of solving non-linear equations.
- Describe the bisection method.
- Discuss error estimates and stopping criteria.
- Illustrate the bisection method with an example.

Solving Non-Linear Equations

Engineers often need to solve non-linear equations. An equation of one variable can be written in the form:

$$f(x) = 0 \quad (125)$$

A solution to such an equation is called a *root*. In past lectures, in the analytical methods portion of this text, we frequently needed to find the root(s) of linear and non-linear equations. We also encountered several cases where we were not able to find the roots analytically.¹ The case that this lecture will address is when such roots cannot be found algebraically. As an example, suppose we wish to find the area marked A_s as shown in Figure 65 which is given in Equation 126.

$$A_s = \frac{1}{2}r^2 (\theta - \sin \theta) \quad (126)$$

where A_s and r are given. This is a non-linear equation and we wish to solve it numerically.

THERE ARE TWO general approaches with numerical methods to solve non-linear equations:

1. bracketing methods; and
2. open methods.

¹ As a quick re-cap:

- In lecture 19 we needed to find the roots of $J_0(x)$. We did not write our own routine for doing this but, at its heart, besselzero() had to solve a non-linear equation. We used these tools again in lectures 31, 32, and 33 along with several homework problems.
- In lecture 28 we needed to find roots of $f(x) = \tan x + x/h$ for a given value of h .

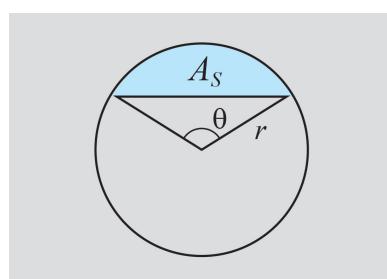


Figure 65: Schematic of example.

In bracketing methods, we start with an interval in which we know a root to exist. Each step of the algorithm seeks to reduce the size of the interval while always maintaining the root inside. When the interval is small enough, we have effectively found the root. In open methods, no such bounding interval is maintained. We start with an estimate of the root and, through successive iterations of the algorithm, attempt to improve our estimate. The iteration stops when our estimation of the root meets a specified termination criterion.

Bisection Method

The bisection method is a bracketing method for finding a root of a *continuous* function when it is known that the root lies within a specified interval: $x \in [a, b]$. The bisection method consists of the following steps:

1. Find an interval $[a, b]$ in which a root exists. One way to find such an interval is to identify an a and b for which $f(a)f(b) < 0$. As is illustrated in Figure 66, if $f(x)$ is continuous, this is a sufficient condition for a root to exist in $[a, b]$.²
2. Calculate an estimate of the numerical solution from Equation 127:

$$x_{NS} = \frac{(a + b)}{2} \quad (127)$$

3. Determine whether the root is in $[a, x_{NS}]$ or in $[x_{NS}, b]$ using the following method:
 - If $f(a)f(x_{NS}) < 0$, the root is in $[a, x_{NS}]$.
 - If $f(x_{NS})f(b) < 0$, the root is in $[x_{NS}, b]$.

Except in the edge case that $f(x_{NS}) = 0$, only one of the above conditions will be true.³

4. Select the subinterval that contains the solution—either $[a, x_{NS}]$ or $[x_{NS}, b]$ as determined in step 3—as the new a and b and return to step 2.

Three iterations of the bisection method are illustrated in Figure 67.

HERE WE OFFER some comments on the bisection method.

1. If $f(x)$ is continuous and exactly one root is contained in the initial interval $[a, b]$, the bisection method is guaranteed to converge with the error in x_{NS} reduced by a factor of two with each iteration of the algorithm.

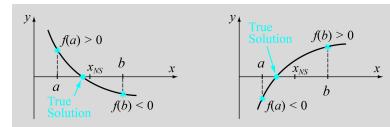


Figure 66: Solution of $f(X) = 0$ must lie in $[a, b]$ if $f(a)f(b) < 0$.

² Note: How one goes about determining a suitable a and b is not part of the algorithm.

³ Health Warning: You should definitely test this edge case.

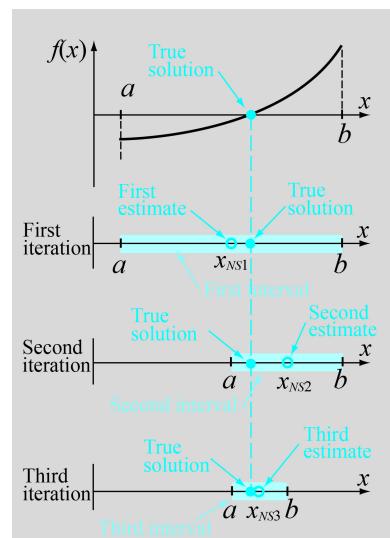


Figure 67: First three iterations of the bisection method.

2. The convergence rate is steady but slow relative to other methods.⁴
3. The test $f(a)f(b) < 0$ is a *sufficient* condition for a root to exist between $[a, b]$ but is not a *necessary* condition. For example, the function $f(x) = (x - 3)^2$ has two roots at $x = 3$, but with $a = 1$ and $b = 4$, we find that $f(1)f(4) > 0$ and the test fails.
4. The algorithm is tailored to find just one root. If multiple roots are desired, one must change the algorithm accordingly.

⁴ Since the error is reduced by a constant fraction with each iteration, the bisection method is said to exhibit *linear convergence*.

We will do an example problem, but before we get to that, a quick discussion of error estimation is in order.

Error Estimates and Stopping Criteria

Error estimation plays an essential role in bracketing algorithms.

After all:

1. Such algorithms do not really “find” the root; instead the algorithm starts with an interval in which the root lies and steps of the algorithm are carried out to make the interval arbitrarily small, all while keeping the root within the interval. In this sense, while the precise real number that is the root of the equation is not found, we place the root within a range that is as small as is desired.
2. Indeed, if $f(x)$ is a non-linear equation, and $f(x^*) = 0$, there is no reason to expect that x^* can be exactly represented on the computer using double-precision floating point numbers.

Note: We use the notation x^* to denote the *exact solution*. In places we will also use the notation: x_{TS} to denote the exact or *true* solution.

Consequently, no matter what result is produced from a bracketing method, we know it is in error. Our job is to be mindful of this error and be sure that it is kept to within acceptable bounds. In addition to controlling the magnitude of the error, bracketing algorithms customarily use error estimates as stopping criteria. We stop the algorithm when the estimated error is reduced below a pre-defined threshold.

There are several standard error estimates that we will consider.

1. True error. This is given by Equation 128.

$$\text{True error} = x^* - x_{\text{NS}} \quad (128)$$

This is not often used as a stopping criterion since, except in special testing cases, we generally do not know what the true solution, x^* , is.

2. Tolerance in $f(x)$ given by Equation 129.

$$\text{TOL} = |f(x^*) - f(x_{\text{NS}})| = |0 - \epsilon| = |\epsilon| \quad (129)$$

where ϵ is an error tolerance chosen by the user, e.g. 10^{-6} . The value of $f(x_{\text{NS}})$ is sometimes referred to as the *residual*. Use of this stopping criterion is generally only satisfactory if $f'(x) \approx 1$ in the vicinity of the root.

3. Tolerance in the solution. If $x_{\text{NS}} = a + b/2$, then:

$$\text{TOL} = \left| \frac{b - a}{2} \right|$$

which is half the subinterval. This is a reasonable stopping criterion for the bisection method or any other bracketing method.⁵

4. Relative error. For most numerical algorithms, if not necessarily the bisection method, the *estimated relative error* is usually a good technique and is shown in Equation 130.

$$\text{Estimated Relative Error} = \frac{\left| x_{\text{NS}}^{(n)} - x_{\text{NS}}^{(n-1)} \right|}{\left| x_{\text{NS}}^{(n-1)} \right|} \quad (130)$$

where $x_{\text{NS}}^{(n)}$ is the estimated numeric solution at iteration n , and $x_{\text{NS}}^{(n-1)}$ is the estimated numeric solution at iteration $n - 1$. This error estimate has the advantage that it is a *relative error measure* and takes into account the magnitude of x_{NS} . A technical disadvantage arises if x_{NS} is close to zero at any iteration.

Refinements on these stopping criteria will be discussed, as appropriate, throughout the course.

MATLAB Implementation

Example: You are tasked to estimate the pressure drop in a 0.1-m long section of a tube ($D = 4.0$ mm diameter) that is used to transfer air with an average velocity of 50 m/s. Under these conditions, the Reynolds number (Re) of the air in the tube is approximately 13,700. Assume the drawn tubing has an equivalent roughness (ϵ) of 0.0015 mm. As the first step, you plan to use the Colebrook equation to estimate the friction factor. The Colebrook equation is given by:

$$\frac{1}{\sqrt{f}} = -2.0 \log_{10} \left(\frac{\epsilon/D}{3.7} + \frac{2.51}{\text{Re}\sqrt{f}} \right)$$

We will start by clearing out the MATLAB workspace and encoding given data.

⁵ Obviously, it is not suitable for non-bracketing algorithms since a and b would then be unknown.

Note: Reynolds number is a non-dimensional parameter that relates the relative magnitude of inertial and viscous forces. It is used to predict fluid flow patterns. The value of the Reynolds number is given by the equation:

$$\text{Re} = \frac{\rho \bar{v} D_c}{\mu}$$

where ρ is fluid density, \bar{v} is a characteristic—e.g. average—velocity, D_c is some characteristic length, such as in this case the diameter of the tube, and μ is the fluid viscosity. Higher Reynolds numbers (such as is given in this example) correspond to turbulent flow fields, low Reynolds number correspond to laminar flow.

```

clear
clc
close 'all'

e = 0.0015; % mm, equivalent roughness
D = 4.0; % mm, tube diameter
Re = 13700; % Reynolds number

```

1
2
3
4
5
6
7

Next we encode the nonlinear function whose roots we seek and select an initial interval in which we expect the root to reside.

```

F = @(f) 1./sqrt(f) + 2.0*log10((e/D)/3.7 + 2.51/(Re*sqrt(f)));

a = 0.001; b = 1; % initial interval
imax = 20; tol = 0.0001; % stopping criteria ❶

Fa = F(a); Fb = F(b);

% verify that Fa*Fb > 0
if (Fa*Fb > 0)
    error('Error: The function has the same sign at a and b'); ❷
end

```

8
9
10
11
12
13
14
15
16
17
18

Now we are ready to commence the bisection method. In this listing we provide code necessary to provide outputs to the command window to update the user on the progress of the algorithm.

```

fprintf('iter      a          b          xNS        f(xNS)\n'
       '      Tol\n');
for i = 1:imax
    xNS = (a + b)/2; ❸
    toli = (b - a)/2;
    FxNS = F(xNS);
    fprintf('%3i %11.6f %11.6f %11.6f %11.6f\n', ...
             i,a,b,xNS,FxNS,toli);

    if FxNS == 0
        fprintf('Exact solution x = %11.6f was found\n',xNS); ❹
        break;
    end
    if toli < tol
        fprintf('Success!! x = %11.6f \n',xNS); ❺
        break; % end the loop since I meet my error tolerance
    end
    % if toli > tol and i == imax, stop iteration.
    if i == imax ❻
        fprintf('Solution not obtained in %i iterations\n',imax);
        break
    end
    % update bracket
    if (F(a)*FxNS < 0)
        b = xNS;
    else
        a = xNS; ❼
    end
end

```

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46

❶ For the bisection method, these two stopping criteria may conflict. We will use the tolerance in the solution criteria which, per the bisection algorithm, is halved with every iteration. At the same time we are limiting the maximum number of iterations. If the maximum number of iterations is too small, depending on the initial choice of a and b , it may be impossible to satisfy the tolerance. The smallest tolerance you can satisfy is: $TOL_{min} = \left\lfloor \frac{b-a}{2^{imax}} \right\rfloor$. With the given choices of a , b , and tol , we should never reach the maximum specified number of iterations. Still, it is okay to be conservative to guard against bugs.

❷ Alternatively you might try to implement some routine that may automatically find an interval that spans a root.

❸ This is step #2 of the bisection method.

❹ Check the edge case where the root was exactly at the midpoint of the interval.

❺ If the tolerance in the solution at iteration i meets the stopping criterion, stop the iteration. The keyword `break` results in an immediate exit from the `for ... end` loop.

❻ Here we warn the user that the algorithm is about to be stopped based on maximum iterations. Since the specified tolerance in the solution was never met, the user might want to re-start with a new (smaller) bracket, increase the tolerance, or increase the maximum allowed number of iterations.

❼ This is step #3 and step #4 of the bisection method.

For this example problem the iteration succeeds and finds an estimated root at $x = 0.029109$ after 14 iterations.

Summary

The bisection method discussed in this lecture is useful. It is both simple to implement in code and reliably converges to a root provided that a suitable initial interval is established. The convergence rate is slow compared to the methods that we will explore in the next lecture. This slow convergence is not a significant problem for the problem types you are likely to encounter in a typical classroom environment; computers are fast and they do not mind working a little bit longer. The convergence rate becomes an issue when evaluation of the function (to which we are finding a root) is computationally expensive. Still, the bisection method is sufficient for most purposes. In upcoming lectures we will explore improved algorithms. One bracketing method we will discuss is called the regular falsi method. It is introduced primarily as a way to show how one can try to make a good algorithm, like bisection method, better.

Assignment #1

1. Write the number 38.8125 as a 64-bit double-precision string using the IEEE-754 standard.
2. In single precision (IEEE-754 standard), 8 bits are used for storing the exponent (the bias is 127), and 23 bits are used for storing the mantissa. What (approximately) are the smallest and largest positive numbers that can be stored in single precision?
3. The value of π can be calculated with the series:

$$\pi = 4 \sum_{n=1}^{\infty} (-1)^{n-1} \frac{1}{2n-1} = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots \right)$$

Write a MATLAB script that calculates the value of π by using n terms of the series and calculate the corresponding true relative error. Calculate π and the true relative error for $n = 10, 20$, and 40 . **Note:** Implement the partial series summation as a *local function* named piApprox that takes one argument—the number of terms (n)—and returns the estimated value of π and the true relative error. Use MATLAB's built-in constant pi for the "true" value of π .

4. Using a hand calculator, determine the root of $f(x) = x - 2e^{-x}$ with the bisection method. Start with $a = 0$ and $b = 1$, and carry out the first three iterations to determine an estimated root and bracket within which the root lies.

5. Write a MATLAB user-defined function that solves for a root of a nonlinear equation $f(x) = 0$ using the bisection method. Implement the function as a *local function* that takes three arguments: *fun*, *a*, and *b*, where *fun* is a handle to the nonlinear function for which a root is to be found and *a* and *b* bracket the root. The bisection iterations should stop when $f(x_{NS}) \leq 0.0000001$ where x_{NS} is the midpoint of the current bracket. The function should also check if points *a* and *b* do indeed bracket a root; if not, the function should return an error message. Use your function to find the root of $f(x) = x - 2e^{-x}$.
6. The van der Waals equation gives a relationship between the pressure *P* (in atm.), volume *V* (in liters), and temperature *T* (in K) for a real gas:

$$P = \frac{nRT}{V - nb} - \frac{n^2a}{V^2} \quad (131)$$

where *n* is the number of moles, *R* = 0.08206 (L-atm)/(mole-K) is the gas constant, and *a* (in L²-atm/mole²) and *b* (in L/mole) are material constants. Consider 1.5 moles of nitrogen (*a*=1.39 L²-atm/mole²) at 25°C stored in a pressure vessel. Use the function you created for problem #5 to determine the volume of the vessel if the pressure is 13.5 atm.

Lecture 4 - Newton's Method and Secant Method

Objectives

The objectives of this lecture are to:

- Describe and demonstrate Newton's method for solving non-linear equations.
- Illustrate the convergence properties of Newton's method and highlight convergence issues.
- Introduce and demonstrate the secant method.

Introduction

The bisection method described in the previous lecture had at least two issues:

1. End-points for the bracket $[a, b]$ in which a root is contained needed to be determined. There is no cut-and-dried sure-fire way to accomplish this task. Luckily, a suitable bracket can often be found even if such a method is hard to automate and may feel a bit "clunky."
2. The convergence rate is "slow"—although we have not yet described any competing methods that are "faster" such that a proper perspective can be established.

The open methods to be described in this lecture eliminate the first drawback of bracketing methods; open methods need no bracket containing a root, only an initial starting point. Also, it will be shown, that the open methods described in this note converge more quickly than bracketing methods such as the bisection algorithm thereby clarifying the impact of the second drawback.

On the other hand, open methods will introduce additional complications that offset some of their advantages. These issues will be described and options for mitigating these complications will be presented.

Newton's Method

Newton's method, also called the Newton-Raphson method, is a classic method for finding roots of equations that are *continuous* and *differentiable*. Given an initial estimate of the solution x_1 , the second estimate is determined by taking the tangent line to $f(x)$ at the point $[x_1, f(x_1)]$ and finding the intersection of the tangent line with the x -axis. A schematic of the first four iterations of Newton's method is shown in Figure 68. Successive estimates are found in the same way. The slope of the tangent line at x_1 is denoted $f'(x_1)$. The point x_2 is determined from Equation 132.

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \quad (132)$$

This process can be generalized for finding solutions x_{i+1} from x_i as shown in Equation 133.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (133)$$

The algorithm for Newton's method can be stated as follows:

1. Choose a point x_1 as an initial guess of the solution.
2. For $i = 1, 2, \dots$, until the error is smaller than a specified tolerance, use Equation 133 to find x_{i+1} and repeat.

SINCE THIS IS NOT a bracketing method, we do not have a "bracket" to use in estimating the solution error like we did in the bisection method. There are a couple of stopping criteria that could be used:

1. Tolerance in $f(x)$. Recall from the last lecture, this means we will stop the iteration when, for some specified tolerance δ , the following relation is satisfied:

$$|f(x_i)| \leq \delta \quad (134)$$

On the surface this is a sensible error measure—if $f(x_i)$ is close to zero, then it stands to reason that x_i is close to the exact solution, x^* . It turns out, however, this is only true if $f'(x_i) \approx 1$ in the vicinity of x^* . To see why, consider the equation below:

$$\begin{aligned} f'(x_i) &\approx \frac{f(x^*) - f(x_i)}{x^* - x_i}^0 \\ &\Rightarrow x^* - x_i \approx \frac{-f(x_i)}{f'(x_i)} \end{aligned}$$

If $f'(x_i) \approx 1$, then $|x^* - x_i| \approx |f(x_i)| \leq \delta$ which is what we want. But if $f'(x_i) \ll 1$, then $|x^* - x_i|$ is much greater than δ so we are not as close to x^* as we want to be.¹

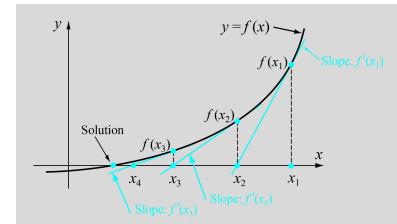


Figure 68: The first four iterations of Newton's method.

Note: The expression for x_2 given in Equation 132 is obtained from:

$$f'(x_1) = \frac{f(x_1) - 0}{x_1 - x_2}$$

¹ Remember that:

$$x^* - x_i = \frac{-f(x_i)}{f'(x_i)}$$

Thus, as $f'(x_i) \rightarrow 0$, then $x^* - x_i \rightarrow \infty$.

2. Estimated relative error. For Newton's method we will use the *estimated relative error* as our measure of convergence. Specifically, the iterations will be stopped when, for some specified tolerance ϵ , the following relation is met:

$$\left| \frac{x_{i+1} - x_i}{x_i} \right| \leq \epsilon \quad (135)$$

Convergence Rate of Newton's Method

To help illustrate the convergence rate of Newton's method, let us do an example. We will find the positive root of the function: $f(x) = x^2 - 2$, which we know is equal to $\sqrt{2}$. MATLAB code to implement the algorithm is shown in the listing below:

```

clear
clc
close 'all'

% find the square root of two
F = @(x) x.^2 - 2; ①
dF = @(x) 2*x;

Xest = 1.; % initial guess ②
Err = 1e-15;
imax = 5;
fprintf('Exact solution = %16.15f \n',sqrt(2));③

for i = 1:imax
    Xi = Xest - F(Xest)/dF(Xest);

    if abs((Xi - Xest)/Xest) < Err
        xNS = Xi;
        break;
    end

    fprintf('After %i iterations , Xi = %16.15f \n',i,Xi);

    Xest = Xi;
end

```

① We implement $f(x) = x^2 - 2$ and its derivative as in-line functions.

② We do not need an initial bracket but we do need an initial guess. For some problems it may be essential that the initial guess is close to the solution.

③ We will use the value of $\sqrt{2}$ to full double-precision to approximate the "exact solution."

An assessment of the output is shown below with the correct digits of the estimates solution shown in bold:

$$\begin{aligned}
 \sqrt{2} &\approx 1.414213562373095 \\
 x_1 &= \mathbf{1.500000000000000} \\
 x_2 &= \mathbf{1.416666666666667} \\
 x_3 &= \mathbf{1.414215686274510} \\
 x_4 &= \mathbf{1.414213562374690} \\
 x_5 &= \mathbf{1.414213562373095}
 \end{aligned}$$

Note that between iteration 2 and 3, and between iteration 3 and 4, the number of correct digits doubles each time. By the time that the 5th iteration is arrived at, the estimated solution is correct to full double precision. It can be shown² that the relative error in Newton's method can be approximated as:

$$e_{i+1} \approx \frac{f''(x^*)}{2f'(x^*)} e_i^2$$

In words: the relative error at step $i + 1$ is proportional to the error at step i squared. This is referred to as *quadratic convergence*.

Convergence Issues with Newton's Method

Alas, not all is well with Newton's method. When the method works, it converges very rapidly. Frequently, however, the method does not converge. Convergence problems can happen under the following conditions:

1. When the value of $f'(x)$ is close to zero in the vicinity of the solution. When that happens, the update from x_i to x_{i+1} as given by Equation 133 is very large. If x_i was close to the solution initially, x_{i+1} may be very far from the solution. Depending on how $f(x)$ varies between x_i and x_{i+1} this may result in convergence to a different root, for example, or other unpredictable behavior.
2. When the function has a local minimum (i.e. $f' \approx 0$) in which case, again, the next iteration can be projected very far from the current value of x_i . This condition is illustrated in Figure 69.
3. An inflection point can result in the iteration scheme entering into a cycle. A schematic of this diabolical (and, thankfully, relatively uncommon) state of affairs is shown in Figure 70.

Notwithstanding these difficulties, it is possible to show that Newton's method will converge if the function, $f(x)$, and its first and second derivatives are all *continuous*, and if the first derivative is not zero at the solution, and if the starting value, x_1 , is "near" the exact solution.³

ANOTHER ISSUE WITH Newton's method, not directly related to convergence, is that to use the method you need both the function as well as its derivative. The derivative is not always easy to obtain. In some cases the "function" to be solved is not available in the form of an equation but rather a complex computational routine that does not admit to analytical differentiation. Combined with the convergence difficulties, there is plenty of motivation to come up with a different algorithm.

² GW Stewart. *Afternotes on Numerical Analysis*. University of Maryland at College Park, 1993

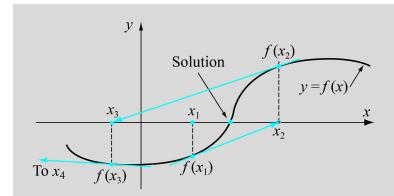


Figure 69: Newton's method convergence issue due to a local minimum.

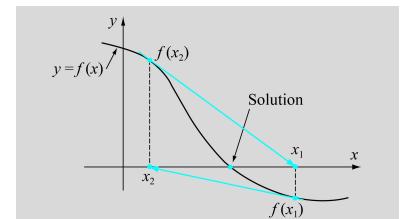


Figure 70: Newton's method failure to converge due to inflection points in vicinity of solution.

³ Whatever the hell "near" is supposed to mean.

Secant Method

The secant method is another open scheme for finding a numerical solution to a non-linear equation. Unlike Newton's method, the secant method does not require an expression for $f'(x)$. With the secant method we instead approximate this derivative by using the *secant line* between two initially provided points. This is illustrated in Figure 71 where it is also shown that it does not matter if x_1 and x_2 bracket the root or not.

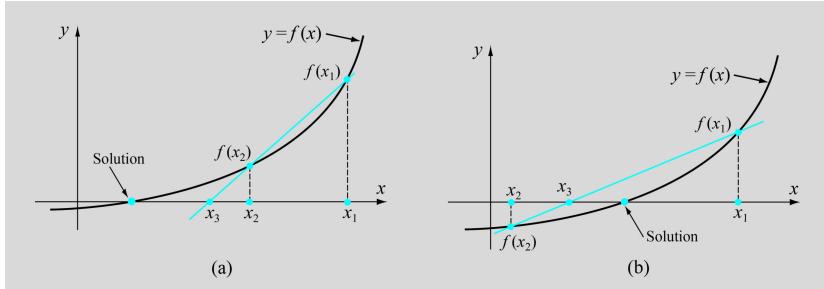


Figure 71: Schematic of the secant method.

The slope of the secant line is given by:

$$\begin{aligned} \frac{f(x_1) - f(x_2)}{x_1 - x_2} &= \frac{f(x_2) - 0}{x_2 - x_3} \\ x_3 &= x_2 - \frac{f(x_2)(x_1 - x_2)}{f(x_1) - f(x_2)} \end{aligned}$$

Once x_3 is determined, it is used along with x_2 to find x_4 and so it goes. We generalize this iteration in Equation 136.

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \quad (136)$$

Successive iterations of the secant method are illustrated in Figure 72.

WHILE THE SECANT method does not rely on an expression for $f'(x)$, it does require additional evaluations of $f(x)$. If each evaluation of $f(x)$ is computationally expensive then the programmer who is implementing the method would be wise to minimize or eliminate any extraneous evaluations of $f(x)$. While this point probably seems obvious, it is worth mentioning that this consideration amounts to a design trade-off. Program complexity and the number of additional variables that you may store to avoid re-evaluating a function can add up. Sometimes it pays to simply re-compute $f(x)$ whenever you need it; sometimes it does not. Keep this in mind when you design your own implementation.

Note: We can slightly re-arrange Equation 136 to get:

$$x_{i+1} = x_i - \frac{f(x_i)}{\frac{f(x_{i-1}) - f(x_i)}{(x_{i-1} - x_i)}}$$

This may help emphasize the similarity between the secant method and Newton's method. If x_{i-1} and x_i are sufficiently close together:

$$\frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i} \approx f'(x_i)$$

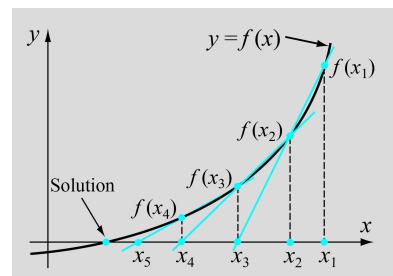


Figure 72: Secant method in action.

Convergence of Secant Method

The MATLAB code for testing the convergence of the secant method is provided in the listing below. Again we solve the equation: $f(x) = x^2 - 2$. We use the starting points, $x_1 = 0$ and $x_2 = 2$.

```

1 clear
2 clc
3 close 'all'

4 % find the square root of two
5 F = @(x) x.^2 - 2;
6
7 Xa = 0;
8 Xb = 2;
9
10 Err = 1e-16;
11 imax = 15;
12 fprintf('Exact solution = %16.15f \n',sqrt(2));
13 for i = 1:imax
14   FXb = F(Xb);
15   Xi = Xb - FXb*(Xa - Xb)/(F(Xa) - FXb);
16
17   if abs((Xi - Xb)/Xb) < Err
18     xNS = Xi;
19     break;
20   end
21   Xa = Xb;
22   Xb = Xi;
23
24 fprintf('After %i iterations , Xi = %16.15f \n',i,Xi);
25 end
26

```

Results are as shown below:

$$\begin{aligned}
\sqrt{2} &\approx 1.414213562373095 \\
x_1 &= 1.000000000000000 \\
x_2 &= 1.333333333333333 \\
x_3 &= 1.428571428571429 \\
x_4 &= 1.413793103448276 \\
x_5 &= 1.414211438474870 \\
x_6 &= 1.414213562688870 \\
x_7 &= 1.414213562373095
\end{aligned}$$

Clearly the convergence rate for the secant method is inferior to Newton's method⁴ and yet, it is still better than bisection method.

Wrap-up

In this lecture we investigated open root-finding methods that, using the derivative of a function or, in the case of the secant method,

⁴ It should be noted that the exact convergence behavior depends, to some extent, on the values chosen for x_1 and x_2 .

an estimate of the derivative of a function, to find the root of a non-linear equation. We found that, for the expense of using the derivative, or estimate of a derivative, of a function, we were able to get quadratic convergence. If you need to get a very precise solution and/or individual function evaluations are computationally expensive, these methods have strong advantages over bisection.

THERE ARE AT LEAST two questions that you should be asking yourself:

1. Can I get even faster convergence if I also make use of the *second* derivative of the function? The answer is: **yes**. The algorithm that takes this approach is called Halley's method⁵ and the error after each iteration is proportional to the error after the previous step *cubed*.
2. Can I get convergence faster than the bisection method while retaining the reliability of the bisection method? The answer is also **yes**. The Brent method combines techniques based on the secant method, bisection method, and inverse quadratic interpolation to converge *at least* as fast as bisection—often much faster—while retaining the robust reliability of bisection. Theoretical details of the Brent method are available in the literature⁶ and a straightforward example implementation is given in a popular Numerical Recipes⁷ book. MATLAB's built-in non-linear equation solver `fsolve` is based on Brent's method.

Homework problems will be assigned that will help you explore the practical details of implementing these algorithms and observing the convergence properties.

⁵ Walter Gander. On Halley's iteration method. *The American Mathematical Monthly*, 92(2):131–134, 1985

⁶ George Elmer Forsythe, Cleve B Moler, and Michael A Malcolm. *Computer methods for mathematical computations*. Prentice-Hall, 1977

⁷ William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992

Lecture 5 - MATLAB Functions for Root Finding

Objectives

The objectives of this lecture are to:

- Describe the goals for more advanced root-finding algorithms and generally outline how they may be achieved.
- Discuss MATLAB's built-in function `fzero` for finding roots of non-linear functions.
- Illustrate how to use the advanced features of `fzero`.

Goals for Advanced Root-Finding Algorithms

The bisection method that we studied in Lecture 3 is reliable but convergence is slow. In particular it exhibits linear convergence where:

$$e_{i+1} = \frac{1}{2}e_i \quad (137)$$

Newton's method, that we learned about in Lecture 4, converges fast when everything works as hoped, but is relatively unreliable. What we want, of course, is an algorithm that converges *super-linearly* and *almost* never fails.

Regula Falsi Method

The regula falsi method is a simple alternative to the bisection method that holds the promise (potentially) for faster convergence. The method starts with a bracket $[a, b]$ like the bisection method but, instead of using the midpoint of the bracket for the next estimate of the solution, the regula falsi method uses the point in $[a, b]$ where the *secant line* between $f(a)$ and $f(b)$ is equal to zero. A schematic of the first two iterations is illustrated in Figure 73.

The straight line connecting $f(a)$ to $f(b)$ is given by Equation 138:

$$y = \frac{f(b) - f(a)}{b - a}(x - b) + f(b) \quad (138)$$

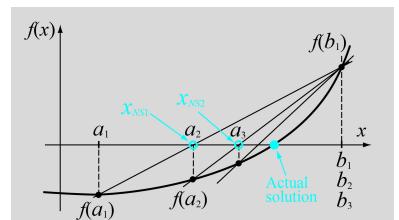


Figure 73: The first two iterations of the regula falsi method.

The point where the line intersects the x -axis is found by setting $y = 0$ and solving for x , giving us:

$$x_{\text{NS}} = \frac{af(b) - bf(a)}{f(b) - f(a)} \quad (139)$$

The algorithm for the regula falsi method is as follows:

1. Find an interval $[a, b]$ in which a root exists. One way to find such an interval is to identify an a and b for which $f(a)f(b) < 0$.
2. Use Equation 139 to find the new estimate of the numerical solution.
3. Determine whether the root is in $[a, x_{\text{NS}}]$ or in $[x_{\text{NS}}, b]$ using the following method:
 - If $f(a)f(x_{\text{NS}}) < 0$, the root is in $[a, x_{\text{NS}}]$.
 - If $f(x_{\text{NS}})f(b) < 0$, the root is in $[x_{\text{NS}}, b]$.
4. Select the subinterval that contains the solution—either $[a, x_{\text{NS}}]$ or $[x_{\text{NS}}, b]$ as determined in step 3—as the new a and b and return to step 2.

The iteration should be stopped when the estimated error is smaller than the pre-determined tolerance. Some notes on this method:

- Like the bisection method, the regula falsi method *will* converge. One can hope that, given the small added complexity, the rate of convergence would be somewhat faster using the regula falsi method.
- For a given function, it is typically the case that only one endpoint will “move towards” the solution. Modifications to the algorithm can be made to address this issue. An example of this, that will be included in the homework assignment, is to combine a step of the bisection method along with the regula falsi method. This measure helps ensure that the interval around the root is consistently reduced from both directions.
- Newton’s method, the secant method, and regula falsi method all, in one way or another, use linear interpolation to estimate the location of the root. We might hope to get a better approximation of the root if we used a higher-order interpolant.¹ Brent’s method combines root-bracketing, secant method, and *inverse quadratic interpolation* methods and is the basis of MATLAB’s `fzero` function.²

¹ This is an idea that we will return to in our lectures on interpolation.

² George Elmer Forsythe, Cleve B Moler, and Michael A Malcolm. *Computer methods for mathematical computations*. Prentice-Hall, 1977

MATLAB's *fzero*

MATLAB's built-in function *fzero* is the basic, bread-and-butter, root finding function that you should use in a MATLAB environment. The basic usage is:

$$x = \text{fzero}(f, xo)$$

where x is the root, f is a handle to the non-linear function, and x_0 is either a point (close to the solution) or a vector $x_0 = [a b]$ whose entries indicate the left and right boundary of an interval that brackets a solution.³

You can specify options and get more information about the solution by using optional input and output variables as shown below:

$$[x, fval, exitflag, output] = \text{fzero}(f, xo, \text{options})$$

where, in addition to the root, x , the function returns:

- $fval$: which is the value of $f(x)$ at the root.
- $exitflag$, which is a variable encoding the reason why *fzero* stopped iterations. Below is a table showing the meaning of various *exitflag* values:

³ MATLAB checks to see if $[a b]$ contains a root—i.e. by checking $f(a)f(b) < 0$ —and returns an error if it does not.

Value	Meaning
1	Function converged to a solution.
-1	Algorithm was terminated by the output function or plot function.
-3	Nan or Inf function value was encountered while searching for an interval containing a sign change.
-4	Complex function value was encountered while searching for an interval containing a sign change.
-5	Algorithm might have converged to a singular point.
-6	<i>fzero</i> did not detect a sign change.

Notice that all of these conditions are bad except for *exitflag*=1. This is one example where MATLAB seems to buck computing trends; it is customary for a wide range of codes to return a value of zero (0) to indicate that all is well.

- a variable called `output` which is a *struct*⁴ containing fields describing performance of `fzero` on this particular invocation.

The `output` struct contains the following fields:

Field Name	Meaning
<code>intervaliterations:</code>	Number of iterations taken to find an interval containing a root.
<code>iterations:</code>	Number of zero-finding iterations.
<code>funcCount:</code>	Number of function evaluations.
<code>algorithm:</code>	Name(s) of algorithm(s) used.
<code>message:</code>	An exit message. (e.g. "Zero found in the interval [a,b]")

The additional input variable, `options`, is also a struct. Users typically construct this object using the MATLAB function `optimset` with a collection of name-value pairs pertaining to input options available to `fzero`. See the MATLAB documentation for full details but a typical usage might be something like:

```
options = optimset('Display','iter','TolX',1e-15);
```

where '`Display`', '`iter`' causes MATLAB to display output at each iteration, and '`TolX`', $1e-15$ sets the termination tolerance on x to be equal to 10^{-15} .

⁴ A struct is a composite data type that facilitates a logically grouped list of variables under one name. Each variable is referred to as a *field* of the struct. Different fields can have different data types.

Assignment #2

1. Determine the fourth root of 200 by finding the numerical solution of the equation $f(x) = x^4 - 200 = 0$. Use Newton's method (by hand). Start at $x = 8$ and carry out the first five iterations.
2. Create a function to carry out Newton's method for finding a root for a nonlinear equation. The function should have the signature: `Xs=NewtonSol(Fun,FunDer,Xest)` where, `Xs` is a root, `Fun` and `FunDer` are handles to the function to be solved and its derivative, and `Xest` is an initial starting guess for the root. Your function should use the estimated relative error as a stopping criterion with a threshold value of 10^{-9} . The number of iterations should be limited to 100; if the solution is not found in 100 iterations the program should stop and display an error message. Use your function to solve the equation given in Problem #1.
3. Steffensen's method is a scheme for finding a numerical solution of an equation of the form $f(x) = 0$ that is like Newton's method but does not require the derivative of $f(x)$. The solution process starts by choosing a point x_1 , near the solution, as the first estimate of the root. The subsequent estimates of the solution, x_{i+1} , are calculated by:

$$x_{i+1} = x_i - \frac{f(x_i)^2}{f(x_i + f(x_i)) - f(x_i)}$$

Write a MATLAB function that solves a nonlinear equation with Steffensen's method. The function signature should be: `Xs = SteffensenRoot(Fun,Xest)` where `Xs` is a root, `Fun` is a handle to the function you wish to solve, and `Xest` is the estimated root. Use your function to find the root of $f(x) = x - 2e^{-x}$. Use 2 as an initial starting point. As termination criteria, use the estimated relative error with 10^{-9} as a tolerance, and maximum iterations with a limit of 100.

4. A simply-supported I-beam is loaded with a distributed load, as shown in the figure. (need to add the figure) The deflection, y , of the centerline of the beam as a function of the position, x , is given by the equation:

$$y = \frac{w_0 x}{360 L E I} (7L^4 - 10L^2 x^2 + 3x^4)$$

where $L = 4$ m is the length, $E = 70$ GPa is the elastic modulus, $I = 52.9 \times 10^{-6}$ m⁴ is the moment of inertia, and $w_0 = 20$ kN/m is the distributed load.

Find the position x where the deflection of the beam is maximum and determine the deflection at this point. [Hint: The maximum deflection is at the point where $y_x = 0$.]

- (a) Write a MATLAB function that solves this non-linear equation using the secant method. The function signature should be $X_s = \text{SecantRoot}(\text{Fun}, X_a, X_b, \text{Err}, \text{imax})$ where Err is the estimated relative error, a tolerance for which you should set at 10^{-9} . Use 1.5 for X_a and 2.5 for X_b .
- (b) Repeat the solution of this equation with `NewtonSol` and `SteffensonRoot`.

Lecture 6 - Newton's Method for Systems of Non-Linear Equations

Objectives

The objectives of this lecture are to:

- Describe Newton's method for finding roots to systems of non-linear equations.
- Do an example problem.
- Describe MATLAB functions for solving a system of non-linear equations.

Newton's Method for Systems of Non-Linear Equations

In Lecture 4 we derived Newton's method for finding the root to a single non-linear equation. The basic iteration procedure was to update our estimate of the root according to the equation:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The idea on which this was derived was to project a line tangent to $f(x_i)$ to the x -axis.

AN ALTERNATIVE WAY to derive this update relation is based on the Taylor series expansion. In a Taylor series expansion, given a function evaluated at some point, $f(x_1)$, we estimate the value of the function at some *other* point, x_2 , using the equation below:

$$\begin{aligned} f(x_2) &= f(x_1) + \frac{f'(x_1)}{1!}(x_2 - x_1) + \frac{f''(x_1)}{2!}(x_2 - x_1)^2 + \dots \\ &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_1)}{n!}(x_2 - x_1)^n \end{aligned}$$

If we ignore terms proportional to $f''(x_1)$ and higher, and if we suppose that x_2 is a root, we get:

$$\begin{aligned} f(x_2) &= 0 = f(x_1) + f'(x_1)(x_2 - x_1) \\ x_2 - x_1 &= \frac{-f(x_1)}{f'(x_1)} \\ \Rightarrow x_2 &= x_1 - \frac{f(x_1)}{f'(x_1)} \end{aligned}$$

which is the same as what we started with.

We WOULD NOW like to generalize Newton's method to find roots for a *system* of two or more equations. We will use the formulation based on the Taylor series expansion to do this.

Consider the case of 2 non-linear functions of 2 variables:

$$f_1(x, y) = 0, \quad f_2(x, y) = 0$$

If x_2 and y_2 are the true (unknown) solution to the equations, and x_1 and y_1 are points sufficiently close to the solution then:

$$\begin{aligned} f_1(x_2, y_2) &= 0 = f_1(x_1, y_1) + (x_2 - x_1) \frac{\partial f_1}{\partial x} \Big|_{x_1, y_1} + (y_2 - y_1) \frac{\partial f_1}{\partial y} \Big|_{x_1, y_1} \\ f_2(x_2, y_2) &= 0 = f_2(x_1, y_1) + \underbrace{(x_2 - x_1)}_{\Delta x} \frac{\partial f_2}{\partial x} \Big|_{x_1, y_1} + \underbrace{(y_2 - y_1)}_{\Delta y} \frac{\partial f_2}{\partial y} \Big|_{x_1, y_1} \end{aligned}$$

where higher order terms are neglected. This set of linear equations can be expressed in a matrix-vector format as shown in Equation 140.

$$\begin{bmatrix} \frac{\partial f_1}{\partial x} \Big|_{x_1, y_1} & \frac{\partial f_1}{\partial y} \Big|_{x_1, y_1} \\ \frac{\partial f_2}{\partial x} \Big|_{x_1, y_1} & \frac{\partial f_2}{\partial y} \Big|_{x_1, y_1} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -f_1(x_1, y_1) \\ -f_2(x_1, y_1) \end{bmatrix} \quad (140)$$

The unknown quantities are Δx and Δy . The matrix in Equation 140 is referred to as the *Jacobian*. We can solve this linear system of equations¹ to find Δx and Δy and thus get x_2 and y_2 from Equation 141.

$$x_2 = x_1 + \Delta x, \quad y_2 = y_1 + \Delta y \quad (141)$$

¹ Actually we have not covered how to do that yet in this class. Students can probably solve this 2×2 system of equations based on their experience in high school math, but we will thoroughly study methods for solving linear systems of equations in the next section of the text.

IN SUMMARY, Newton's method for solving a system of non-linear equations is made up of the following steps:

1. Start with an initial guess, x_1, y_1 .
2. Form and solve the linear system of equations given in Equation 140 to obtain Δx and Δy .
3. compute x_{i+1} and y_{i+1} from: $x_{i+1} = x_i + \Delta x$, and $y_{i+1} = y_i + \Delta y$.
4. Repeat steps #2 and #3 until Δx and Δy are within some specified error tolerance.²

Example: The equations of the catenary curve and the ellipse, which are shown in Figure 74, are given by:

$$\begin{aligned}f_1(x, y) &= y - \frac{1}{2} (e^{x/2} + e^{-x/2}) \\f_2(x, y) &= 9x^2 + 25y^2 - 225\end{aligned}$$

Use Newton's method to determine the point of intersection of the curves that resides in the first quadrant of the coordinate system.

The Jacobian for this system of equations is given by:

$$\begin{bmatrix} -\frac{1}{4} (e^{x/2} + e^{-x/2}) & 1 \\ 18x & 50y \end{bmatrix}$$

We begin by clearing out the workspace and defining the given functions and the Jacobian:

```
clear
clc
close 'all'

F1 = @(x,y) y - 0.5*(exp(x./2) + exp(-x./2));
F2 = @(x,y) 9*x.^2 + 25*y.^2 - 225;

dF1x = @(x,y) -0.25*(exp(x./2) - exp(-x./2));
dF1y = @(x,y) 1;

dF2x = @(x,y) 18*x;
dF2y = @(x,y) 50*y;

Jac = @(x,y) [dF1x(x,y) dF1y(x,y); dF2x(x,y) dF2y(x,y)];
```

² We will use a relative error tolerance that will be illustrated in the example code.

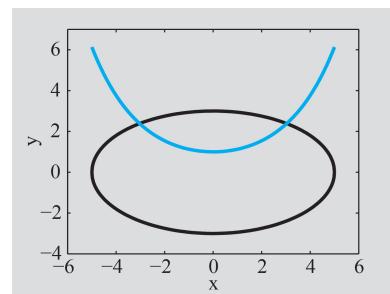


Figure 74: Example system of non-linear equations.

Next we will define x_1 and y_1 and specify our stopping criteria.

```
xi = 2.5; yi = 2;
Err = 1e-10; imax = 10;
```

15
16

Note that we do not plan on making many iterations. If Newton's method works, it will very quickly converge.

Now we implement Newton's method:

```

for i = 1:imax
    J = Jac(xi,yi);
    F = -[F1(xi,yi); F2(xi,yi)];
    dp = J\F; ❶
    xip = xi + dp(1);
    yip = yi + dp(2);
    Err_x = abs((xip - xi)/xi); ❷
    Err_y = abs((yip - yi)/yi);

    fprintf('i = %i x = %-7.4f y = %-7.4f Error in x = %-7.4g
    Error in y = %-7.4g \n', ...
        i,xip,yip,Err_x,Err_y);

    if (Err_x < Err) && (Err_y < Err) ❸
        break
    else
        xi = xip; yi = yip;
    end

end

```

17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

❶ We use MATLAB's built-in "back-slash" operator to solve the linear system of equations. This will be discussed in Section VIII of this text.

❷ Here we compare $\Delta x/x$ and $\Delta y/y$ for our relative error tolerance. We are really not computing the error, we are computing how small our updates are to x and y .

❸ The operator `&&` is the logical *and* operator.

This script converges to the solution $x = 3.0311553917$ and $y = 2.38586565356$ in 5 iterations.

Implementation with FSOLVE

The primary MATLAB built-in function you should use for solving a system of non-linear equations is `fsolve`. The basic syntax for using `fsolve` is:

```
x = fsolve(fun,x0);
```

A more complete syntax that provides additional output information and an interface for passing options to the algorithm is:

```
[x, fval, exitflag, output, jacobian] = ...
    fsolve(fun,x0,options);
```

The values of `x` and `fval` are similar to what one obtains when using `fzero` except, of course they are now both vectors. Values for `exitflag` and `output` are different—interested readers should consult the MATLAB documentation—but `exitflag=1` still means success. As with `fzero`, a function is used to construct an appropriate `options` structure; when using `fsolve` you should use the `optimoptions` function for this purpose. An example usage of `optimoptions` is included in the MATLAB listing below.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

clear
clc
close 'all'

F = @(x) ex3p5(x);
Xo = [2.5, 2.0];

option_set = 2;
% 1 = no outputs
% 2 = detailed outputs

switch option_set      ❸

    case 1
        options = optimoptions('fsolve','Display','none');

    case 2
        options = optimoptions('fsolve','Display',...
            'iter-detailed','MaxIterations',1000,...
            'StepTolerance',1e-10);

    otherwise % default options
        options = optimoptions('fsolve');

end

[x,fval,exitflag,output] = fsolve(F,Xo,options);

fprintf('Root found at x = %8.7g, y = %8.7g \n',...
    x(1),x(2));
fprintf('fval = \n'); disp(fval);
fprintf('exitflag = %d \n',exitflag);

%% Local Function
function out = ex3p5(x)    ❹
[m,n] = size(x); % expect scalar or vector input
assert(min(m,n)==1,...)
    'Error! Vector input expected for x! \n';
out = nan(m,n); % construct output

out(1) = x(2) - 0.5*(exp(x(1)./2) + exp(-x(1)./2));
out(2) = 9*x(1).^2 + 25*x(2).^2 - 225;

end

```

❸ Here we use a `switch ... case` structure to select between sets of options for purposes of demonstration. The options specified in `option_set=1` amount to telling `fsolve()` to quietly solve the problem with no output to the command window. The options specified in `option_set=2` specify detailed output on each iteration and provides non-default values for the '`MaxIterations`' and '`StepTolerance`' parameters. Consult the MATLAB documentation on `fsolve` for more details.

❹ The first argument to `fsolve` has to be a (single) handle to a function. This local function implements the system of two equations from the example. The first argument to `ex3p5` is a vector; one element for each equation in our system of equations.

Part VIII

Solving Linear Equations

Lecture 7 - Introduction to Linear Algebra and Gauss Elimination

Objectives

The objectives of this lecture are to:

- Provide background and describe why engineers what to solve systems of linear equations.
- Describe and demonstrate the Gauss elimination algorithm.
- Do an example in MATLAB.

Background

Linear systems of equations arise in a variety of contexts. For example, consider a simple analysis of a truss structure as shown in Figure 75. If you are to conduct a simple static analysis of this structure, you would need to derive equilibrium equations to show that the sum of forces at each point—A, B, C, D, E, and F—in the x - and y -direction equals zero. A total of eight equations are derived in this way and shown below.

$$\begin{array}{lll}
 -F_{AB} - 0.3846F_{AC} = 3625 & 0.9231F_{AC} = 1690 & \sum F_x, \sum F_y \text{ at A} \\
 F_{AB} - 0.7809F_{BC} = 0 & 0.6247F_{BC} - F_{BD} = 0 & \sum F_x, \sum F_y \text{ at B} \\
 -0.3846F_{AC} - 0.7809F_{BC} - F_{CD} + 0.3846F_{CE} = 0 & & \sum F_x \text{ at C} \\
 0.9231F_{AC} + 0.6247F_{BC} - 0.9231F_{CE} = 0 & & \sum F_y \text{ at C} \\
 F_{CD} + 0.8575F_{DE} = 0 & F_{BD} - 0.5145F_{DE} - F_{DF} = 0 & \sum F_x, \sum F_y \text{ at D}
 \end{array}$$

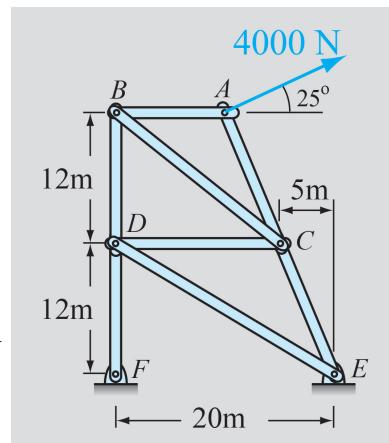


Figure 75: Two-dimensional truss structure.

We cannot, in general, solve these equations individually, the unknown values—the tension in each of the eight truss elements—are distributed among all of the equations. In general we must solve the equations simultaneously; the resulting linear equations are shown in matrix-vector form in Equation 142.

$$\begin{bmatrix} -1 & -0.3846 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.9231 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -0.7809 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.6247 & -1 & 0 & 0 & 0 & 0 \\ 0 & -0.3846 & -0.7809 & 0 & -1 & 0.3846 & 0 & 0 \\ 0 & 0.9231 & 0.6247 & 0 & 0 & -0.9231 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0.8575 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -0.5145 & -1 \end{bmatrix} = \begin{bmatrix} F_{AB} \\ F_{AC} \\ F_{BC} \\ F_{BD} \\ F_{CD} \\ F_{CE} \\ F_{DE} \\ F_{DF} \end{bmatrix} = \begin{bmatrix} 3625 \\ 1690 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (142)$$

A second, and perhaps more generally relevant case, is when a linear differential operator is represented in a discrete form. Consider the boundary value problem expressed in Equation 143.

$$\frac{d^2u}{dx^2} = f(x), \quad 0 < x < L, \quad u(0) = u(L) = 0 \quad (143)$$

As we will study later in this course, the spatial domain $x \in [0, L]$ can be discretized into uniform segments, the differential operator d^2/dx^2 , the solution $u(x)$, and the input function $f(x)$ can all be expressed in a discrete form and assembled into a system of linear equations. This is shown in Equation 144:

$$\underbrace{\frac{1}{h^2} \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & -2 & 1 & 0 \\ \vdots & \cdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}}_{\frac{d^2u}{dx^2}, \quad u(0)=u(L)=0} \underbrace{\begin{bmatrix} u(x_0) \\ u(x_1) \\ \vdots \\ u(x_{n-1}) \\ u(x_n) \end{bmatrix}}_{u(x)} = \underbrace{\begin{bmatrix} 0 \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \\ 0 \end{bmatrix}}_{f(x) \text{ with BC}} \quad (144)$$

where $h = x_{i+1} - x_i$. Depending on how accurate of a solution is desired, the system illustrated in Equation 144 can have thousands of equations or more. For a known $f(x)$, we can use the methods we will describe in this and the next few lectures to solve the equations to find $u(x)$.

In this expression we use the second-order central difference approximation to d^2u/dx^2 which is:

$$\frac{d^2u}{dx^2} = \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{h^2}$$

and a matrix is formed from the resulting system of equations. The first and last row of the matrix along with the first and last element of $f(x_i)$ are modified to satisfy the given boundary conditions. These techniques will be discussed more fully when we present finite difference methods for solving boundary value problems in Lecture 31.

Gauss Elimination

Gauss elimination is the most basic *direct* method¹ for solving systems of linear equations. When it succeeds, the system of equations is solved within a pre-determined number of mathematical operations.

THE BASIC PROCEDURE is done in two steps. First we start with a

¹ In later lectures we will describe *iterative* methods for solving linear systems of equations. In iterative methods we start with an initial guess of the solution vector and iteratively improve upon it. Eventually a solution is obtained but the number of operations required cannot be determined in advance.

system of equations that includes a square, invertible matrix:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and convert it to an upper-triangular matrix using a process called *forward elimination*.

Example: Carry out the process of forward elimination for the 4×4 system of equations shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

In this process, the first row is unchanged and, for the first step, is referred to as the *pivot equation*. The coefficient a_{11} is called the *pivot*. We subtract $m_{21} = a_{21}/a_{11}$ of equation (row) 1 from equation (row) 2:

$$\begin{aligned} a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 && \leftarrow \text{row 2} \\ - \underbrace{\frac{a_{21}}{a_{11}}}_{m_{21}} (a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4) &= m_{21}b && \leftarrow -m_{21} \times \text{row 1} \end{aligned}$$

which equals:

$$0 + \underbrace{(a_{22} - m_{21}a_{12})}_{a'_{22}} x_2 + \underbrace{(a_{23} - m_{21}a_{13})}_{a'_{23}} x_3 + \underbrace{(a_{24} - m_{21}a_{14})}_{a'_{24}} x_4 = \underbrace{b_2 - m_{21}b_1}_{b'_2}$$

The term $a'_{21} = a_{21} - (a_{21}/a_{11})a_{11} = 0$.

thereby eliminating the first term in the second row. This leaves us:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Matrix and vector entries that get modified are annotated with a ('). Additional primes are added for subsequent modifications.

We do the same elimination for subsequent rows to put zeros in the entire first column.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}$$

We will repeat the process for the next column and rows and continue until all entries below the main diagonal are set to zero in this process.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b'_4 \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b''_4 \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b'''_4 \end{bmatrix}$$

eliminate a'_{32} by subtracting $m_{32} = a'_{32}/a'_{22}$ of 2nd row from 3rd row

eliminate a'_{42} by subtracting $m_{42} = a'_{42}/a'_{22}$ of 2nd row from 4th row

eliminate a''_{43} by subtracting $m_{43} = a''_{43}/a''_{33}$ of 3rd row from 4th row

The first phase of Gauss elimination is complete and the system of equations is in upper triangular form.

THE SECOND STEP of Gauss elimination is back substitution. With the equations in upper triangular form, we can solve the system of equations one variable at a time, starting with the last variable, x_4 :

$$x_4 = \frac{b'''_4}{a'''_{44}}$$

Next we can solve for x_3 :

$$\frac{b''_3 - a''_{34}x_4}{a'_{33}}$$

followed by x_2 :

$$x_2 = \frac{b'_2 - (a'_{23}x_3 + a'_{24}x_4)}{a'_{22}}$$

and lastly x_1 :

$$x_1 = \frac{b_1 - (a_{12}x_2 + a_{13}x_3 + a_{14}x_4)}{a_{11}}$$

Hopefully, if you have been following closely, the pattern is clear.² If we had a larger system of equations, we would do the same thing, just more of it.

² The pattern is, for an $n \times n$ matrix:

$$x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

Note that the second term in the numerator can be expressed as a vector dot-product.

MATLAB Implementation

Let us create a MATLAB script to use Gauss elimination to solve the following 4×4 system of linear equations:

$$\begin{bmatrix} 4 & -2 & -3 & 6 \\ -6 & 7 & 6.5 & -6 \\ 1 & 7.5 & 6.25 & 5.5 \\ -12 & 22 & 15.5 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 12 \\ -6.5 \\ 16 \\ 17 \end{bmatrix}$$

We start by clearing out the workspace and providing the problem data.

```
clear
clc
close 'all'

% System to be solved
A = [4, -2, -3, 6;
      -6, 7, 6.5, -6;
      1, 7.5, 6.25, 5.5;
      -12, 22, 15.5, -1];

% right hand side
b = [12;
      -6.5;
      16;
      17];
```

We will implement the Gauss elimination algorithm as a local function, so that part will come last. This next listing will show the code to call our local function and compare our solution result to MATLAB's built-in function for solving linear equations—the backslash operator: `\`.³

```
x = Gauss(A,b);

fprintf('Solution with local function Gauss: \n');
disp(x);

fprintf('Solution with backslash: \n');
x_gold = A\b;
disp(x_gold);

rel_error = norm(x-x_gold,2)/norm(x_gold,2); ❶
fprintf('Relative difference: %g \n', rel_error);
```

❶ In this calculation of the relative error, we make use of a MATLAB built-in function: `norm(v,p)` where `v` is a vector. We do not have space to go into detail here, but a vector norm has the same purpose as a function norm described in Lecture 15 in Section III of this book. The

³ This operator is shorthand for the built-in function `mldivide(A,b)`, where `A` is the matrix and `b` is the right hand side.

Note: Sometimes it is very helpful to write the code you will use to test your function, before you write the function. For one thing, it will help clarify in your mind the interface to your function—i.e. what you will provide as inputs and what you will get as output. It also takes care of the necessary, but sometimes tedious, task of providing a test to confirm that your function works correctly. I cannot overstate the importance of code testing. It is a sad fact that we cannot include in this course a detailed treatment of testing frameworks that are used for large-scale software projects. This is something that, I think, is often missing from the training of undergraduate engineers.

argument p specifies which norm to use. Our choice of p=2 results in use of the Euclidean norm, which many engineers are familiar with as the vector magnitude.

Finally we are ready to implement the Gauss elimination algorithm as a local function. We will start with the function entry through the forward elimination phase.

```
%>>> %% Local function implementing Gauss Elimination
function x = Gauss(A,b)
[n,c] = size(A);
assert(n==c,'Error! A must be square!'); ❷

% forward elimination phase
for j = 1:n-1 ❸
    for i = (j+1):n
        m = A(i,j)/A(j,j); % calculate pivot
        A(i,j:n) = A(i,j:n) - m*A(j,j:n); ❹
        b(i) = b(i) - m*b(j); % update the right hand side
    end
end
```

27
28
29
30
31
32
33
34
35
36
37
38
39
40

❷ We assume that A is square. This is an assumption that we should check.

❸ This loop ends on the second-to-last row since the last row is not a pivot row for any elimination operations.

❹ This is a little bit tricky but not too hard to understand provided you understand the notation: A(j,j:n) which refers to the jth row of A, columns j through n.

At this point, the matrix A should be in upper-triangular form. Now we commence back-substitution.

```
%>>> % back substitution phase
x = nan(n,1); ❺
for i = n:-1:1 ❻
    x(i) = (b(i) - ... ❼
        A(i,(i+1):n)*x((i+1):n))/A(i,i);
end

assert(sum(isnan(x))==0,... ❽
    'Error! An element of x is NaN!');

end
```

41
42
43
44
45
46
47
48
49
50
51

❺ The variable x is constructed as a *column vector*. This is important since it is required for the semantics of the matrix-vector multiplication carried out three lines down to be correct.

❻ Notice that this for ... end loop steps *backward*. Obviously that is required to carry out the back-substitution as described.

❼ Look at this notation carefully. It is a concise articulation of:

$$x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

Note that for i=R, when we index x((i+1):n) *nothing* is returned since, for i=n, i+1=n+1 and thus equivalent to (n+1):n which MATLAB constructs as an empty vector.

❽ It turns out that a lot can go wrong in the update of x(i) and, when everything is done, it is worth checking to see that all members of x have been updated and that none of them are 'NaN'. Throwing an error may be a bit heavy-handed but it is sure, at least, to get the user's attention.

For this example, the script finds and returns the correct solution:

$$x = \begin{bmatrix} 2.0 \\ 4.0 \\ -3.0 \\ 0.5 \end{bmatrix}$$

IT IS WORTH asking: “*what can go wrong*” with Gauss elimination?

1. If $A(j, j) = 0$ then the pivots, calculated on line 35 will be ‘`NaN`’. It may seem that this is an unlikely problem; how likely is it, after all, to have a zero element along the diagonal? As it turns out, in general: quite likely. Most of the matrices that we will be interested in are mostly zeros; only a handful of entries in each row will be non-zero. It so happened that all of the diagonal entries of the matrix shown in Equation 142 are non-zero but that is only because I chose to order the equations and arrange the unknowns such that the diagonals would be non-zero. But the order of the equations is arbitrary as is the ordering of the unknowns so we easily could have a zero in the first row and first column causing Gauss elimination to fail on the first step of elimination.

I hasten to add that the values of the pivots, other than the first one, are not apparent right away since, in general, the values of the soon-to-be-pivots change with each step of forward elimination. You cannot usually tell if a pivot will be zero just by looking at a matrix. Thus, zero-pivots can pop up unexpectedly like a land-mine and derail the algorithm.

2. A pivot does not have to be zero to cause problems. Pivots that are *much smaller* than other entries in a matrix can exacerbate floating point round-off. It can be shown that collectively these errors can cause the Gauss elimination procedure to be numerically unstable.⁴

A resolution to both of these issues will be presented in the next lecture.

⁴ Lloyd N Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 2022

Lecture 8 - Gauss Elimination with Pivoting

Objectives

The objectives of this lecture are to:

- Describe Gauss elimination with pivoting algorithm and provide a rationale for its use.
- Produce a detailed examination of the MATLAB code to implement the algorithm.
- Introduce the “Matrix Market” as a source for relevant test matrices.
- Run examples illustrating the benefits of this new algorithm.

Gauss Elimination with Pivoting

One problem with the basic Gauss elimination algorithm presented in the last lecture is the possibility that the pivot will be zero. In the event of such a pivot, the algorithm will fail and we need to find a way to prevent it from happening if at all possible. A problem that may appear to be less serious, but nonetheless worthy of our attention, is the case where the pivot is much smaller than other entries in its row and/or column. Consider the following linear system:

$$\begin{bmatrix} 0.0003 & 12.34 \\ 0.4321 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 12.343 \\ 5.321 \end{bmatrix}$$

For the first step of forward-elimination, the pivot element is: 0.0003 and $m_{21} = 0.4321/0.0003 = 1,440.33$. When I eliminate the a_{21} entry and update the entry for a_{22} and b_2 :

$$A(i, j : n) = A(i, j : n) - m * A(j, j : n)$$

the value of m is very large compared to any entry in A , and, for lack of a more precise way of putting it, this aggravates round-off errors in the result.

The basic idea is: if the pivot is small or zero, why not just exchange the pivot row with a different row? In order to avoid undoing any of the forward elimination steps, the row to be exchanged must be “below” the current pivot row. Also, if we want to execute such an exchange, we need to have some criteria by which to decide which row we want to exchange. The criteria we will use is this: *select the row corresponding to the element in the pivot column with the largest absolute value*. This is most easily clarified with an example.

Consider the linear system of equations below:

$$\begin{bmatrix} 4 & -2 & -3 & 6 \\ -6 & 7 & 6.5 & -6 \\ 1 & 7.5 & 6.25 & 5.5 \\ -12 & 22 & 15.5 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 12 \\ -6.5 \\ 16 \\ 17 \end{bmatrix}$$

The pivot column is $A(1:4,1)$ and the entry with the largest magnitude is, -12 , in row 4. We would, therefore, *swap row 1 and 4*. The system of equations is now:

$$\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ -6 & 7 & 6.5 & -6 \\ 1 & 7.5 & 6.25 & 5.5 \\ 4 & -2 & -3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 17 \\ -6.5 \\ 16 \\ 12 \end{bmatrix}$$

Notice we also had to switch the entries for $b(1)$ and $b(4)$. We use this pivot to eliminate all of the entries in the first column below the first row. The resulting system of equations is shown below:

$$\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ 0 & -4.0 & -1.25 & -5.5 \\ 0 & 9.333 & 7.542 & 5.417 \\ 0 & 5.333 & 2.167 & 5.667 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 17 \\ -15 \\ 17.417 \\ 17.667 \end{bmatrix}$$

The pivot column for the next step of forward elimination is now $A(2:4,2)$ and the element with the largest magnitude is 9.333 in row 3. We thus swap row 2 and 3 and eliminate the entries in column 2 below row 2.

For step i of forward elimination, $A(i,i)$ is the pivot. $A(i,i:n)$ is the pivot row, and $A(i:n,i)$ is the pivot column.

$$\underbrace{\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ 0 & 9.333 & 7.542 & 5.417 \\ 0 & -4.0 & -1.25 & -5.5 \\ 0 & 5.333 & 2.167 & 5.667 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\text{Swapped rows 2 and 3.}} = \begin{bmatrix} 17 \\ 17.417 \\ -15 \\ 17.667 \end{bmatrix} \rightarrow \underbrace{\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ 0 & 9.333 & 7.542 & 5.417 \\ 0 & 0 & -1.982 & -3.189 \\ 0 & 0 & -2.143 & 2.571 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\text{Eliminate entries below the pivot.}} = \begin{bmatrix} 17 \\ 17.417 \\ -7.536 \\ 7.714 \end{bmatrix}$$

Now we have one more entry to eliminate, but, before we do, we will swap the 3rd and 4th row to obtain the largest magnitude pivot.

$$\underbrace{\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ 0 & 9.333 & 7.542 & 5.417 \\ 0 & 0 & -2.143 & 2.571 \\ 0 & 0 & -1.982 & -3.189 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\text{Swap 3rd and 4th row.}} = \begin{bmatrix} 17 \\ 17.417 \\ 7.714 \\ -7.536 \end{bmatrix} \rightarrow \underbrace{\begin{bmatrix} -12 & 22 & 15.5 & -1 \\ 0 & 9.333 & 7.542 & 5.417 \\ 0 & 0 & -2.143 & 2.571 \\ 0 & 0 & 0 & -0.8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\text{Eliminate the } a_{4,3} \text{ entry.}} = \begin{bmatrix} 17 \\ 17.417 \\ 7.714 \\ -0.4 \end{bmatrix}$$

At this time, we are ready for the back-substitution phase of the algorithm which is unchanged from regular Gauss elimination and, for a small matrix such as in this example, the answer is essentially the same.

MATLAB Implementation

Since the only difference lies in the pivoting, we will restrict our attention to the forward elimination phase of the algorithm.

```
function x = GaussPivot(A,b)
[m,n] = size(A);
assert(m==n,'Error! A must be square!');

for j = 1:n-1
    % select pivot from remaining rows in column j
    [~,j_piv] = max(abs(A(j:n,j))); ①

    % correct row index returned by max()
    j_piv = j_piv + (j-1);          ②

    % swap row j and j_piv, b(j) and b(j_piv)
    % save row to be over-written
    a_row_tmp = A(j,:); b_tmp = b(j); ③
    % over-write with new pivot row
    A(j,:) = A(j_piv,:); b(j) = b(j_piv);
    % replace row
    A(j_piv,:) = a_row_tmp; b(j_piv) = b_tmp;
```

① The built-in function `max()` takes a vector and returns the largest element along with its index. Of course the built-in function `abs()` returns a vector in which all of the elements are converted to their absolute value. The first return argument to `max()` is the actual maximum value but we actually do not need the value; we just need to know which row it is in, which is provided by the return argument `j_piv`. We need both arguments on the right hand side—`[~,j_piv]`—but since we never use the first one, instead of storing the result in a variable, we replace the variable with a tilde (`~`). If we assigned the first argument to a variable but then never used it, MATLAB's code analyzer would issue a warning. This usage avoids such a warning.

② Since we only used a portion of the column of `A` in our call to `max`, we need to adjust the value of `j_piv` to reflect the true row index of the maximum value in the pivot column.

③ This—save, over-write, and replace—idiom for copying/swapping elements frequently appears in algorithms for scientific computing.

```
% introduce zeros in column j below pivot
for i = (j+1):n
    m = A(i,j)/A(j,j);
    A(i,j:n) = A(i,j:n) - m*A(j,j:n); ④
    b(i) = b(i) - m*b(j);
end
end
```

19
20
21
22
23
24
25
26

④ This part of the algorithm is the same as the non-pivoting version.

SOME QUESTIONS THAT you might have right now include:

1. “Is ‘row swapping’ really a valid matrix operation?” The answer is: yes. We can effect a row swap operation, such as for example swapping the first and fourth rows, by multiplying both sides of our linear system of equations by a *permutation matrix*. The details of this matrix will be discussed in the next lecture. While we are on that topic, the operation of “subtracting” a factor of one row from another can also be expressed as a matrix multiplication operation. This will also be discussed in the next lecture.
2. “Isn’t it a lot of work to swap the rows?” The answer is: it can be, but it is worth it. Some high performance libraries manage to replicate the *effect* of swapping the rows without doing all of the work—e.g. by permuting row indices and accessing matrix elements indirectly via such a permuted array of indices. Whatever the case, it turns out that, for real matrices of interest, it is worthwhile to do the extra work because, if we do not, the answer we get will often be entirely wrong.

Matrix Market Test Repository

The Matrix Market is an online repository of matrices that can be used to test and benchmark algorithms for solving linear systems of equations.¹ Example matrices that we have used so far are helpful in illustrating an algorithm and performing simple tests to make sure everything is working. MATLAB also has tools for generation of random matrices that can also be helpful but, if you want to know if the algorithm you have developed will work well on *relevant* matrices, you need a resource like the Matrix Market. With the Matrix Market you can browse through and choose from a selection of hundreds of *actual* linear systems of equations created as part of a real system. The Matrix Market also includes information about test matrices such as whether or not it is symmetric, positive definite, has real or complex entries, and for sparse matrices, the total number of non-zero entries. Matrix Market provides utility routines to allow automatically loading matrix data into the MATLAB environment.

¹ National Institute of Standards and Technology. Matrix Market. URL <https://math.nist.gov/MatrixMarket/>. Accessed on July 18, 2023

We will use some matrices obtained from the Matrix Market to compare the performance of the Gauss elimination and Gauss elimination with pivoting algorithms. The metric we will use to measure performance will be the *relative residual*. The residual is calculated from Equation 145.

$$r = b - Ax \quad (145)$$

The *relative residual* is just the residual divided by the norm of b .

$$\text{Relative Residual} = \frac{\|b - Ax\|_2}{\|b\|_2} \quad (146)$$

If you have the exact solution, x^* , the residual will be zero:

$$Ax^* = b \Rightarrow r = b - Ax^* = 0$$

In general, however, we do not get the exact solution. This is because the elements of A and b are represented with double precision floating point numbers which, in general, are not exact. Each mathematical operation we do with these inexact numbers, incurs additional errors that accumulate throughout the algorithm. We claimed that Gauss elimination without pivoting incurred more of these errors; we will use samples from the Matrix Market to see how differently the algorithms perform on relevant matrices.

1. $A = \text{rand}(50,50)$. Lest you think that tools like Matrix Market might be unnecessary, we will start by using a random 50×50 matrix generated by MATLAB. We compare the relative residual of the solution using three different solvers. The results are shown in Table 4.

Note that, while the relative residual is small in all cases, the relative residual for Gauss elimination without pivoting is *roughly 100 times* as big as that for Gauss elimination with pivoting.

2. $A = \text{rand}(150,150)$. We will try again with a slightly larger matrix and see what happens. The results are shown in Table 5.

Again we see that the relative residual is small, but there is a marked difference in the result with and without pivoting. Let us now use matrices from real applications.

3. LNS 131. This matrix is from the Harwell-Boeing Collection. It is a 131×131 sparse, unsymmetric matrix with real entries that was derived from a simple fluid flow modeling problem. The results are shown in Table 6.

What you see in the table is not a typo. The relative residual from the answer we obtained from Gauss elimination without pivoting

Algorithm	Relative Residual
Gauss Elim.	4.931e-13
Gauss Pivot	3.115e-15
Backslash	2.617e-15

Table 4: Comparison for random 50×50 matrix.

Algorithm	Relative Residual
Gauss Elim.	2.482e-12
Gauss Pivot	2.284e-14
Backslash	2.803e-14

Table 5: Comparison for random 150×150 matrix.

Algorithm	Relative Residual
Gauss Elim.	3.119
Gauss Pivot	1.486e-7
Backslash	1.118e-7

Table 6: Comparison for LNS 131 from the Matrix Market.

is *greater than one*. This means our answer is essentially worthless. The relative residual from Gauss elimination with pivoting and backslash are nearly the same and 8 orders of magnitude smaller. This is a relevant matrix and this should help illustrate why pivoting is so essential.

4. LNS 511. This matrix is also from the Harwell-Boeing Collection. It is, as you may guess, a 511×511 unsymmetric matrix with real entries. The results are shown in Table 7.

You can see from the last two examples that the performance of both Gauss elimination with pivoting and the backslash operator are worse for the bigger matrices. As we will discuss in future lectures, the issue is not that the matrices are bigger, although that does not help; the issue is related to the *condition number* of the respective matrices. The condition number of LNS131 is 9.8×10^9 ; the condition number for LNS 511 is 5.2×10^{10} . We will learn that a higher condition number is related to the accuracy that we can obtain with a numeric solution represented in floating point numbers. A higher condition number means the solution will be less accurate. We see that trend in the last two test cases and we can see that, without the benefits obtained by pivoting, the answer we get from Gauss elimination is essentially worthless for relevant matrices.

Algorithm	Relative Residual
Gauss Elim.	1.276
Gauss Pivot	9.045e-5
Backslash	5.460e-5

Table 7: Comparison for LNS 511 from the Matrix Market.

Assignment #3

1. Consider a non-linear system of equations comprising the following functions:

$$\begin{aligned}f(x, y) &= x^2 + y^2 - 25 = 0 \\g(x, y) &= x^2 - y - 2 = 0\end{aligned}$$

Create a function to carry out Newton's method for finding a root for this system of non-linear equations. The function should have the signature: `Xs = NewtonSystemSol(Fun, Jac, Xo, Tol, MaxIt)` where `Xs` is a root, `Fun` and `Jac` are handles to the system of functions to be solved and the Jacobian matrix, respectively. `Xo` is an initial estimate of the root, `Tol` is the error tolerance and `MaxIt` is the maximum number of iterations to perform. The error tolerance should be based on the relative change of each component of the estimated root. `Tol` should be set to 1×10^{-9} , `MaxIt` should be set to 25, and you should use `Xa = [2.5, 2.5]` to find the root in the upper right-hand quadrant of the coordinate system.

2. A coating on a panel surface is cured by radiant energy from a heater. The temperature of the coating is determined by radiative and convective heat transfer processes. If the radiation is treated as diffuse and gray, the following non-linear system of equations determine the unknowns: J_h , T_h , J_c , and T_c .

$$\begin{aligned}5.67 \times 10^{-8} T_c^4 + 17.41 T_c - J_c &= 5188.18 \\J_c - 0.71 J_h + 7.46 T_c &= 2352.71 \\5.67 \times 10^{-8} T_h^4 + 1.865 T_h - J_h &= 2250 \\J_h - 0.71 J_c + 7.46 T_h &= 11093\end{aligned}$$

Use MATLAB's built-in function `fsolve` to find values of J_h , T_h , J_c and T_c that satisfy this system of equations. Use `optimoptions()` to set the maximum iterations at 50, and the '`StepTolerance`' to 1×10^{-9} . For a starting value, use $T_c = T_h = 298$, $J_c = 3000$ and $J_h = 5000$.

3. Referring to Problem #2 above, if you were to solve that system of equations using Newton's method, you would need to determine the Jacobian matrix. Write down a symbolic version of the Jacobian matrix.

4. Solve the following system of equations using the Gauss elimination method.

$$\begin{aligned} 2x_1 + x_2 - x_3 &= 1 \\ x_1 + 2x_2 + x_3 &= 8 \\ -x_1 + x_2 - x_3 &= -5 \end{aligned}$$

5. The axial force, F_i , in each of the 13-members of the pin-connected truss (add figure) can be calculated by solving the following system of 13 equations:

$$\begin{aligned} F_2 + 0.707F_1 &= 0 \\ F_3 - 0.797F_1 - 2000 &= 0 \\ 0.7071F_1 + F_4 + 6229 &= 0 \\ -F_2 + 0.659F_5 + F_6 &= 0 \\ -F_4 - 0.753F_5 - 600 &= 0 \\ -F_3 - 0.659F_5 + F_7 &= 0 \\ 0.753F_5 + F_8 &= 0 \\ -F_6 + 0.659F_9 + F_{10} &= 0 \\ -F_8 - 0.753F_9 - 800 &= 0 \\ -F_7 - 0.659F_9 + F_{11} &= 0 \\ 0.753F_9 + F_{12} - 2429 &= 0 \\ -F_{10} + 0.707F_{13} &= 0 \\ -F_{12} - 0.7071F_{13} - 600 &= 0 \end{aligned}$$

Lecture 9 - LU Factorization

Objectives

The objectives of this lecture are to:

- Quantify the computational work of Gauss elimination.
- Describe the LU factorization.
- Demonstrate how to use the LU factorization for solving systems of equations.

Computational Work of Gauss Elimination

Throughout this course we have taken a rather crass approach to the amount of work done by the computer. This reflects, in part, a basic reality that for many undergraduate students of engineering, most of the computer-based work comprises time that students spend writing and debugging code. The amount of time that the computer actually spends executing the program in many cases is trivial.

Looking ahead to some of the numerical methods that will be studied in this course, the situation will change. Programs that analyze initial boundary value problems based on the finite difference method or finite element method will need to solve systems of linear or non-linear equations. These systems of equations are often large—it is not unusual to have millions of equations and unknowns—and the time that the computer spends assembling and solving these equations becomes significant. Applications of this type are so important that modern high performance computing systems are regularly benchmarked based on how fast they can solve such problems.¹

In this section will undertake a high-level analysis of the computational work required to perform Gauss elimination. We will break this analysis into two parts: a) the forward elimination phase; and b) the back-substitution phase.

¹ University of Tennessee, Knoxville. HPL Benchmark. URL <https://icl.utk.edu/hpl/index.html>. Accessed on July 19, 2023; and Top500. The Top 500 List. URL <https://top500.org/>. Accessed on July 19, 2023

Forward Elimination

MATLAB code for the forward elimination phase is shown in the listing below:

```

for j = 1:n-1 ❶
    for i = (j+1):n ❷
        m = A(i,j)/A(j,j); % calculate pivot ❸
        A(i,j:n) = A(i,j:n) - m*A(j,j:n); ❹
        b(i) = b(i) - m*b(j); % update the right hand side ❺
    end
end

```

1
2
3
4
5
6
7
8

- ❶ The outer loop, `for j=1:n-1...end`, will be traversed $\mathcal{O}(n)$ times.²
- ❷ The inner loop, `for i=(j+1):n ... end`, also will be traversed $\mathcal{O}(n)$ times.
- ❸ This line performs one floating point operation (FLOP) to calculate the pivot.
- ❹ This line performs $\mathcal{O}(n)$ floating point operations. These include multiplication of a vector by a scalar and vector addition.
- ❺ This line performs two FLOPs: scalar multiplication and scalar addition.

Overall the forward-elimination part of the algorithm requires: $\mathcal{O}(n) \times \mathcal{O}(n) \times \mathcal{O}(n)$ operations to traverse the two nested loops and carry out the operations inside. The asymptotic operation count is thus: $\mathcal{O}(n^3)$.

² Here we adopt “Big oh” notation. It is used to characterize the complexity and running time of an algorithm in asymptotic fashion. So in this case, $\mathcal{O}(n)$ means that, as $n \rightarrow \infty$, the running time of this part of the code increases linearly with n . Small details like constant factors are ignored. It also means that lower order terms get ignored. For example, a function that requires $n^3 + n^2 + n$ calculations would be characterized as $\mathcal{O}(n^3)$ since, as $n \rightarrow \infty$, the n^2 and n terms become insignificant relative to n^3 .

Back-Substitution

MATLAB code for the back-substitution phase is shown in the listing below:

```

x = nan(n,1);
for i = n:-1:1      ❻
    x(i) = (b(i) - ...
              A(i,(i+1):n)*x((i+1):n))/A(i,i);
end

```

1
2
3
4
5

- ❻ This loop is traversed $\mathcal{O}(n)$ times.
- ❼ This line performs $\mathcal{O}(n)$ FLOPS for vector dot product, scalar addition and division.

Overall, the back-substitution part of the algorithm requires: $\mathcal{O}(n) \times \mathcal{O}(n)$ operations; the asymptotic operation count is therefore: $\mathcal{O}(n^2)$.

TAKEN TOGETHER, the entire Gauss elimination algorithm requires $\mathcal{O}(n^3) + \mathcal{O}(n^2)$ operations for a total asymptotic complexity of: $\mathcal{O}(n^3)$. For large enough matrices, if you double the matrix size, the computer will need to perform a factor of $\approx 2^3$ more floating point operations and, roughly speaking, can be expected to take 2^3 times as long. This is shown in Figure 76.

LU Factorization

One problem with the Gauss elimination algorithm as described so far is this: if you want to repeat the solution with a different right-hand-side, you would need to repeat all of the work you had just done. We will not cover it in this class, but one possible solution to this problem is to devise an algorithm like Gauss elimination³ that, in addition to solving the problem for the unknown vector x , also finds A^{-1} , the inverse of A . With the inverse, we could find x for a new right hand side with a simple matrix-vector multiplication.

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b \\ \Rightarrow x &= A^{-1}b \end{aligned}$$

As asymptotic analysis will readily show, matrix multiplication has $\mathcal{O}(n^2)$ complexity, so we can re-compute x for any new b in a small fraction of the time that it would take to re-perform Gauss elimination. We will not do this, however. Instead we will perform a *factorization* of A in such a way that we can solve the linear system, $Ax = b$, with $\mathcal{O}(n^2)$ operations, for any new right hand side, b . This is called the *LU factorization* and it is both faster and more-accurate (in floating point arithmetic) than finding the matrix inverse.⁴

THE LU FACTORIZATION decomposes an invertible matrix into:

- A lower triangular matrix L ; and
- an upper triangular matrix U

such that $A = LU$. It turns out that, while carrying out the Gauss elimination algorithm we actually computed the elements of the LU

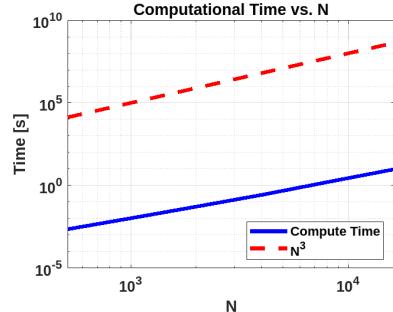


Figure 76: Computing time vs. n for Gauss elimination.

³ The Gauss-Jordan elimination does this.

⁴ Nick Higham. What is the Matrix Inverse? URL <https://nhigham.com/2022/03/28/what-is-the-matrix-inverse/>. Accessed on July 19, 2023

factorization:

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_A = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{21} & 1 & 0 & 0 \\ m_{31} & m_{32} & 1 & 0 \\ m_{41} & m_{42} & m_{43} & 1 \end{bmatrix}}_L \underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix}}_U$$

Where the pivots comprise the elements of L and the transformed coefficient matrix after Gauss elimination is U . In a sense, all we have to do is “write down what we are doing” during the forward elimination portion of Gauss elimination in order to obtain the LU factorization. This is illustrated in the MATLAB code below:

```
function [L,U] = LU_Factor(A)
% LU factorization without pivoting

[m,n] = size(A); % get rows and columns of A
U = A; ①
L = eye(m,n); % constructor for identity matrix.

for k = 1:(m-1)
    for j = (k+1):m
        L(j,k) = U(j,k)/U(k,k); ②
        U(j,k:m) = U(j,k:m) - L(j,k)*U(k,k:m); ③
    end
end
end
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14

① Initialize $U = A$. Carry out operations on U just as you did for A in forward elimination phase of Gauss elimination.

② Compute the pivot and store in L .

③ Carry out elimination step on U .

Now that we have the LU factorization, suppose we want to solve a new linear system with the same matrix A but a new right hand side: c .

$$Ax = c$$

$$LUx = c$$

Let us denote $y = Ux$ and solve:

$$Ly = c$$

Since L is lower triangular, we can solve for y using *forward substitution*. MATLAB code for forward substitution is presented below:

```
function y = ForwardSub(L,c)
n = length(c);
y(1,1) = c(1)/L(1,1);
for i = 2:n
    y(i,1)=(c(i)-L(i,1:(i-1))*y(1:(i-1),1))./L(i,i);
end
end
```

1
2
3
4
5
6
7

Then we can obtain x by solving $Ux = y$ with backward substitution.

MATLAB code for backward substitution is presented below:

```

function x = BackwardSub(U,y)
n = length(y);
for i = n:-1:1
    x(i,1)=(y(i)-U(i,(i+1):n)*x((i+1):n,1))/U(i,i);
end
end

```

1
2
3
4
5
6

which you should recognize as being the same as the backward substitution step used in Gauss elimination.

The MATLAB listing below puts this all together for a simple example:

```

A = [4, -2, -3, 6;
      -6, 7, 6.5, -6;
      1, 7.5, 6.25, 5.5;
      -12, 22, 15.5, -1];

b = [12;
      -6.5;
      16;
      17];

[L,U] = LU_Factor(A);
y = ForwardSub(L,b);
x = BackwardSub(U,y);

%% Local functions
function [L,U] = LU_Factor(A)
% LU factorization without pivoting
[m,n] = size(A); % get rows and columns of A
U = A;
L = eye(m,n); % constructor for identity matrix.

for k = 1:(m-1)
    for j = (k+1):m
        L(j,k) = U(j,k)/U(k,k);
        U(j,k:m) = U(j,k:m) - L(j,k)*U(k,k:m);
    end
end
end

function y = ForwardSub(L,c)
n = length(c);
y(1,1) = c(1)/L(1,1);
for i = 2:n
    y(i,1)=(c(i)-L(i,1:(i-1))*y(1:(i-1),1))./L(i,i);
end
end

function x = BackwardSub(U,y)
n = length(y);
for i = n:-1:1
    x(i,1)=(y(i)-U(i,(i+1):n)*x((i+1):n,1))/U(i,i);
end
end

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43

LU Factorization with Pivoting

As we saw in the last lecture, pivoting is an essential operation if you want to accurately solve relevant matrices. Pivot operations can be captured in a *permutation matrix*. Switching rows in a vector or a matrix is equivalent to multiplying by the identity matrix with rows switched. For example, when pivoting is performed on the matrix example from Lecture 8, the final permuted row ordering is:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \xrightarrow{\text{Swapped } 1\&4} \begin{bmatrix} 4 & 3 & 1 & 2 \end{bmatrix}$$

We can represent this as a permutation matrix, P , constructed as the identity matrix with rows correspondingly reordered.

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

To solve a system of equations with LU-factoring with pivoting:

$PA = LU$	Perform LU factorization on PA .
$b_p = Pb$	Permute b .
$Ly = b_p$	Forward solve for y .
$Ux = y$	Backward solve for x .

In MATLAB code:

```
[L,U] = LU_Factor(P*A);
bp = P*b;
y = ForwardSub(L,bp);
x_p = BackwardSub(U,y);
```

1
2
3
4

In this class we will not implement our own LU-factorization with pivoting and, instead, use MATLAB built-in functions for this task.

Lecture 10 - MATLAB Built-in Methods and Error Estimates

Objectives

The objectives of this lecture are to:

- Describe the most important MATLAB built-in method for finding the direct solution to linear systems of equations.
- Define matrix norms and condition number.
- Introduce an error estimate for solution of linear systems of equations.

MATLAB Built-in Methods

For most applications, the MATLAB function to use for solving linear systems of equations is `mldivide(A,b)` which is most conveniently accessed via the “backslash” operator `\`.

MATLAB’s function `mldivide(A,b)` selects from an array of algorithms depending on the structure of A . A portion of the logic is shown in Figure 77.

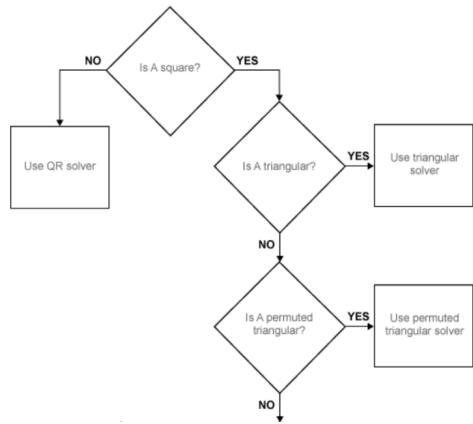


Figure 77: Logic tree for `mldivide()`, part 1.

Most algorithms require A to be square. If A is not square, then A is not invertible and the problem is more aptly described as a least-squares linear fit that we will discuss in later lectures. A “QR solver” carries out the decomposition $A=QR$, where Q is the same size as A but with orthonormal columns; R is $[n \times n]$ and upper triangular.

If A is triangular or permuted triangular, back- or forward- substitution should be used instead of full Gauss elimination or LU factorization.

The remaining logic tree for `mldivide(A,b)` is shown in Figure 78.

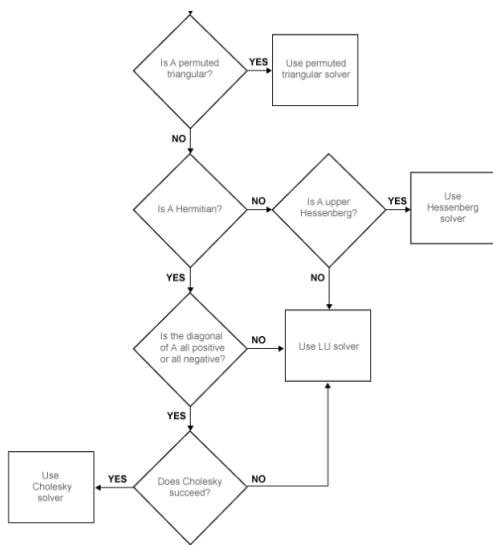


Figure 78: Logic tree for `mldivide()`, part 2.

A *Hermitian* matrix satisfies the equality $A^* = A$ where A^* denotes the transpose and complex conjugate of A . Think of a Hermitian matrix as a symmetric matrix generalized for complex numbers.

If a matrix is Hermitian and if the matrix is positive definite (all positive or all negative entries along the diagonal being a hint that this property is true) then a Cholesky factorization is possible and is roughly twice as fast as an LU factorization. Otherwise an LU factorization is used.

If A is not Hermitian, MATLAB checks if it, perchance, is upper Hessenberg. A matrix is *upper Hessenberg* if it has non-zero entries on the upper triangular portion of A plus is non-zero on the first diagonal below the main diagonal. Matrices of this structure are formed as an intermediate step in matrix eigenvalue calculations. A Hessenberg solver will take advantage of the zeros below the first sub-diagonal and find a solution more expeditiously than a general LU solver.

¹ Lloyd N Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 2022

AS SHOULD BE APPARENT, a *lot* of technology is crammed into the innocent-looking `mldivide` function. We cannot hope to investigate all of these methods in this course. I refer interested readers to an outstanding text by one of my favorite mathematical authors for a more thorough (and rigorous) overview of methods of numerical linear algebra.¹ Instead, I offer the following advice:

1. If the full matrix fits in memory and you only need to solve it once, use `mldivide(A,b)`—or just `A\b`.
2. Use a factorization, or *decomposition*, if you must solve the system multiple times for different values of b where you do not know the values of b in advance.² MATLAB offers a built-in function to do this: `decomposition(A,type)`, where the second (optional) argument is a string to specify the type of decomposition that is desired. These types include: '`lu`', '`qr`', '`chol`', and '`auto`' among others.³ An example listing using this feature is given below.

```

n = 5;
A = rand(n,n);
b1 = rand(n,1); b2 = rand(n,1);

Ad = decomposition(A, 'auto');

x1 = Ad\b1;
x2 = Ad\b2;
  
```

² If you have multiple right-hand-sides and you know them in advance, you can simply arrange them into a matrix, B , and use `mydivide(A,B)`.

³ When you select `type='auto'`, which also happens to be the default if no type is specified, MATLAB determines which decomposition is best.

3. If the matrix is very large, consider an iterative method that we will discuss in the next lecture. In applications, large matrices tend

to be *sparse*—most of the elements are zero. The decomposition of large sparse matrices is often dense, however, so these methods may not be practical. We will focus on this issue in the next lecture.

Matrix Norms and Condition Number

A norm is a real number assigned to a matrix or vector that satisfies the following four properties:

1. $\|x\| \geq 0$ and $\|x\| = 0$ if and only if $x = 0$.
2. $\|\alpha x\| = \alpha \|x\|$ for any constant α .
3. $\|Ax\| \leq \|A\| \|x\|$ where A is a matrix and x is a vector of dimensions conformable with A .
4. For any two vectors x , and y : $\|x + y\| \leq \|x\| + \|y\|$. This is referred to as the “triangle inequality.”

Some common vector and matrix norms are defined in Table 8. The

Vector Norms	Matrix Norms
$\ x\ _\infty = \max_{1 \leq i \leq n} x_i $	$\ A\ _\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n a_{ij} $ max sum of row absolute value
$\ x\ _1 = \sum_{i=1}^n x_i $	$\ A\ _1 = \max_{1 \leq j \leq n} \sum_{i=1}^n a_{ij} $ max sum of column absolute value
$\ x\ _2 = \left(\sum_{i=1}^n x_i ^2 \right)^{1/2}$	$\ A\ _2 = \max \left(\frac{\ Ax\ _2}{\ x\ _2} \right)$ over all vectors x

matrix 2-norm— $\|A\|_2$ —is *induced* from the vector 2-norm. It is meant to characterize the *action* of the matrix A on some vector x . For this book, the most important norm is the vector 2-norm and the induced matrix 2-norm. If the type of norm is not mentioned, you should assume it is a 2-norm.

Matrix Condition Number

The condition number is a useful property of a matrix. It is defined by Equation 147:

$$\kappa(A) = \|A\| \|A^{-1}\| \quad (147)$$

where $\|\cdot\|$ refers to any valid matrix norm. Some important condition number facts⁴ include:

1. The condition number of the identity matrix (of any size) is 1.

Table 8: Common vector and matrix norms.

A generalization of the vector norms presented is the p -norm:

$$\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

As p increases, the value of $\max_{1 \leq i \leq n} |x_i|^p$ gets large relative to all of the other values of $|x_i|^p$; in the limit of $p \rightarrow \infty$ only the largest value $|x_i|^p$ matters and thus we get the “infinity norm”: $\|x\|_\infty$. A less common matrix norm is the Frobenius norm or “taxi-cab” norm:

$$\|A\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}$$

which, if nothing else, has a fun name.

⁴ If you find this to be boring, Google “Chuck Norris Facts” instead.

2. All other matrices have a condition number of 1 or greater.
3. The numerical value of $\kappa(A)$ depends on the norm used in calculating it but, generally speaking, a matrix that has a very large condition number is said to be *ill conditioned*.
4. In the matrix 2-norm, the matrix condition number is equal to:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

where σ are singular values. For real, symmetric matrices, the singular values are equal to the absolute values of the eigenvalues.

Relative Error Bounds

We can use the condition number of a matrix to derive error bounds on the solution to a linear system of equations. As a reminder, for a numeric solution to a linear system of equations, we can define the residual as:

$$r = Ax^{NS} - b$$

where x^{NS} is the numeric solution. The relative residual is:

$$\text{relative residual} = \frac{\|r\|}{\|b\|}$$

Also recall that we define the error as:

$$e = x^* - x^{NS}$$

where x^* is the exact solution. The relative error is:

$$\text{relative error} = \frac{\|e\|}{\|x^*\|}$$

We would like to know what the relative error is but, in general, we do not know the exact solution. What we *can* do is obtain a bound on the relative error based on the condition number and relative residual; both of which we can compute. This error bound is given in Equation 148.

$$\frac{1}{\kappa(A)} \frac{\|r\|}{\|b\|} \leq \frac{\|e\|}{\|x^*\|} \leq \kappa(A) \frac{\|r\|}{\|b\|} \quad (148)$$

We can use this expression to predict an error bound and quantify the impact of attempting to solve an ill conditioned system of equations.

The singular value decomposition (SVD) of a matrix A is given by:

$$A = U\Sigma V^*$$

where U and V^* are orthonormal matrices in which the columns are referred to as left singular vectors and right singular vectors respectively and Σ is a diagonal matrix with the *singular values* of A . The SVD of a real symmetric matrix is related to the eigenvalue decomposition:

$$A^* A = V \Sigma^* \Sigma V^*$$

where Σ is a diagonal matrix with the eigenvalues along the diagonal.

Lecture 11 Sparse Matrices and Iterative Methods

Objectives

The objectives of this lecture are to:

- Describe sparse matrices and their importance in scientific computing.
- Introduce iterative solution methods.

Introduction

Up to this point we have discussed a handful of methods for solving linear systems of equations. We studied Gauss elimination with and without pivoting and also LU factorization which, to be sure, is basically the same as Gauss elimination. We also learned a little bit about the array of solution methods that are used in MATLAB's built-in tools for linear equations such as the QR factorization, LDL and Cholesky factorizations. These are fundamental methods that should be in the toolbox of every engineer.

There are practical situations where these direct solution methods for linear equations are not suitable. Physical conservation laws that are encoded in partial differential equations, discretized and solved using methods like the finite element method (FEM) and finite volume method (FVM) result in large systems of linear equations. A high-resolution simulation will routinely require upwards of 10^5 degrees of freedom¹ to adequately resolve the physics of interest such as: the flow field around an automobile to estimate drag forces; stress and strain distribution within a complex structural component to ensure material failure limits are not exceeded; or the component temperature in the vicinity of a weld process.

Just *storing* the matrices used to represent these linear systems—something, for example, like a $10^5 \times 10^5$ matrix²—requires us to take a new approach from what we have described up to now. Likewise solving the systems to find the unknown vectors—whether the vector

¹ In this context, the number of *degrees of freedom* can be interpreted to refer to the length of a vector describing the unknown function.

² Such a matrix has 10^{10} entries. If each entry is represented with a double-precision floating point number (8 bytes each), that adds up to roughly 80 Gigabytes of memory just to *store* the matrix.

represents temperature, pressure, a velocity component, or a component of material displacement—requires new techniques.

Sparse Matrices

Matrices that arise as part of a finite difference method or finite element method are almost always *sparse*. This means that for every row of the matrix, corresponding to a linear equation pertaining to a single degree of freedom, most of the entries are equal to zero.

As an example, consider a finite element discretization of the transient heat equation on a domain that is rectangular in shape, but with a hole in the middle. The mesh is depicted in Figure 79. The governing equation is shown in Equation 149:

$$\frac{k}{\rho c_p} \frac{\partial u}{\partial t} = \nabla^2 u + S \quad (149)$$

where k is the thermal conductivity, ρ is density, c_p is the specific heat, S corresponds to a constant uniform heat source, u is the temperature, and t is time. In the finite element method, this equation and specified boundary conditions are translated into a linear system of equations:

$$Au = b$$

The non-zero structure of this system of equations can be observed using MATLAB's built-in function `spy(A)`, and the result is shown in Figure 80. The system of equations has 9824 nodes, each with a single degree of freedom. The resulting 9824×9824 matrix A has a total of 67,904 non-zero entries; an average of about 8 non-zero elements per equation.³ This pattern, which is typical for such matrices, can be exploited by only storing and carrying out arithmetic with the non-zero entries of the matrix.

There are a number of sparse-matrix storage formats in use. MATLAB uses the compressed-sparse-column (CSC) storage format.⁴ If there are nnz non-zero entries in a $n \times n$ sparse array, MATLAB stores one (double precision) vector of length nnz with the non-zero values—call this vector: ENTRY; another vector of integers of length nnz —call this vector: ROW—with the row-number for each non-zero; and a third vector of integers of length $n + 1$ —call this vector: COL—that stores the index (from the ENTRY array) of the first non-zero from each column and terminated with $nnz + 1$. An example of this format is shown in Figure 81.

The total storage required is $nnz \times 8$ bytes for the ENTRY array + $nnz \times 4$ bytes for the ROW array + $(n + 1) \times 4$ bytes for the COL array. In asymptotic notation, $\mathcal{O}(nnz + n)$ bytes of storage are required.

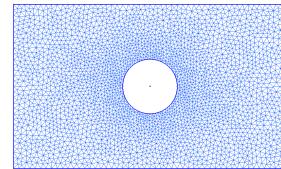


Figure 79: A triangular mesh for a finite element method analysis of the transient heat equation.

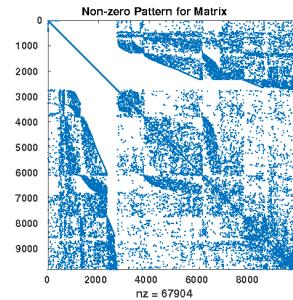


Figure 80: Non-zeros in linear system for transient heat conduction.

³ Bottom Line Up Front: this number of non-zeros per row is typical for two-dimensional systems with linear, triangular elements. The number of non-zeros per equation for the FEM or FVM depends on a number of factors: a) number of degrees of freedom per node; b) the number of spatial dimensions; c) the number of internal degrees of freedom for each finite element, among other factors. This will be discussed in detail in later lectures on finite element methods.

⁴ John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM journal on matrix analysis and applications*, 13(1):333–356, 1992

Matrix:		
$[8 \ 0 \ 0]$	$[1 \ 16 \ 2]$	$[2 \ 0 \ 8]$
ENTRY:	ROW:	COL:
$\begin{bmatrix} 8 \\ 1 \\ 2 \\ 16 \\ 2 \\ 8 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 2 \\ 3 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 4 \\ 5 \\ 7 \end{bmatrix}$

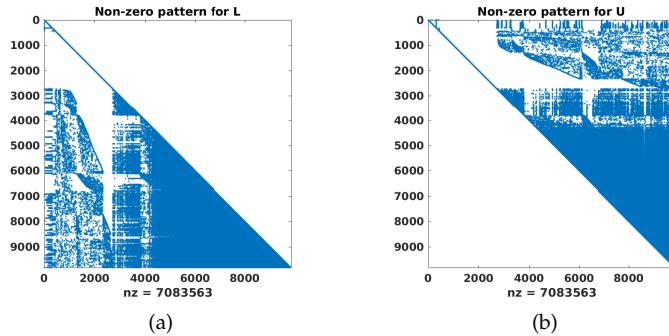
Figure 81: Example sparse matrix in Compressed Sparse Column format.

Contrast this with $\mathcal{O}(n^2)$ for dense matrices. When $nnz \ll n^2$, the savings is huge. The key issues to keep in mind are these:

1. Exploiting the sparsity of these matrices is not a performance enhancement, it is a necessity. Simulations of practical interest often result in systems of equations that can *only* be stored as a sparse matrix.
2. The resulting matrix, if it is to be solved, *cannot* be solved by Gauss elimination related algorithms like LU-factorization.

The reason for the second point is that, even for matrices that are sparse, the LU-factorization (or modified coefficient array for Gauss elimination) for that matrix usually is not sparse. The forward elimination process common to both of the aforementioned techniques systematically destroys the sparsity pattern. This effect is shown in Figure 82.

Note: In terms of FLOPs, sparse matrix calculations are slower. Dense matrix algorithms are heavily optimized to make good use of the memory hierarchy of common computer architectures resulting in high computational intensity—calculations performed for each byte loaded from memory—and consequently achieve high performance. Sparse matrix algorithms have lower computational intensity and the memory access patterns are harder to optimize for multi-threaded execution. As a result, sparse matrix operations are relatively slow. Readers are encouraged to experiment with MATLAB to get a better feel for the relative performance of dense and sparse matrix operations.



For the larger systems of equations that we want to solve for more interesting problems, this kind of fill-in defeats any benefit obtainable through use of sparse matrices.

Iterative Solution Methods

The fundamental idea of iterative methods for solving linear systems of equations is this: given an estimate of the solution, $x^{(k)}$, find a method to generate a new estimate, $x^{(k+1)}$, that is easy to compute and that ultimately converges to the solution of $Ax = b$. Some of the methods that we will discuss in this lecture all involve *splitting* the matrix A into a decomposition $A = M - K$ with M non-singular and

very easy to invert. If this is done, the iterative scheme is as follows:

$$\begin{aligned} Ax &= b \\ (M - K)x &= b \\ \rightarrow Mx &= Kx + b \\ \rightarrow x &= M^{-1}(Kx + b) \\ &= \underbrace{M^{-1}K}_{R}x + \underbrace{M^{-1}b}_{c} \end{aligned}$$

The resulting iterative method is shown in Equation 150:

$$x^{(k+1)} = Rx^k + c \quad (150)$$

where k is the iteration number.

Instead of solving a system of equations by factoring the coefficient matrix, we create what we hope to be a convergent sequence of approximations to the solution by repeated matrix-vector multiplication. Each matrix-vector multiplication operation takes, for dense matrices, $\mathcal{O}(n^2)$ operations. Since the matrix R is sparse and operations with zero entries of the matrix are eliminated, each matrix-vector multiplication takes only $\mathcal{O}(nnz)$ operations which, for sparse matrices, is effectively $\mathcal{O}(n)$.

Jacobi Iteration

In this method, M is defined to be the diagonal of A , and K is the sum of the strictly lower and upper triangular portion of A .⁵

$$M = \text{diag}(A) \quad (151)$$

$$K = \tilde{L} + \tilde{U} \quad (152)$$

where $-\tilde{L}$ is the strictly lower triangular part of A and $-\tilde{U}$ is the strictly upper triangular part of A so that:

$$A = M - (\tilde{L} + \tilde{U})$$

For a diagonal matrix, the inverse is just the same matrix with the diagonal elements inverted:

$$M^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & & \\ & \ddots & \\ & & \frac{1}{a_{nn}} \end{bmatrix}$$

The iterative method is thus expressed:

$$\begin{aligned} x^{(k+1)} &= Rx^{(k)} + c \\ &= M^{-1}(\tilde{L} + \tilde{U})x^{(k)} + M^{-1}b \\ &= M^{-1}b - M^{-1}\tilde{L}x^{(k)} - M^{-1}\tilde{U}x^{(k)} \\ &= M^{-1}[b - \tilde{L}x^{(k)} - \tilde{U}x^{(k)}] \end{aligned}$$

Note: The matrix R plays an important part in the mathematical theory of iterative methods. One relevant result is that if

$$\|R\| < 1$$

in some matrix norm, then the iterative method will converge. To the best of my knowledge, this is *why* we carry out this business with splitting.

⁵ James W Demmel. *Applied numerical linear algebra*. SIAM, 1997

We have manipulated the matrix equations to arrive at a more familiar form of the equation for Jacobi iteration, which is shown in Equation 153:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right] \quad (153)$$

where we express the solution for one element of $x^{(k+1)}$ at a time and the matrix-vector products are shown explicitly. A simple implementation of Equation 153 in MATLAB is shown in the following listing where x_{in} refers to $x^{(k)}$ and x_{new} refers to $x^{(k+1)}$:

```
for i = 1:rows
    x_new(i) = (1/A(i,i))*(b(i) - ... ❶
        A(i,1:(i-1))*x_in(1:(i-1)) - ... ❷
        A(i,(i+1):n)*x_in((i+1):n)); ❸
end
```

- ❶ $x(i) = M^{-1}(b - \dots)$
- ❷ $\dots \tilde{L}x - \dots$
- ❸ $\dots \tilde{U}x)$

The Jacobi method will converge to a solution provided that A is *diagonally dominant*.⁶ A matrix is diagonally dominant if the following relation is true:

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|$$

The main advantages of the Jacobi method are that:

1. it is easy to implement; and
2. it is easy to parallelize. In principle, the elements of $x^{(k+1)}$ can be updated in any order or all at the same time.

In the listing above, we do not take advantage of the ability to parallelize the Jacobi iteration. If we were to do this in MATLAB, we would need to structure the iteration so as to maximize Matrix-vector operations rather than element-wise operations so heavily used above. An improved implementation is shown below:

```
K = -(A - diag(diag(A))); ❹
M_inv = sparse(diag(1./diag(A))); ❺
(Code to set-up and start iteration)

x_new = M_inv*(K*x_in + b);
```

⁶ This is a sufficient condition for convergence but not a necessary condition. The only guarantee is that if A is diagonally dominant, the method will converge.

❹ The MATLAB function `diag(A)` returns a vector with just the elements on the main diagonal of A . The expression `diag(diag(A))` returns a square, diagonal matrix with only the elements on the main diagonal of A .

❺ This line provides a sparse matrix equivalent to M^{-1} .

Both K and M_{inv} are represented as sparse matrices.

The performance improvement from using the vectorized implementation will depend on A but for large matrices, each iteration will run much faster. A performance comparison is shown in Figure 83 for 900×900 matrix run on a typical workstation using MATLAB R2022a. The vectorized version obtains the same answer but was more than 700 times faster. An important take-away from this example is: *how you write your MATLAB code can greatly impact performance*.

```
Command Window
Test matrix with 900 rows, 900 cols
Calculation with simple Jacobi iteration
Iteration: 500, relative update:0.000167804.
Iteration: 1000, relative update:3.47462e-06.
Iterative solution successful!
Residual norm: 7.93753e-07, after 1459 iterations.
Calculation with vectorized Jacobi iteration
Iteration: 500, relative update:0.000167804.
Iteration: 1000, relative update:3.47462e-06.
Iterative solution successful!
Residual norm: 7.93753e-07, after 1459 iterations.
Relative error: 1.29142e-05

Time for simple Jacobi: 16.116 sec
Time for vectorized Jacobi: 0.020949 sec
f1 >>
```

Figure 83: Jacobi method performance comparison for 900×900 test matrix.

The main disadvantage of the Jacobi method is *slow convergence*; compared to other methods, it typically takes many iterations to achieve a solution within a specified error tolerance.

MATLAB Implementation of Jacobi Method

In this section we will present a full implementation of the Jacobi method.

As usual, we start by clearing out the workspace memory and command window output and close any figures. We use a `switch ... case` construct to allow selection between different linear systems.

```
%>> Example: Jacobi Method Demonstration
clear
clc
close 'all'

sys_choice = 2;

switch sys_choice

    case 1
        A = [9 -2 3 2;
              2 8 -2 3;
              -3 2 11 -4;    ①
              -2 3 2 10];
        b = [54.5;
              -14;
              12.5;
              -21];
        rows = 4; cols = 4;
        x_in = zeros(cols,1); ②
        show_x = 1; ③

    case 2
        [A,rows,cols,entries] = mmread('gr_30_30 mtx');
        fprintf('Test matrix with %d rows, %d cols \n', ...
                rows,cols);
        b = rand(cols,1);
        x_in = zeros(rows,1);
        figure
        spy(A)
        title('Sparsity Pattern of Test Matrix')
        show_x = 0;

    otherwise
        error('Invalid system choice!\n');
end
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

① A diagonally dominant test matrix.

② We customarily use a zero vector for $x^{(0)}$. In more complex schemes a simple alternative method is used to obtain a low-order approximation of the solution that can thus be used for $x^{(0)}$. This, unsurprisingly results in faster convergence than the naive choice used here.

③ Here we use `show_x` as a flag to indicate whether or not we want to show the answer in the command window. If I set `show_x=1`, that indicates I want the solution, x , to be output to the MATLAB command window. If `show_x=0` then x is not output. If x is a vector with only a few elements, it is worthwhile to visually inspect the result and see that it is correct. If x has hundreds or thousands of elements, such exercises are useless.

④ This is a 900×900 symmetric, positive definite matrix that has *weak* diagonal dominance. Weak diagonal dominance relaxes the requirement that the magnitude of the diagonal entry be greater than the sum of the absolute values of the off-diagonal entries; it can also be *equal* to the sum of the off-diagonal elements.

We need to specify stopping criteria. For iterative methods such as those under consideration in this lecture, one possible stopping criteria is shown in Equation 154.

$$\text{Estimated Relative Error} = \frac{\|x^{(k+1)} - x^{(k)}\|_\infty}{\|x^{(k)}\|_\infty} \quad (154)$$

This stopping criteria has the disadvantage that it does not provide any real assurance that $x^{(k+1)}$ is close to x^* . It only says that $x^{(k+1)}$ is close to $x^{(k)}$. When evaluating the solution, you should also verify that the residual is small.⁷

The code listing below sets stopping criteria, invokes the Jacobi solver, and evaluates performance.

```
%>> %% Solve using Jacobi Iteration
%
% set stopping criteria
imax = 1500; % max iterations
tol = 1e-7; % estimated relative error tolerance

fprintf('Calculation with simple Jacobi iteration \n');
tic;
[x_jac ,norm_res ,num_iter ,exit_code] = ...    ❸
    jacobi_solver(A,b,x_in,tol,imax);
time_jac = toc;

if exit_code == 1 ❹
    fprintf('Iterative solution converged!\n');
    fprintf('Residual norm: %g, after %d iterations.\n',...
        norm_res,num_iter);
end

if show_x
    fprintf('x = \n'); disp(x_jac); ❺
end

fprintf('Calculation with vectorized Jacobi iteration \n');
tic;
[x_jac_v ,norm_res_v ,num_iter_v ,exit_code_v] = ...
    jacobi_solver_v(A,b,x_in,tol,imax);
time_jacv = toc;

if exit_code_v == 1
    fprintf('Iterative solution converged!\n');
    fprintf('Residual norm: %g, after %d iterations.\n',...
        norm_res_v,num_iter_v);
end

if show_x
    fprintf('x = \n'); disp(x_jac_v);
end
```

As previously discussed, even if the iterative scheme converges, we should check the relative residual to see if we have converged to something like the correct solution. Code to do this is in the next listing.

```
%>> %% Check Relative Residual
%
fprintf('Relative residual Jacobi: %g \n',norm_res); ❻
fprintf('Relative residual vectorized Jacobi: %g \n',...
    norm_res_v);
```

Finally we present the full implementation of the Jacobi iteration. The basic version is in the listing below:

⁷ You might ask: why not just use the relative residual, $\|Ax^{(k+1)} - b\|/\|b\|$, as a stopping criteria? My wan answer is: a lot of people use estimated relative error instead. A somewhat better reason is that if the estimated relative error from Equation 154 is very small, then $x^{(k)}$ is changing only slightly with more iterations and, if the answer is bad at that point, it probably will not get better any time soon. My only advice is that you should check the relative residual when you are done to make sure your iteration did not converge to a garbage solution.

^❸ The MATLAB functions tic and toc act as a simple timing tool. The call to tic starts the clock; when we call toc the clock is stopped and the elapsed time (in seconds) is returned and, in this case, assigned to the variable time_jac.

^❹ Here we make use of the exit_code returned from our function. If exit_code =1, that means the iterative solver stopped based on the estimated relative error.

^❺ Here we make use of the show_x variable. In this context, if show_x is *not* equal to zero, the Boolean expression if show_x evaluates as true and the if ... end block is executed; in this case, displaying x to the MATLAB command window.

^❻ The relative residual calculation is implemented as part of the iterative schemes. It is not used as a stopping criterion but is passed back as a return argument.

```

%% Local functions
function [x_new,norm_res,num_iter,exit_code] = ...
    jacobi_solver(A,b,x_in,tol,imax)
[n,~] = size(A);
rel_update = inf;
x_new = x_in; % initialize x_new
for iter = 1:imax
    if (iter > 1) && (mod(iter,500) == 0)
        fprintf('Iteration: %d, relative update:%g. \n',...
            iter,rel_update);
    end
    for i = 1:n
        x_new(i) = (1/A(i,i))*(b(i) - ...
            A(i,1:(i-1))*x_in(1:(i-1)) - ...
            A(i,(i+1):n)*x_in((i+1):n));
    end
    if norm(x_in,'inf') ~= 0 % prevent nan
        rel_update = ...
            norm(x_new - x_in,'inf')/norm(x_in,'inf');
    end
    % check exit criteria
    if rel_update < tol
        exit_code = 1; % success
        break; % "break out" of the for loop
    end
    if iter == imax
        % maximum iterations reached
        exit_code = 0;
    end
    x_in = x_new;
end
norm_res = norm(A*x_new - b,2)/norm(x_new,2);
num_iter = iter;
end

```

And the, slightly different, vectorized version:

```

function [x_new,norm_res,num_iter,exit_code] = ...
    jacobi_solver_v(A,b,x_in,tol,imax)
rel_update = inf;
x_new = x_in; % initialize x_new
K = -(A - diag(diag(A)));
M_inv = sparse(diag(1./diag(A)));

for iter = 1:imax
    if (iter > 1) && (mod(iter,500) == 0)
        fprintf('Iteration: %d, relative update:%g. \n',...
            iter,rel_update);
    end
    x_new = M_inv*(K*x_in + b);

    if norm(x_in,'inf') ~= 0 % prevent nan
        rel_update = ...
            norm(x_new - x_in,'inf')/norm(x_in,'inf');
    end
    % check exit criteria
    if rel_update < tol
        exit_code = 1; % success
        break; % "break out" of the for loop
    end
end

```

```

if iter == imax
    % maximum iterations reached
    exit_code = 0;
end
x_in = x_new;

end
norm_res = norm(A*x_new - b, 2) / norm(x_new, 2);
num_iter = iter;
end

```

59
60
61
62
63
64
65
66
67
68

Gauss-Seidel Method

The Gauss-Seidel method is a minor modification of the Jacobi method. Recall the basic update scheme for the Jacobi iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right]$$

and notice that, when we are computing the update for $x_i^{(k+1)}$, if we are indeed carrying out these calculations sequentially for increasing values of i , we already have updated values of $x_j^{(k+1)}$ on-hand for $1 \leq j < i$. Why not use them? Answer: there is no reason why not; let us use them. The update scheme for Gauss Seidel is given in Equation 155.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right] \quad (155)$$

The good news is that use of updated values in this way results in the Gauss-Seidel method converging in roughly half as many iterations as the Jacobi method. The bad news is that the order in which we calculate $x_i^{(k+1)}$ matters; we cannot do them in parallel like we could with the Jacobi method.⁸ Still, the performance improvement from the vectorized implementation of Jacobi was *heavily* dependent on the fact that we are working in a MATLAB environment. It would be quite difficult to replicate an equivalent implementation if you were using, for example, C++ or FORTRAN.⁹ So, if you *could not* make use of the inherent parallelism of the Jacobi method, Gauss-Seidel would offer you a significant performance benefit.

Method of Successive Over-Relaxation

If using updated values of $x^{(k+1)}$, as we do in Gauss-Seidel, provides some convergence benefit, maybe we could get more benefit if we used *more* of the updated value. The method of successive over-

⁸ And, perhaps it does not need to be mentioned, given that the vectorized Jacobi method is hundreds of times faster than the non-vectorized version, one may be hard-pressed to give up such an advantage.

⁹ To clarify, it is certainly possible to use C++ or FORTRAN or some other language to implement the Jacobi method in a way that outperforms MATLAB. In MATLAB, the user benefits from libraries of highly optimized algorithms—many of which are, in fact, implemented in lower level programming languages behind the scenes. The point is that the user does not need to know any of that in order to benefit from those libraries. If you are writing your own code, it is incumbent upon you to carry out all of the multi-threaded optimization *or* undertake the non-trivial task of leveraging a high performance library.

relaxation (SOR) does this as shown in Equation 156:

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right] \quad (156)$$

where ω is called a *relaxation parameter*. It can be shown that, in order to converge, $0 < \omega < 2$; if $\omega < 1$, the iteration is referred to as *under-relaxation*; if $1 < \omega < 2$, the iteration is referred as *over-relaxation*.¹⁰ The convergence benefits obtained through SOR is heavily dependent upon properties of the matrix, A , of the linear system you are trying to solve. Readers are encouraged to work through the exercises in the next assignment to get some hands-on experience with this behavior.

¹⁰ If $\omega = 1$, then SOR is equivalent to Gauss-Seidel.

Lecture 12 Preconditioning and MATLAB Built-in Methods

Objectives

The objectives of this lecture are to:

- Further discuss required conditions for convergence of iterative methods based on matrix splitting.
- Discuss preconditioning and its importance for iterative solution methods.
- Describe MATLAB built-in methods for solving sparse systems of equations using iterative methods.

Conditions for Convergence

In the last lecture we discussed three iterative methods based on a matrix splitting: $A = M - K$ where M is non-singular and relatively easy to compute. We used this splitting to try and find solutions to a linear system of equations: $Ax = b$ in an iterative fashion:

$$x^{(k+1)} = \underbrace{M^{-1}K}_{R} x^{(k)} + \underbrace{M^{-1}b}_{c}$$

where $x^{(0)}$ is an initial guess.

It will probably not come as a surprise to learn that we cannot hope to do this successfully for just any matrix A , resulting R , or initial guess $x^{(0)}$.

In the last lecture we learned that if a matrix is strictly row diagonally dominant:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

then Jacobi and Gauss Seidel would converge. This was presented as a *sufficient* condition for convergence—i.e. if true, the method would converge; if not true, the method *might* converge. A more rigorous convergence criterion is based on the spectral radius of $R = M^{-1}K$.¹

¹ James W Demmel. *Applied numerical linear algebra*. SIAM, 1997

Definition 18 (Spectral Radius)

The spectral radius of R is $\rho(R) = \max |\lambda|$, where the maximum is taken over all the eigenvalues, λ , of R .

Theorem 10 (Convergence of Splitting-Based Iterative Method)

The iteration $x^{(k+1)} = Rx^{(k)} + c$ converges to the solution of $Ax = b$ for all starting vectors $x^{(0)}$ and for all b if and only if $\rho(R) < 1$.

It can further be shown that the *rate of convergence*, $r(R)$, giving the increase in the number of correct decimal places in the solution per iteration, can be given by:

$$r(R) = -\log_{10} \rho(R) \quad (157)$$

To quickly summarize the results of this section:

1. For a given splitting-based iterative method, like the Jacobi method, convergence is only assured if: $\rho(R) < 1$; and
2. convergence will require fewer iterations the smaller $\rho(R)$ is.

This begs the question: is there anything we can do if $\rho(R) \geq 1$?

For less dire conditions: if $\rho(R)$ is less than 1 but the rate of convergence is maddeningly slow, is there anything we can do about it?

The answer to both questions is: *yes*, and we describe how in the next section.

Preconditioning

The basic idea of preconditioning is to transform A so that $\rho(R)$ is made smaller. For any non-singular $n \times n$ matrix P , the systems below have the same solution:

$$\begin{aligned} Ax &= b \\ P^{-1}Ax &= P^{-1}b \end{aligned}$$

And so we ask: what choices of P would make $\rho(R)$ smaller? As an extreme choice, set $P = A^{-1}$; then by definition: $P^{-1}A = I$. If this is the case, our matrix splitting is: $P^{-1}A = I = M - K$, where $M = I$, $M^{-1} = I$ and $K = 0$. The corresponding iteration would be:

$$\begin{aligned} x^{(k+1)} &= Rx^{(k)} + c \\ &= M^{-1}Kx^{(k)} + M^{-1}Pb \\ &= \underbrace{I[0]}_{R=0} x^{(k)} + \underbrace{IA^{-1}b}_{A^{-1}b=x} \end{aligned}$$

The spectral radius of R is zero and we converge in one step for any choice of $x^{(0)}$.

Obviously this is not a practical method; if we knew what A^{-1} was, we would not be playing with an iterative method. A more practical approach is to find a P that is *nearly* equal to A^{-1} but easier to compute and store. When working in a MATLAB environment, our main choice for preconditioner will be the *incomplete LU factorization* given by the built-in function `ilu(A,options)`.

As the name suggests, the incomplete LU factorization creates an approximate LU factorization of sparse matrix A . Recall that the reason why we avoid such factorization for sparse matrices is that it results in undesired “fill-in” of the non-zero elements of A . But suppose we carried out the LU factorization but only allowed non-zeros in locations where A is non-zero. In this case, $A \neq LU$ but $U^{-1}L^{-1} \approx A^{-1}$ and this can serve as a basis for an iterative method.

LU and Incomplete LU Preconditioning in MATLAB

To demonstrate preconditioning, we will use a test matrix obtained from the Matrix Market—*BCSSTK15*—which is a real, symmetric, positive definite matrix based on the structural analysis of an off-shore oil platform.² The sparsity pattern is shown in Figure 84.

This matrix is not diagonally dominant³ so we might want to check the spectral radius to determine if the Jacobi method might succeed. MATLAB code to determine spectral radius is provided in the listing below:

```
K = -(A - diag(diag(A)));
M_inv = sparse(diag(1./diag(A)));

rho = abs(eigs(M_inv*K,1)); ❶
fprintf('Spectral radius of R: %g \n',rho);
```

For *BCSSTK15*, $\rho(A) \approx 3.17$, so we expect the un-preconditioned Jacobi method to fail—and, indeed, it does.

Suppose we take an extreme approach and use the complete LU factorization as a preconditioner.

```
[L,U] = lu(A);
nnzL = nnz(L); nnzU = nnz(U);

PA = U\ (L\A); Pb = U\ (L\b); ❷

[x_jac2 , norm_res2 , num_iter2 , exit_code2] = ...
    jacobi_solver(PA,Pb,x_in,tol,imax);
if exit_code2 == 1
    fprintf('Preconditioned Jacobi solution successful! \n');
    fprintf('Number of iterations: %d \n',num_iter2);
    fprintf('tol = %g \n',norm_res2);
end
```

The output is shown in Figure 85. Note the spectral radius of R is near zero and we converged to the solution right away.⁴

² This matrix can be obtained at <https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc2/bcsstk15.html>.

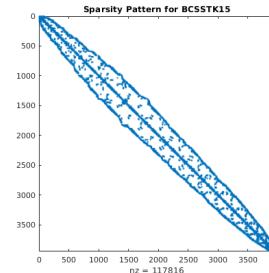


Figure 84: Sparsity pattern for BCSSTK15.

³ You can visit the URL on the Matrix Market for BCSSTK15 to see a variety of matrix properties; diagonal dominance is one such property. You can, of course, alternatively write your own function to determine whether or not a matrix is diagonally dominant.

❶ The built-in function `eigs(A,k)` returns the k -largest eigenvalues of input (sparse) matrix A .

❷ $U\backslash(L\backslash A)$ is equivalent to $U^{-1}L^{-1}A$ and $U\backslash(L\backslash b)$ is equivalent to $U^{-1}L^{-1}b$. Recall, if $A = LU$, $A^{-1} = U^{-1}L^{-1}$.

⁴ The only reason 2 iterations were used is because the stopping criterion was based on $|x^{(k+1)} - x^{(k)}|$ and $x^{(0)}$ was far from the solution.

```
Spectral radius of R: 6.56054e-14
Preconditioned Jacobi solution successful!
Number of iterations: 2
tol = 2.39397e-17
```

Figure 85: Output for preconditioning with *complete LU* factorization.

The bad news for this example is that, while A has 117,816 non-zero entries, L and U both have on the order of a million non-zeros. On top of that, we just performed a full LU factorization of A so time spent on Jacobi iterations were wasted.

Let us now, instead us an incomplete LU factorization.

```

1 opts_ilu.type='ilutp';
2 opts_ilu.droptol=1e-4; ❸
3 opts_ilu.uddiag=1;
4 [iL,iU] = ilu(A,opts_ilu);
5 nnz_iL = nnz(iL); nnz_iU = nnz(iU);
6
7 iPA = iU\ (iL\A); iPb = iU\ (iL\b);❹
8
9 [x_jac3,norm_res3,num_iter3,exit_code3] = ...
10 jacobi_solver(iPA,iPb,x_in,tol,imax);
11 if exit_code3 == 1
12   fprintf('Preconditioned Jacobi solution successful! \n');
13   fprintf('Number of iterations: %d \n',num_iter3);
14   fprintf('tol = %g \n',norm_res3);
15 end

```

❶ This is done for illustration purposes only. Even if iL and iU are sparse with a low number of non-zeros, iPA and iPb will be full. Built-in methods can use iL and iU without sparsity-destroying operations like this. The output is shown in Figure 86. In this case, the spectral radius of R is higher but still less than 1 and, as expected, the number of required iterations to satisfy our solution tolerance is also higher. Unlike as was the case with the complete LU factorization, the output matrices iL and iU are much more sparse, with a total number of roughly 600,000 non-zeros.

If we *increase* the drop tolerance to `opts_ilu.droptol=5e-4` we can further reduce the number of non-zeros in iL and iU to about 400,000 at the expense of increasing the spectral radius to 0.98 and, as expected, increasing the number of iterations required to reach our stopping criterion to 344.

MATLAB Built-In Iterative Solvers

There are numerous iterative solvers built into MATLAB. We will mention only two of them and, sadly, treat them essentially as black-boxes.

1. `pcg` - preconditioned conjugate gradient method.
2. `gmres` - generalized minimum residuals method.

Both of these algorithms are examples of Krylov subspaces methods, the details of which are beyond the scope of this class. We will use `pcg` for linear systems that are symmetric and positive definite. We will use `gmres` for all other methods.

❸ Use these options for `ilu(A,options)`.

The type='ilutp' refers to incomplete LU with *threshold* and *pivoting* which improves the reliability of the algorithm. The droptol is the *drop tolerance* of the incomplete LU factorization. The *higher* your value of droptol, the *more sparse* the resulting L and U will be; if droptol is lower, L and U will be less sparse. In the limit, if droptol=0, then the complete LU factorization is produced. Setting udiag=1 results in replacing zero diagonal entries of U with the local drop tolerance. Selection of this option makes the `ilu` algorithm more reliable. See the MATLAB documentation for `ilu` for a more complete description of all available options.

```

Spectral radius of R: 0.855869
Preconditioned Jacobi solution successful!
Number of iterations: 55
tol = 6.16349e-07

```

Figure 86: Output for Jacobi iteration with incomplete LU preconditioning.

Preconditioned Conjugate Gradient

The preconditioned conjugate gradient algorithm is one of the most competitive methods for use with sparse, symmetric matrices that are positive definite. An excellent and easy to read description is available in the open literature.⁵ Since A is symmetric, we can use a slightly more efficient algorithm to construct the preconditioning matrix—the incomplete Cholesky factorization: `ichol(A,options)`. An example in its use is shown in the listing below:

```
%>%% preconditioned conjugate gradient
1 opts.type = 'ict';
2 opts.droptol = 1e-4; ⑥
3 opts.michol = 'on';
4 L = ichol(A,opts);
5 [x1,f1,r1,it1,rv1] = pcg(A,b,tol,imax,L,L'); ⑦
6
7 if f1 == 0
8   fprintf('pcg with ichol preconditioner solution successful!\n');
9   fprintf('Residual norm: %g, after %d iterations.\n',...
10        r1,it1);
11 end
12
```

⁵ Jonathan Richard Shewchuk. Conjugate gradient method without the agonizing pain. *School of Computer Science, Carnegie Mellon University Pittsburgh, 1994*

⑥ See the MATLAB documentation for `ichol` for more information on these options. The option: `type='ict'` directs usage of the incomplete Cholesky with threshold dropping. This option along with `droptol` affects the extent to which non-zeros in L are dropped or retained. As with `ilu()`, `droptol=0` results in a full Cholesky factorization. The option `michol='on'` indicates that the modified incomplete Cholesky factorization is to be performed, use of which improves the reliability of the algorithm.

⑥ Several of MATLAB's built-in preconditioners use *left* and *right* preconditioners. As we have described them so far, we have always use a *left* preconditioner. A right preconditioner works the same way but the matrix is applied to the right:

$$APx = bP$$

A preconditioning matrix P can be broken up into a left and right preconditioning matrix: $P = P_L P_R'$ and applied:

$$P_L A P_R' x = P_L b P_R'$$

The last two arguments for `pcg` correspond to the left and right preconditioning matrix respectively.

Readers are encouraged to experiment with the preconditioned conjugate gradient built-in function with different options settings to solve a sparse, symmetric, positive definite, linear system of their choice.

Generalized Minimum Residuals

You should use GMRES for systems that are square and non-singular. Since the input matrix is not necessarily symmetric or positive definite, you should use `ilu` to generate the preconditioning matrices. An example usage is shown in the listing below.

```
%>%% GMRES with Incomplete LU Preconditioner
1 opts_ilu.type='ilutp';
```

```
opts_ilu.droptol=1e-3;
restart = [];
maxit = min(imax, size(A,1));
[L,U] = ilu(A,opts_ilu);
[x4,f14,rr4,it4,rv4] = ...
gmres(A,b,[],tol,maxit,L,U);
```

7 Please see the MATLAB documentation for explanation regarding the arguments to gmres

Assignment #4

1. Determine the LU decomposition of the matrix below by hand.

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 5 & 1 \\ 6 & -2 & 2 \end{bmatrix}$$

2. Carry out (by hand) the first three iterations of the solution of the following system of equations using the Gauss-Seidel iterative method. For the first guess of the solution, take the value of all unknowns to be zero.

$$8x_1 + 2x_2 + 3x_3 = 51$$

$$2x_1 + 5x_2 + x_3 = 23$$

$$-3x_1 + x_2 + 6x_3 = 20$$

3. Consider the linear system given below:

$$\begin{bmatrix} 4 & 0 & 1 & 0 & 1 \\ 2 & 5 & -1 & 1 & 0 \\ 1 & 0 & 3 & -1 & 0 \\ 0 & 1 & 0 & 4 & -2 \\ 1 & 0 & -1 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 32 \\ 19 \\ 14 \\ -2 \\ 41 \end{bmatrix}$$

- (a) Find the solution using the built-in MATLAB function $[L,U,P] = \text{lu}(A)$ —be sure to get and use the permutation matrix P .
- (b) Find the relative residual for your solution.
- (c) Find the inverse of the matrix, A^{-1} , using the built-in MATLAB function $\text{inv}(A)$, and compute the 2-norm of A and A^{-1} using the built-in MATLAB function $\text{norm}(A,p)$, where p should be set to 2 for the 2-norm.
- (d) Calculate the size of the error bound:

$$\frac{1}{\kappa(A)} \frac{\|r\|}{\|b\|} \leq \frac{\|e\|}{\|x^*\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$$

where $\kappa(A)$ is the condition number of A , and r is the residual.

- (e) Repeat steps a) through d) for the test matrix BCSSTK26 from the Matrix Market. Once you have read the matrix into MATLAB, convert the matrix to a “full” (non-sparse) format using the MATLAB built-in function $A_{\text{full}} = \text{full}(A_{\text{sparse}})$. or the right-hand-side vector b , use a vector of all ones. Note how different the error bound is in this case.

4. Using Jacobi, Gauss-Seidel, and SOR ($\omega = 1.4$) iterative methods, write and run code to solve the following linear system of equations:

$$\begin{bmatrix} 7 & 3 & -1 & 2 \\ 3 & 8 & 1 & -4 \\ -1 & 1 & 4 & -1 \\ 2 & -4 & -1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ -3 \\ 1 \end{bmatrix}$$

The stopping criterion should be the relative change of the estimated solution in the 2-norm:

$$\text{tolerance} = \frac{\|x^{(k+1)} - x^{(k)}\|_2}{\|x^{(k)}\|_2}$$

with tolerance set to 10^{-9} . Compare the number of iterations required in each case.

5. For this problem we will compare two iterative methods to solve BCSSTK26—a test matrix derived from a seismic simulation of a nuclear power plant structure. For the right-hand-side vector b , use a vector of all ones.
- Use the SOR method with an estimated relative error tolerance of 10^{-3} . Vary the relaxation parameter over the range $\omega \in [1.25, 2.0]$. Make a plot of the number of iterations required as a function of ω and approximate the best value of ω for this system.
 - Use the MATLAB built-in function `gmres` without a preconditioner to solve this problem.
 - Use `gmres` again but with an incomplete LU preconditioner. Use `opts_ilu.type='ilutp'` and `opts_ilu.dtol = dtol` and vary the value of $dtol$ over the range $dtol \in [10^{-7}, 10^{-3}]$. What happens as $dtol$ gets larger?
Hint: Use the built-in function `spy(L)` to show the sparsity pattern of L and note the number of non-zeros of L . How does `nna(L)` change as $dtol$ is changed?
 - This is a relatively small sparse system and can be solved using direct methods. Use the MATLAB built-in function `timeit` to estimate the time required to solve the system of equations using the built-in `mldivide()`—“backslash”—function.

Organize your findings from part a), b) and c) into a short written document. A couple of paragraphs should be sufficient. Include the plot you created from part a) and any other graphics/tables that you find helpful to communicate what you observed in parts b) through d).

Review #1

List of Topics

1. Mathematical preliminaries:
 - (a) Number representation.
 - i. Unsigned integer representations.
 - ii. IEEE-754 floating point representation.
 - (b) Sources of error.
 - (c) Background of Linear Algebra.
 - i. matrix/vector operations.
 - ii. matrix properties.
2. Solution of Non-linear Equations.
 - (a) Bisection method.
 - (b) Newton's method (single equation and system of equations).
 - (c) Secant method.
 - (d) Steffen's method.
 - (e) MATLAB built-in methods (single equation and system of equations).
3. Solution of Linear System of Equations.
 - (a) Gauss elimination with and without pivoting.
 - (b) LU factorization with and without pivoting.
 - (c) MATLAB built-in methods.
 - (d) Error bounds for solution of linear systems of equations.
 - (e) Iterative methods:
 - i. Jacobi, Gauss-Seidel, SOR.
 - ii. MATLAB built-in methods including: pcg and gmres with preconditioning.

Review Questions

1. Write down the sign bit, exponent bits, and first 6 bits of the mantissa for IEEE-754 double precision floating point representation of the number 2023.

2. Write down the number 81 in 32-bit unsigned integer format (little-endian layout).
 3. Carry out 2 iterations of the secant method to find the root of $f(x) = x - 2e^{-x}$ starting with $x_1 = 0$ and $x_2 = 1$. (i.e. find x_3 and x_4)

4. Consider the following system of nonlinear equations:

$$\begin{aligned}-2x^3 + 3y^2 + 42 &= 0 \\ 5x^2 + 3y^3 - 69 &= 0\end{aligned}$$

Using Newton's method for a system of equations and starting at $x = y = 1$ write the linear system of equations you would need to solve to find Δx and Δy to update the estimated solution.

5. Using MATLAB's built-in function: $[L,U,P] = \text{lu}(A)$ function, find the solution to the linear system of equations shown below:

$$\left[\begin{array}{cccccc} 0 & 3 & 8 & -5 & -1 & 6 \\ 3 & 12 & -4 & 8 & 5 & -2 \\ 8 & 0 & 0 & 10 & -3 & 7 \\ 3 & 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & -6 & 0 & 2 \\ 3 & 0 & 5 & 0 & 0 & -6 \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 34 \\ 20 \\ 45 \\ 36 \\ 60 \\ 28 \end{bmatrix}$$

Part IX

Curve Fitting and Interpolation

Lecture 13 Least Squares Curve Fitting

Objectives

The objectives of this lecture are to:

- Derive the basic formulas for least squares curve fitting.
- Do an example using MATLAB.

Introduction

As an engineer, it is very likely that at some point in time during your career, you will be called upon to examine and interpret experimental data. Two common needs in such analysis are to take a set of experimental data and either:

1. develop a model to represent the data. i.e. find a best fit line (or other function) through the data; or
2. evaluate the data to determine how well it agrees with some previously defined model. i.e. fit model parameters to the data and assess how well the experimentally determined values conform to model expectations.

As an example, we will perform a data-fitting analysis of some wind-tunnel results as presented in a popular numerical methods textbook.¹ The behavior that we will explore is the dissipation of vortices shed from the tips and trailing edges of an airfoil in the wind tunnel. For this example, the tangential velocity (V_θ) of a vortex is measured as it travels down the axis of a wind-tunnel. The data is non-dimensionalized by dividing the tangential velocity of the vortex by the free-stream velocity (V_∞) and by dividing the vortex position (R) relative to the airfoil by the chord-length (C) of the airfoil. This non-dimensionalization process will allow experimenters to correlate the results from one particular experiment to larger scale tests on geometrically similar prototypes. The raw data is given in Table 9 and is shown graphically in Figure 87.

¹ Gilat, Amos and Subramaniam, Vish. *Numerical Methods for Engineers and Scientists: an Introduction with Applications Using MATLAB*. Wiley, Hoboken, NJ, third edition, 2014

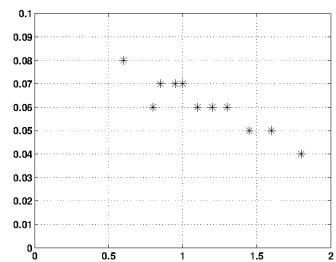


Figure 87: Experimental data from wind tunnel testing. The y -axis is the ratio of the tangential velocity of a vortex to the free stream flow velocity ($y = V_\theta/V_\infty$). The x -axis is the ratio of the distance from the vortex core to the chord of an aircraft wing. ($x = R/c$).

$x = V_\theta/V_\infty$	0.6	0.8	0.85	0.95	1.0	1.1	1.2	1.3	1.45	1.6	1.8
$y = R/C$	0.08	0.06	0.07	0.07	0.07	0.06	0.06	0.06	0.05	0.05	0.04

In the following sections, we will explore ways in which curves can be fit through this data which, in some sense, are the “best”-fit curves.

“Guessed”-fit Curves

It is entirely reasonable, and completely in accord with time honored engineering tradition, to take experimental data as presented in the previous section, use careful judgment and intuition and draw a line that seems to fit the data reasonably well. We will call this the *Guessed-fit* curve and an example of this is shown in Figure 88.

From this carefully drawn line we may conclude that experimental results show a linear relationship between tangential velocity and distance downstream from the airfoil. Obviously, this model is not perfect; most data-points are off the line. Still, we may reasonably decide that overall, this is not a bad representation of what the data are telling us and leave well enough alone.

Measure of Fitness

A hand-drawn curve may be well enough for rough analysis, but for the purposes of this lecture, let us assume that we would like to know how good our roughly drawn curve is and wonder if there may be a way to do better. We have a good fit; but how good is it? In this section we will answer that question. We will define a *measure of fitness* so that we may quantitatively determine how “good” a candidate curve is in representing the data.

For this purpose, we define the *residual*. In words, the residual (r_i) is the difference, at each experimental data point (x_i), between the y -value given from experimental data (y_i) and the y -value computed from our linear “guessed” fit curve (y_{guessed}). The mathematical expression for this is given in Equation 158.

$$r_i = y_i - \underbrace{(b + Mx_i)}_{y_{\text{guessed}}} \quad i = 1, 2, \dots, n \quad (158)$$

where b is the y -intercept of this linear fit and M is the slope of the linear fit through the data and n is the number of data points.

With an eye towards a more general approach, we will re-state Equation 158 using matrix-vector notation in Equation 159.

Table 9: Numerical data from wind-tunnel experiment.

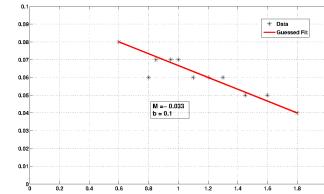


Figure 88: *Guessed-fit* linear estimation of the experimental data. $M = -0.033$ is the measured slope of the estimated line and $b = 0.1$ is the y -intercept.

$$\begin{aligned}
 \mathbf{r} &= \mathbf{y} - \underbrace{\left(b \cdot \mathbf{x}^0 + M \cdot \mathbf{x}^1 \right)}_{\mathbf{y}_{\text{guessed}}} \\
 &= \mathbf{y} - \begin{bmatrix} \mathbf{x}^0 & \mathbf{x}^1 \end{bmatrix} \begin{bmatrix} b \\ M \end{bmatrix} \\
 &= \mathbf{y} - \mathbf{X}\mathbf{c}
 \end{aligned} \tag{159}$$

To be clear, please note that x^k refers to element-wise exponentiation and in particular, \mathbf{x}^0 means: “each element of \mathbf{x} raised to the power of zero,” and \mathbf{x}^1 means: “each element of \mathbf{x} raised to the first power.”

In order to determine how well a given curve fits the data, the residual given in Equation 159 is not quite satisfactory; it is a vector, not a number. The usual solution to this problem is to use the Euclidean length of the residual as shown in Equation 160:

$$\begin{aligned}
 ||\mathbf{r}|| &= \sqrt{\mathbf{r}^T \mathbf{r}} \\
 &= \sqrt{\sum_{i=1}^n r_i \cdot r_i}
 \end{aligned} \tag{160}$$

For historical reasons, we will depart from this convention and use the “Euclidean length squared,” or simply the square of the residual. We show this in Equation 161 where we also explicitly expand \mathbf{r} as defined in Equation 159 to show the residual in terms of the given data \mathbf{y} , the matrix \mathbf{X} of our fitting curve and parameter vector \mathbf{c} .

$$\begin{aligned}
 \mathbf{r}^2 &= \mathbf{r}^T \mathbf{r} \\
 &= (\mathbf{y} - \mathbf{X}\mathbf{c})^T (\mathbf{y} - \mathbf{X}\mathbf{c}) \\
 &= \mathbf{y}^T \mathbf{y} - 2\mathbf{c}^T \mathbf{X}^T \mathbf{y} + \mathbf{c}^T \mathbf{X}^T \mathbf{X}\mathbf{c}
 \end{aligned} \tag{161}$$

The error measure given in Equation 161 now is a single non-negative number that will in general be zero only if the fitted line passes exactly through all data points. This is the measure of fitness that we will use. Using the given values of \mathbf{y} , \mathbf{X} and \mathbf{c} for the “guessed”-fit curve, we find that $(\mathbf{r}_{\text{guessed}})^2 = 0.0152$.

Method of Least Squares

So far we have naively attempted to fit the data as best as we can by guessing a linear function that might in some way represent the data. We have defined an error measure that confirms our suspicion that

our linear curve fit is not perfect. It is natural to wonder: is there a line that *best* fits the data² and if so, how do we find it? The answer is “*yes, there is a way to find the best fit line*” and the method to find it is called the method of least squares.

Algebraic Derivation

The standard algebraic derivation of the method of least squares starts with the squared residual given in Equation 161. As you should take a moment to confirm, once we have selected a linear estimator³ \mathbf{X} , the only free parameter in Equation 161 are the coefficients that make up \mathbf{c} . The goal is to figure out how to choose \mathbf{c} such that the error given in Equation 161 is as small as possible.

Recall from your introductory calculus courses that the way to minimize a function is to take the first and second derivative of the function; solve for the values of the free parameter (\mathbf{c}) so that the first derivative is equal to zero; and verify that the second derivative is positive. When the first derivative is zero, the function is at an extremum; when the second derivative is positive, that extremum is a minimum.

Carrying out this idea, we will take the derivative of Equation 161 and set the first derivative equal to zero:

$$\begin{aligned} \frac{d}{d\mathbf{c}} \mathbf{r}^2 &= 0 - 2\mathbf{X}^T \mathbf{y} + \underbrace{\mathbf{X}^T \mathbf{X} \mathbf{c} + \mathbf{c}^T \mathbf{X}^T \mathbf{X}}_{\mathbf{X}^T \mathbf{X} \mathbf{c} = \mathbf{c}^T \mathbf{X}^T \mathbf{X}} = 2(-\mathbf{X}^T \mathbf{y} + \mathbf{X}^T \mathbf{X} \mathbf{c}) = 0 \\ &\Rightarrow -\mathbf{X}^T \mathbf{y} + \mathbf{X}^T \mathbf{X} \mathbf{c} = 0 \\ &\Rightarrow \mathbf{X}^T \mathbf{X} \mathbf{c} = \mathbf{X}^T \mathbf{y} \end{aligned} \quad (162)$$

We now have to ask: can we find a unique vector \mathbf{c} such that the last line in Equation 162 is satisfied? The answer is: yes—provided only that the columns of \mathbf{X} are linearly independent,⁴ $\mathbf{X}^T \mathbf{X}$ will be positive-definite and thus non-singular.⁵ This means that a unique value of \mathbf{c} will exist and that it will be non-zero:

$$\mathbf{c} = (\mathbf{X}^T \mathbf{X})^{-1} (\mathbf{X}^T \mathbf{y}) \quad (163)$$

MATLAB code to carry out this process is given below:

```
x = [0.6 0.8 0.85 0.95 1.0 1.1 1.2 1.3 1.45 1.6 1.8];
y = [0.08 0.06 0.07 0.07 0.07 0.06 0.06 0.06 0.05 0.05 0.04];

X = nan(length(x), 2);
X(:, 1) = (x') .^ 0;
X(:, 2) = (x') .^ 1;

c = (X' * X) \ (X' * y');
```

² At least “best” by some error measure. Different error measures sometimes yield different answers as to what constitutes “the best.”

³ i.e. we have chosen what functions will be used to make up the columns of \mathbf{X} —for the time being we decided it would be composed of the 0th and 1st powers of x

⁴ If the columns of \mathbf{X} are linearly independent, this means—by definition—that $\mathbf{X}\mathbf{y} = 0$ if and only if $\mathbf{y} = 0$.

⁵ When a matrix— $\mathbf{A} = \mathbf{X}^T \mathbf{X}$ —is positive definite, that means that $\mathbf{y}^T \mathbf{A} \mathbf{y} = 0$ if and only if $\mathbf{y} = 0$. So $\mathbf{y} \neq 0$ and if the columns of \mathbf{X} are linearly independent ($\mathbf{X}\mathbf{y} \neq 0$), then $\mathbf{y}^T \mathbf{X}^T \mathbf{X}^T \mathbf{y} \geq 0$ and can only be equal to zero if $\mathbf{y} = 0$. As is discussed in previous lectures, positive-definiteness is a sufficient condition for a solution to Equation 162 to exist.

Executing this code with our given data we solve for \mathbf{c} which gives us the y -intercept and slope of a different linear curve for the data which we will tentatively call the “best” linear fit. The resulting line is presented along with the previous “guessed”-fit curve for comparison in Figure 89. Using Equation 161, we find that the squared residual of this solution is: 0.0140 which is slightly better than our previous “guessed” estimate of 0.0152.

The second step is to prove that the coefficient array \mathbf{c} really is a minimum and not a maximum or saddle-point. One answer is to say that if we, once again, take the derivative of Equation 162, we get:

$$\begin{aligned} \frac{d^2}{d\mathbf{c}^2} \mathbf{r}^2 &= \frac{d}{d\mathbf{c}} \left(-\mathbf{X}^T \mathbf{y} + \mathbf{X}^T \mathbf{X} \mathbf{c} \right) \\ &= \mathbf{X}^T \mathbf{X} \end{aligned} \quad (164)$$

The problem with this, is that the last line of Equation 164 is not simply a number; it is a matrix. It turns out that if the columns of \mathbf{X} are linearly independent, then the square matrix $\mathbf{X}^T \mathbf{X}$ is symmetric and positive definite. The property of a matrix: “symmetric, positive definite” carries with it some implications:

1. all of the eigenvalues of the matrix are real and positive; and
2. the matrix is invertible

Though it may smack of hand-waving, the author requests your indulgence and accept that these properties carry the same implications as a positive second derivative for the residual function. The curve found via the method of least squares is, in fact, the curve with the minimum residual; not an inflection point and definitely not the maximum.⁶

Based on the mathematical results of this section we can assert that no *linear* estimator of this data set can achieve a squared residual error of less than 0.0140.

Linear Least Squared with Non-Linear Estimator

So far, we have accomplished much, but what do we do in the case where we do not expect the x and y data that we collected in our experiment to be linearly related? The answer is a straight-forward extension of the method of least squares presented in the preceding section.

Suppose we would like to find the best quadratic fit through the data? That is, we are seeking some function: $c_1 + c_2 x + c_3 x^2$ such that the squared residual is as small as possible. All that we need to do is

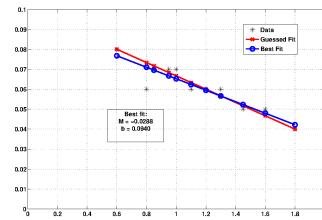


Figure 89: Best fit linear estimation of the experimental data. $M = -0.0288$ is the measured slope of the estimated line and $b = 0.0940$ is the y -intercept.

⁶ The existence of the “guessed”-fit curve with a higher residual than the curve found using the method of least squares should be convincing proof that the method of least squares, at least, does not find the curve with the maximum residual.

to re-define \mathbf{c} to accommodate the extra parameter and re-define \mathbf{X} to incorporate the extra functional dependence on x . Specifically:

$$\begin{aligned}\mathbf{c} &= \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}^T \\ \mathbf{X} &= \begin{bmatrix} x^0 & x^1 & x^2 \end{bmatrix}\end{aligned}\quad (165)$$

We use this newly defined \mathbf{X} and solve for the corresponding values of \mathbf{c} using Equation 163 *exactly as before.*(!!) Nothing in the process need change because, just as with the linear estimator, all that is required is that the columns of \mathbf{X} be linearly independent. To highlight how general this concept is, we will again change our notation slightly and write \mathbf{X} as:

$$\mathbf{X} = \begin{bmatrix} f_1(x) & f_2(x) & \cdots & f_k(x) \end{bmatrix} \quad (166)$$

where here k is the index of the columns of \mathbf{X} . For the linear case, $f_1(x) = 1, f_2(x) = x$. For the quadratic case, we simply add $f_3(x) = x^2$. Each of the functions: f_1, f_2 and f_3 are linearly independent.⁷ Taking this a step further, we could use *any* set of linearly independent functions and Equation 163 would still have a unique solution that would provide the parameter vector \mathbf{c} such that the squared residual will be as small as possible.

The process of using high order polynomials to fit data is so common, that MATLAB has a built-in function—polyfit—to automate least-squares estimation with polynomial estimators. The code block below does this for second order, third order and sixth order polynomials. The resulting estimators are given in Figure 9o.

```

x = [0.6 0.8 0.85 0.95 1.0 1.1 1.2 1.3 1.45 1.6 1.8];
y = [0.08 0.06 0.07 0.07 0.07 0.06 0.06 0.06 0.05 0.05 0.04];
cSecond = polyfit(x,y,2);
ySecond = cSecond(1)*x.^2 + cSecond(2)*x + ...
    cSecond(3);
cThird = polyfit(x,y,3);
yThird = cThird(1)*x.^3 + cThird(2)*x.^2 + ...
    cThird(3)*x + cThird(4);
cSixth = polyfit(x,y,6);
ySixth = cSixth(1)*x.^6 + cSixth(2)*x.^5 + ...
    cSixth(3)*x.^4 + cSixth(4)*x.^3 + cSixth(5)*x.^2 + ...
    cSixth(6)*x + cSixth(7);
```

Model Estimators

As can be seen from Figure 9o, it is possible to find estimators that greatly reduce the squared residual.⁸ As higher order estimators are used, the resulting curve through the data becomes problematic. For example, how would one justify the “curvy” nature of the 6th-order

⁷ Here again a definition is worthwhile. For a set of functions to be linearly independent it implies that for scalar values $c_1, c_1f_1(x) + \cdots + c_kf_k(x) = 0$ if and only if $c_1 = \cdots = c_k = 0$. All of the monomials: $1, x, x^2$, etc... as easily seen to be linearly independent.

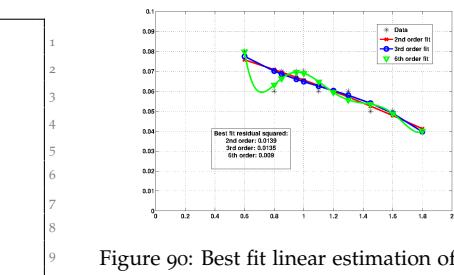


Figure 9o: Best fit linear estimation of the experimental data using 2nd order, 3rd order and 6th order estimators.

⁸ In general it is possible to find a polynomial that exactly interpolates any number of data points with the resulting residual equal to zero. It turns out, doing this with polynomial interpolants is usually a very bad idea—at least with ordinary monomials and with equally spaced data points—and when using non-exact floating point arithmetic, and a large number of data points is involved, it is practically impossible. Better interpolation methods with non-equally-spaced points are almost always used where very accurate interpolation is necessary (for example: when implementing high order finite element methods).

approximator? If we re-performed the experiment with improved instrumentation and made our measurements more carefully, do we *actually* expect the data to follow the 6th order curve? Probably not.

It is also worthwhile to consider that the purpose of doing all of this least squares estimation is *not* always to find some curve that comes close to interpolating all of the data. Rather, the purpose may be to fit the data to a rational scientific model where the experimental data either confirms and strengthens the proposed model for the phenomena under consideration, or serves as the basis for creation of a new model.⁹

For this example, it turns out that theoretical models do exist regarding the expected vortex velocity as it travels downstream from an airfoil in a wind-tunnel. This model predicts that the relationship between y —the ratio between the vortex tangential velocity and the free-stream velocity—and x —the ratio of the distance from the vortex core to the chord of the airfoil section—should have the form of Equation 167.

$$y = \frac{A}{x} + \frac{Be^{-2x^2}}{x} \quad (167)$$

As before, we can develop an estimator that conforms with this model in exactly the same way we did for the polynomial estimators. MATLAB code accomplishing this is provided in the code block below:

```
X = nan(length(x),2);
f1 = @(t) (1./t); % functional form of first term
f2 = @(t) exp(-2*t.^2)./t; % functional form of second term
X(:,1) = f1(x');
X(:,2) = f2(x');
cModel = X\y';
```

⁹ *Approximation Theory and Approximation Practice*. Society of Industrial and Applied Mathematics, Philadelphia, PA, 2013

Please note a couple of differences between this code and code previously provided:

1. Anonymous functions are used to simplify the construction of \mathbf{X} . This is a stylistic choice but could be useful for cases where you want to automate this process.
2. The usual “backslash” left division operator was used instead of the formula specified in Equation 163. The reason this works is because MATLAB automatically checked and noted that \mathbf{X} was not a square matrix (normally a requirement for left matrix division); seeing it was not square, then it checked to see if each column of \mathbf{X} is linearly independent; MATLAB determined that they were and used the equivalent of Equation 163 to solve for \mathbf{c} .¹⁰

The resulting curve, along with the linear estimator is shown in Figure 91.

¹⁰ The normal equations is the easiest method to derive, but actually solving the normal equations directly turns out not to be the very best way of finding \mathbf{c} .

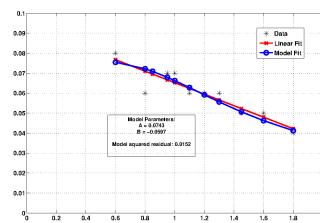


Figure 91: Best fit curve with theoretical model parameters. The linear Least Squares estimator is shown for reference.

As we can see, the residual squared for the theoretical model estimator is larger than the residual squared for the linear estimator shown previously. Remember, the purpose of fitting a curve through the data points in this context is to gain insight as to how well the recorded experimental data can fit against a known (or proposed) theoretical model. If we wanted a residual of zero, we could have gotten it through the mechanical process of providing an exact interpolant through the data; but that would hardly yield a reasonable model. Even though the residual of this new model is higher than for a linear estimator, we gain the benefit of seeing how well the theoretical model stands up in the face of experimental evidence.

MATLAB Example

In an electrophoretic fiber-making process, the diameter of the fiber, d , is related to the current flow, I . The following measurements are made during production:

I (nA)	300	300	350	400	400	500	500	650	650
d (μm)	22	26	27	30	34	33	33.5	37	42

Table 10: Process data from electrophoretic fiber-making process.

1. Use linear least-squares regression to determine the coefficients m and b in the function $y = mx + b$ that best fits the data.
2. Use least-squares regression to determine the coefficients a , b , and c in the function $y = a + bx + cx^2$.
3. Use least-squares regression to determine the coefficients a and b in the function $y = a + b\sqrt{I}$.

We start, as always, by clearing out the workspace and command window and closing any open figure windows. We will also input the given data.

```
clear
clc
close 'all'

%% Input Data
I = [300 300 350 400 400 500 500 650 650]';
d = [22 26 27 30 34 33 33.5 37 42]';
```

The linear least-squares regression is carried out in the following code; the best fit line is shown in Figure 92

```
X = [ I.^0 I.^1];
b = d;

c = (X'*X)\(X'*b);

linEst = @(x) c(1) + c(2)*x;
```

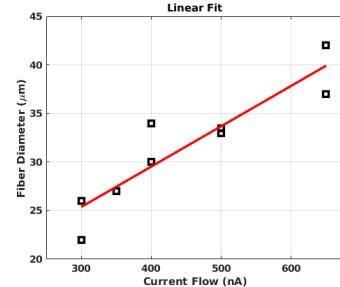


Figure 92: Linear fit, $m = 0.416$, $b = 12.913$.

A least-squares regression to fit a second-order polynomial to the data is carried out in the next listing.

```
X = [I.^0 I.^1 I.^2];
b = d;
c = (X'*X)\(X'*b);
quadEst = @(x) c(1) + c(2)*x + c(3)*x.^2;
```

A plot of the resulting estimator is shown in Figure 93.

Using an estimator like $y = a + b\sqrt{I}$ is accommodated in exactly the same way; there is a constant term—proportional to I^0 —and a term proportional to $I^{1/2}$. The MATLAB code is shown in the listing below and the resulting estimator, along with the previously found estimators, is shown in Figure 94.

```
X = [I.^0 I.^0.5];
b = d;
c = (X'*X)\(X'*b);
est3 = @(x) c(1) + c(2)*x.^0.5;
```



Figure 93: Second-order polynomial fit. $a = 0.436$, $b = 0.0979$, and $c = -5.89e - 5$.

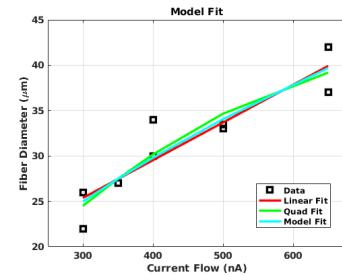


Figure 94: Model fit of data. $a = -6.20$, $b = 1.80$.

Lecture 14 Curve Fitting with Non-linear Functions

Objectives

The objectives of this lecture are to:

- Explain how to carry out least squares curve fitting with a non-linear equation.
- Do an example using MATLAB.

Curve Fitting with Nonlinear Equation

The method of least squares requires the estimator to be a linear combination of functions—although, in most cases the individual functions are not linear. If the estimator is non-linear, then you need to linearize it, if possible, through an appropriate transformation. A variety of useful transforms are presented in Table 11.¹

¹ Gilat, Amos and Subramaniam, Vish. *Numerical Methods for Engineers and Scientists: an Introduction with Applications Using MATLAB*. Wiley, Hoboken, NJ, third edition, 2014

Table 11: Transforming nonlinear equations to linear form.

Nonlinear equation	Linear Form	Relationship to $Y = a_1X + a_0$	Values for linear least-squares regression
$y = bx^m$	$\ln(y) = m \ln(x) + \ln(b)$	$Y = \ln(y), X = \ln(x), a_1 = m, a_0 = \ln(b)$	$\ln(x_i)$ and $\ln(y_i)$
$y = be^{mx}$	$\ln(y) = mx + \ln(b)$	$Y = \ln(y), X = x, a_1 = m, a_0 = \ln(b)$	x_i and $\ln(y_i)$
$y = b10^{mx}$	$\log(y) = mx + \log(b)$	$Y = \log(y), X = x, a_1 = m, a_0 = \log(b)$	x_i and $\log(y_i)$
$y = \frac{1}{mx+b}$	$\frac{1}{y} = mx + b$	$Y = \frac{1}{y}, X = x, a_1 = m, a_0 = b$	x_i and $\frac{1}{y_i}$
$y = \frac{mx}{b+x}$	$\frac{1}{y} = \frac{b}{m} \frac{1}{x} + \frac{1}{m}$	$Y = \frac{1}{y}, X = \frac{1}{x}, a_1 = \frac{b}{m}, a_0 = \frac{1}{m}$	$\frac{1}{x_i}$ and $\frac{1}{y_i}$

We will use these transformations in the examples that follow.

Example #1: Data are provided in the table below.

x	1	2	3	5	8
y	0.8	1.9	2.2	3	3.5

Table 12: Table of data for Example #1.

Determine the coefficients m and b in the function $y = [m\sqrt{x} + b]^{1/2}$ that best fits the data.

Solution: Here, right off the bat, we have a case that is not represented in Table 11. Nonetheless, we will persevere and notice without too much difficulty that if I make the transformation $p = y^2$, then the estimator for p is given by: $p = m\sqrt{x} + b$. MATLAB code to load the data and calculate the coefficients of the, now, linearized, estimator is provided below.

```

1 clear
2 clc
3 close 'all'

4 % data
5 x = [1 2 3 5 8]';
6 y = [0.5 1.9 2.2 3 3.5]';

7 % linearized estimator
8 p = y.^2;
9 X = [x.^0 x.^0.5];

10 C = (X'*X)\(X'*p);

11 %remember to "undo" the linearizing transformation
12 est1 = @(x) sqrt(C(1) + C(2)*sqrt(x));
13
14
15
16

```

Notice how we needed to apply the inverse of the linearizing transformation to recover the desired estimator. Results are shown in Figure 95.

Example #2: Consider the following given data.

x	-2	-1	0	1	2
y	1.5	3.2	4.5	3.4	2

Note: Both of the vectors for x and y are constructed so as to be *column vectors*. We follow this practice for the other examples as well so that we can employ the same MATLAB equations for carrying out least squares regression and satisfy the semantics of each linear algebraic operation.

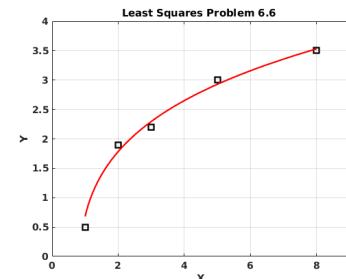


Figure 95: Plot of least squares estimator for Example #1.

Table 13: Table of data for Example #2.

Determine the coefficients a and b in the function $y = \frac{a}{x^2+b}$ that best fit the data.

Solution: The form of this non-linear estimator is similar to that

presented in the 4th row of Table 11.

$$\begin{aligned} p &= \frac{1}{y} \\ &= \frac{x^2 + b}{a} \\ &= \frac{x^2}{a} + \frac{b}{a} \\ &= c_1 x^2 + c_2 \end{aligned}$$

where $c_1 = 1/a$ and $c_2 = b/a$. We implement the linear least squares in MATLAB in the, by now, familiar style:

```

clear
clc
close 'all'

% data
x = [-2 -1 0 1 2]';
y = [1.5 3.2 4.5 3.4 2]';

% linearized estimator
X = [x.^2 x.^0];
b = y.^(-1);

C = (X'*X)\(X'*b);

% remember to "undo" the linearizing transformation
a = 1./C(1); ①
b = C(2)*a; ②

```

A plot of the resulting estimator is shown in Figure 96.

Example #3: Water solubility in jet fuel, W_s , is a function of temperature, T , and can be modeled by an exponential function of the form:

$$W_s = b e^{mT}$$

Table 14 presents measured values of water solubility over a range of temperatures.

T ($^{\circ}\text{C}$)	-40	-20	0	20	40
W_s (%wt.)	0.0012	0.002	0.0032	0.006	0.0118

Using linear least squares, determine the constants m and b that best fit the data.

Solution: The form of this non-linear estimator is similar to that presented in the 2nd row of Table 11. To linearize the estimator we

① Since $c_1 = 1/a$, then $a = 1/c_1$.

② Here $c_2 = \frac{b}{a}$ so

$$\begin{aligned} c_2 a &= \frac{b}{a} a \\ &= b \end{aligned}$$

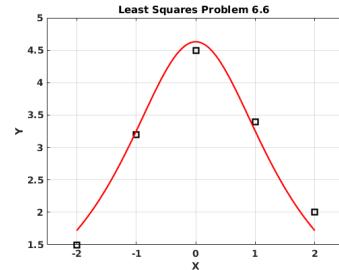


Figure 96: Plot of least squares estimator for Example #2.

Table 14: Table of data for Example #3.

first take the natural logarithm of both sides:

$$\begin{aligned}\ln W_s &= \ln b e^{mT} \\ \ln W_s &= \ln b + \ln e^{mT} \\ &= \ln b + mT \\ &= c_1 T^0 + c_2 T^1\end{aligned}$$

where $c_1 = \ln b$ and $c_2 = m$. We carry out the linear least squares process as usual.

```
clear
clc
close 'all'

% data
T = [-40 -20 0 20 40]';
W = [0.0012 0.002 0.0032 0.006 0.0118]';

X = [T.^0 T.^1];
p = log(W);

C = (X'*X)\(X'*p);

b = exp(C(1)); %❸
m = C(2);

est3 = @(x) b*exp(m*x);
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

❸ Since $c_1 = \ln b$ we undo the transformation by exponentiating both sides.

The resulting estimator is plotted against the given data in Figure 97.

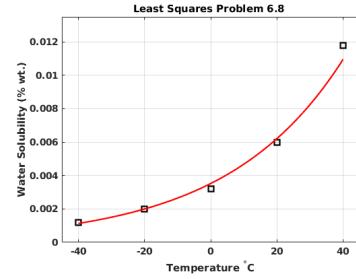


Figure 97: Plot of least squares estimator for Example #3.

Assignment #5

1. Create a user-defined function for linear regression. The signature should be: [a,Er] = LinReg(x,y). In addition to determining the constants a_0 and a_1 for a linear least-squares fit to the data, the function should also calculate the squared residual:

$$Er = \sum_{i=1}^n [y_i - (a_1 x_i + a_0)]^2$$

The input arguments x , and y are vectors with the values of the data points. Use the function to find the coefficients for a linear least squares fit to the following data and find the error.

x	1	3	4	6	9	12	14
y	2	4	5	6	7	8	11

2. The following are measurements of the rate coefficient, k , for the reaction $\text{CH}_4 + \text{O} \rightarrow \text{CH}_3 + \text{OH}$ at different temperatures T .

T (K)	595	623	761	849	989	1076	1146	1202	1382	1445	1562
$k \times 10^{20}$ ($\text{m}^3 \cdot \text{s}$)	2.12	3.12	14.4	30.6	80.3	131	186	240	489	604	868

Use the method of least squares to best fit a function of the form:

$$\ln(k) = A + b \ln(T) - \frac{E_a}{RT}$$

This is derived as a linearization of the Arrhenius equation:

$$k = AT^b e^{-E_a/RT}$$

where A and b are constants, $R = 8.314 \text{ J/mole/K}$ is the universal gas constant, and E_a is the activation energy for the reaction. Determine the values of A (m^3/s) and E_a (J/mole) in the Arrhenius expression.

3. The following data is given:

x	0.2	0.5	1	2	3
y	3	2	1.4	1	0.6

By hand, determine the coefficients m and b in the function $y = \frac{1}{mx+b}$ that best fit the data using linear least squares fit.

4. The resistance, R , of a tungsten wire as a function of temperature can be modeled with the equation:

$$R = R_0 [1 + \alpha (T - T_0)]$$

where R_0 is the resistance corresponding to temperature T_0 , and α is the temperature coefficient of resistance. Determine R_0 and α such that the equation will best fit the data presented below. Use $T_0 = 20^\circ \text{C}$.

$T (\text{ }^\circ\text{C})$	20	100	180	260	340	420	500
$R (\Omega)$	500	676	870	1060	1205	1410	1565

5. In a uni-axial tension test, a dog-bone-shaped specimen is pulled in a machine. During the test, the force applied to the specimen, F , and the length of a gage section, L , are measured. The true stress, σ_t , and the true strain, ϵ_t , are defined by:

$$\sigma_t = \frac{F}{A_0} \frac{L}{L_0} \quad \text{and} \quad \epsilon_t = \ln \frac{L}{L_0}$$

where A_0 and L_0 are the initial cross-sectional area and gage length, respectively. The true stress-strain curve in the region beyond the yield stress is often modeled by:

$$\sigma_t = K \epsilon_t^m$$

The following are values of F and L measured in an experiment. Determine the values of the coefficients K and m that best fit the data. The initial cross-sectional area and gage length are $A_0 = 1.25 \times 10^{-4} \text{ m}^2$, and $L_0 = 0.0125 \text{ m}$.

$F (\text{kN})$	24.6	29.3	31.5	33.3	34.8	35.7	36.6	37.5	38.8	39.6	40.4
$L (\text{mm})$	12.58	12.82	12.91	12.95	13.05	13.21	13.35	13.49	14.08	14.21	14.48

Lecture 15 Interpolation with Lagrange Polynomials

Objectives

The objectives of this lecture are to:

- Discuss the problem of interpolation and illustrate how it can be done with linear least squares.
- Introduce Lagrange polynomials as a preferable strategy for interpolation.
- Illustrate the techniques with a MATLAB example.

Interpolation with Least Squares

Interpolation is a lot like curve fitting except that we expect the estimator to be *exact* at the given data points. In principle this can be done with linear least squares. For any given n data points, we can construct a polynomial interpolant of degree $n - 1$ that matches the given data exactly.

Example: Create a 4th order interpolant for the following 5 data points using linear least squares.

x	1	4	7	10	13
y	2	6	4	8	10

MATLAB code to carry out this process is provided in the listing below:

```
clear
clc
close 'all'
% Data
x = [1 4 7 10 13]';
y = [2 6 4 8 10]';
% Create least squares interpolant
N = length(x);
X = x.^0:N; ①
c = (X'*X)\(X'*y);
nthInterp = @(x) (x.^0:N)*c; ②
```

1
2
3
4
5
6
7
8
9
10
11

① This is slightly tricky but the effect is to create a matrix, X , whose columns comprise x^n for $n \in [0, 1, 2, 3, 4, 5]$.

② This is also tricky; note, in particular, that the x in this line of code is different from the x in the data section. The effect is to create an interpolant of the form:

$$\text{nthInterp}(x) = c(1) + c(2)x^1 + \cdots + c(n)x^{n-1}$$

The data and least squares interpolant are presented in Figure 98.

Notice that, while the interpolant—as required—manages to pass through all of the data points, it is not an altogether great representation of the overall trend of the data.¹ Some other problems that are inherent to this approach:

1. As n increases, the columns of X are more “like” each other. As a result, the condition number of $(X^T X)$ increases and, as we learned in Lecture 10, there is, as a consequence, greater numerical error in solving the linear system to find the coefficients. In fact, when I execute this code in MATLAB, an error is issued due to the high condition number of $(X^T X)$:

```
Command Window
Warning: Matrix is close to singular or badly scaled. Results
may be inaccurate. RCOND =  5.746925e-24.
fx >>
```

2. Solving a set of linear equations for the interpolating coefficients is inconvenient for some applications. Working in a MATLAB environment we might forget that, generally speaking, solving a linear system of equations is complicated.

For applications later in the course, such as finite element methods, we will be very picky about the quality of interpolation functions and we will not be particularly interested in solving systems of linear equations on each domain that we hope to use such interpolations. Luckily, there are better methods that avoid both of these problems.

Lagrange Polynomial Interpolation

Suppose we have 3 data points— x_1 , x_2 , and x_3 —and we want to derive a 2nd order polynomial interpolant, and we specify the polynomial as follows:

$$f(x) = y = a_1(x - x_2)(x - x_3) + a_2(x - x_1)(x - x_3) + a_3(x - x_1)(x - x_2)$$

Notice the form of this polynomial; when we evaluate $f(x)$ at x_1 , the first term on the right is the only non-zero term; when we evaluate $f(x_2)$, only the second term on the right is non-zero and similarly for when we evaluate $f(x_3)$. This makes it easy to find values for a_1 , a_2 and a_3 . This process is shown below for finding a_1 :

$$\begin{aligned} f(x_1) &= y_1 \\ &= a_1(x_1 - x_2)(x_1 - x_3) + \cancel{a_2(x_1 - x_1)}^0(x_1 - x_3) + \cancel{a_3(x_1 - x_1)}^0(x_1 - x_2) \\ \Rightarrow a_1 &= \frac{y_1}{(x_1 - x_2)(x_1 - x_3)} \end{aligned}$$

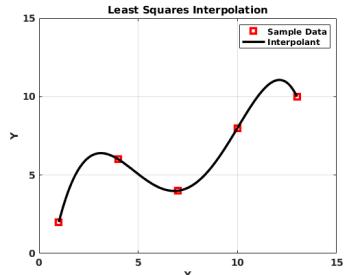


Figure 98: Least squares interpolation of data points.

¹ I suspect that you would not want to really use this interpolant to estimate values between the data points.

Figure 99: Warning issued by MATLAB due to the high condition number of $(X^T X)$.

We can derive equations for a_2 and a_3 in the same way:

$$a_2 = \frac{y_2}{(x_2 - x_1)(x_2 - x_3)}$$

$$a_3 = \frac{y_3}{(x_3 - x_1)(x_3 - x_2)}$$

Thus we have specified, what we will call, a Lagrange polynomial through these points:

$$f(x) = \frac{(x - x_2)(x - x_3)y_1}{(x_1 - x_2)(x_1 - x_3)} + \frac{(x - x_1)(x - x_3)y_2}{(x_2 - x_1)(x_2 - x_3)} + \frac{(x - x_1)(x - x_2)y_3}{(x_3 - x_1)(x_3 - x_2)}$$

In general, we construct the Lagrange polynomial as shown in Equation 168:

$$f(x) = \sum_{i=1}^n y_i L_i(x) = \sum_{i=1}^n y_i \underbrace{\prod_{\substack{j=1 \\ j \neq i}}^n \frac{(x - x_j)}{(x_i - x_j)}}_{\text{Lagrange function}} \quad (168)$$

While this formulation avoids the aforementioned problems, it still can result in a low-quality interpolant. It turns out that the quality of the interpolant depends on the number of points one hopes to interpolate and *the spacing* of these points. In particular, uniformly spaced interpolating points results in low quality interpolants.

Consider, as an example, following function:

$$f(x) = \frac{8a^3}{x^2 + 4a^2}$$

where a is a parameter. This is the so-called Witch of Agnesi² problem and it is a classic test case for interpolation schemes.³ If we set $a = 0.15$ and construct a Lagrange interpolant with uniformly spaced points, we get the result shown in Figure 100. The interpolation improves as n increases everywhere except near the endpoints of the domain. At the endpoints, the interpolation becomes worse.

If instead of using uniformly spaced points, we choose particular set of non-uniformly spaced points, the quality of the interpolation is much improved. In Figure 101 we select *Chebyshev nodes* for interpolation.⁴ To be sure, there are many contexts in which the analyst, who is devising the interpolation scheme, is not at liberty to choose the sample points. In cases where you can choose the sample points, know that how you choose your points makes a difference.

Matlab Example

In this example, we will illustrate a case where the user does *not* have a choice in the sample points. Nonetheless, it will be worthwhile to

² Witch of Agnesi. URL https://en.wikipedia.org/wiki/Witch_of_Agnesi

³ Approximation Theory and Approximation Practice. Society of Industrial and Applied Mathematics, Philadelphia, PA, 2013

⁴ Chebyshev nodes, on the interval $x \in [-1, 1]$ are given by:

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right), k = 1, \dots, n$$

or for an arbitrary interval $x \in [a, b]$ via the following mapping:

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2k-1}{2n}\pi\right), k = 1, \dots, n$$

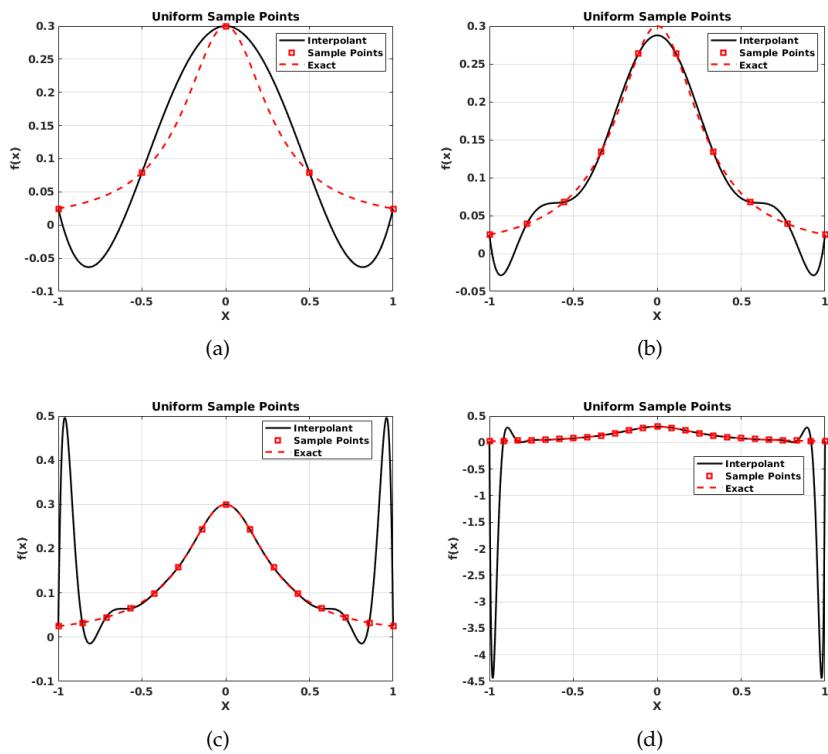


Figure 100: Lagrange interpolation with $n = 5$, $n = 10$, $n = 15$, and $n = 20$ uniformly spaced points.

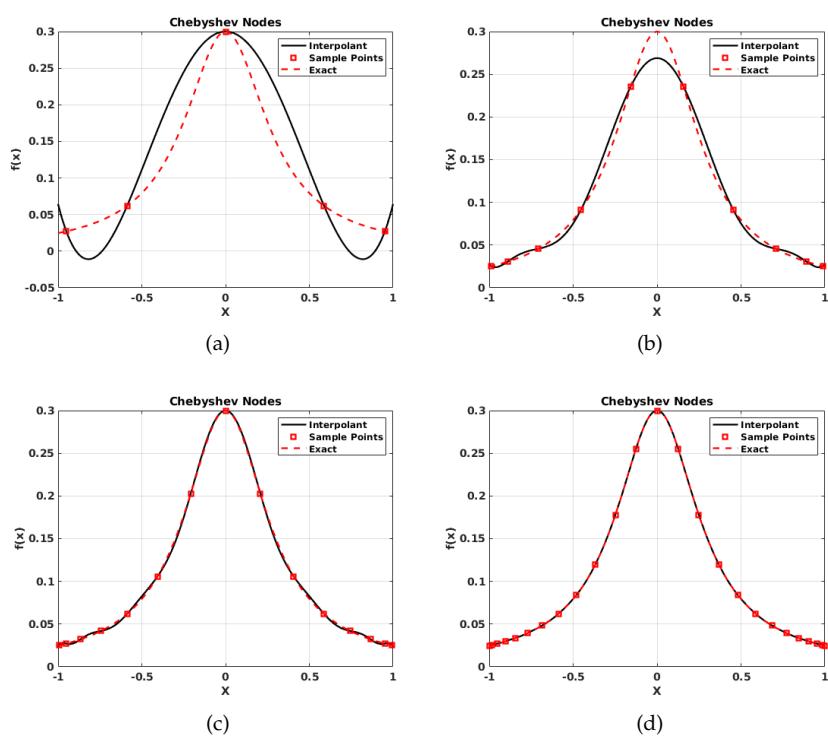


Figure 101: Lagrange interpolation with $n = 5, n = 10, n = 15$, and $n = 20$ points placed at Chebyshev nodes.

show the MATLAB code so you can adapt it to a case of interest to you.

We will start by clearing out the workspace and loading the data we wish to interpolate.

```

clear
clc
close 'all'

%% Load data
Strain = [0 0.4 0.8 1.2 1.6 2.0 2.4 2.8 3.2...
    3.6 4.0 4.4 4.8 5.2 5.6 6.0]';% dimensionless
Stress = [0 3.0 4.5 5.8 5.9 5.8 6.2 7.4 9.6...
    15.6 20.7 26.7 31.1 35.6 39.3 41.5]'; % MPa

```

Once the data is defined, we can create interpolations with both monomials and Lagrange polynomials.

```

%% N-th order Interpolation (monomials)

N = length(Strain);
X = Strain.^0:N;
c = (X'*X)\(X'* Stress);
nthInterp = @(x) (x.^0:N)*c;

%% Lagrange Interpolation
F = genLagrangePolyInterp(Strain ,Stress);

```

Of course, all of the work for the Lagrange polynomial interpolation is packaged into the local function `genLagrangePolyInterp(X,Y)`. The MATLAB code to make that happen is shown next.

```

%% Local function for Lagrange polynomial
function F = genLagrangePolyInterp(X,Y)
% function F = genLagrangePoly(X,Y) generates a Lagrange
% polynomial that may be used to interpolate a function
% inputs
% X = x-values of a function
% Y = f(X) for some function
%
% Outputs
% F - a function handle with the Lagrange interpolant

n = length(X);

F = @(x) 0; % initialize the interpolant

for i = 1:n
    L = @(x) Y(i); % initialize the Lagrange Function
    for j = 1:n
        if j ~= i
            L = @(x) L(x).*((x - X(j))./(X(i) - X(j)));
        end
    end
    F = @(x) F(x)+L(x);
end
end

```

We can plot the resulting interpolants; these are shown in Figure 102 and Figure 103 for monomial and Lagrange interpolation respectively. As can be seen, the Lagrange interpolant is not great. Nonetheless, the use of Lagrange polynomials for interpolation is a valuable tool, particularly for cases where the analyst can select the sample points for the interpolant.

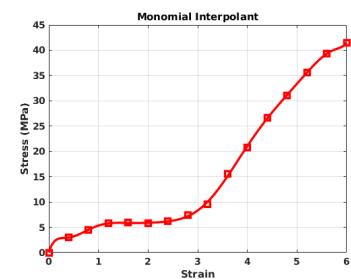


Figure 102: Monomial interpolant for stress-strain data.

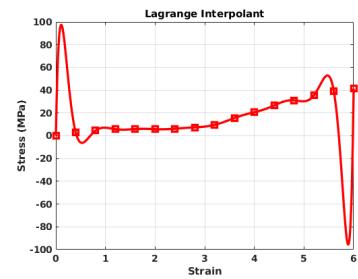


Figure 103: Lagrange interpolation for stress-strain data.

Lecture 16 - Curve Fitting and Interpolation with Built-in MATLAB Tools

Objectives

The objectives of this lecture are to:

- Demonstrate the use of MATLAB built-in tools for polynomial curve fitting.
- Demonstrate the use of MATLAB built-in interpolation functions.

Polynomial Curve Fitting in MATLAB

MATLAB has an easy-to-use tool for polynomial curve fitting: $p = \text{polyfit}(x, y, m)$. In this function, x and y are vectors describing the data to be fit and m is the order of the polynomial fit desired. The function returns a vector p which is a vector of length $(m + 1)$ and stores the coefficients for the polynomial fit:

$$f(x) = p(1)x^m + p(2)x^{m-1} + \cdots + p(m+1)x^0$$

This might seem awkward and/or unexpected, but do not despair, MATLAB provides a convenient function to evaluate this polynomial: $\text{polyval}(p, x)$.

Example: The percent of households that own at least one computer in selected years from 1981 to 2010, according to the U.S. Census Bureau, is listed in Table 15.

Table 15: Percentage of U.S. households that own at least one computer.

Year	1981	1984	1989	1993	1997	2000	2001	2003	2004	2010
% w/computers	0.5	8.2	15	22.9	36.6	51	56.3	61.8	65	76.7

Use MATLAB built-in functions to create a 3rd-order polynomial interpolant and estimate the percent ownership of computers in 2008 and 2013.

```

1 clear
2 clc
3 close 'all'

4 base_year = 1981;
5 year = [1981,1984, 1989, 1993, 1997, ...
6 2000, 2001, 2003, 2004, 2010];
7
8 pct_w_comp = [0.5, 8.2, 15, 22.9, 36.6, 51, ...
9 56.3, 61.8, 65, 76.7];
10
11 year = year-base_year; ❶
12 m = 3;
13 p = polyfit(year,pct_w_comp,m);
14
15 x1 = 2008 - base_year;
16 x2 = 2013 - base_year;
17
18 fprintf('Estimated percent ownership in %d is %g percent.\n',...
19 x1+base_year, polyval(p,x1));
20 fprintf('Estimated percent ownership in %d is %g percent.\n',...
21 x2+base_year, polyval(p,x2));
22

```

The resulting polynomial fit is shown in Figure 104. The estimated percent computer ownership in 2008 is 74.25% and in 2013 is 80.00%.

Interpolation with MATLAB

Although we have thoroughly covered curve fitting and extended that application to interpolation, we really have not yet discussed any methods that can be reliably used for data interpolation. Readers may find this somewhat confusing since they almost certainly have carried out simple linear interpolation at some point in their lives. There are, of course, more sophisticated and reliable algorithms for data interpolation but we will not cover them in detail in this text. Instead, we will rely on built-in MATLAB tools that are readily available. The workhorse function for one-dimensional interpolation is: `interp1(x,y,x_q,method)`. The first two inputs comprise the data to be interpolated. The third input variable, `x_q`, is a vector of *query points* where interpolated data is desired.

MATLAB provides several methods including:

1. '`linear`': This method requires at least 2 points in the data vectors and produces a piece-wise-linear interpolant.
2. '`nearest`': As the name implies this method will interpolate elements of the query vector to the nearest neighbor in the data vector. This results in a piece-wise-constant interpolation.
3. '`next`' or '`previous`': Produces a piece-wise-constant interpolation, as the name implies, to the data point after or before the query point along the x axis.

❶ Translating the variable years in this way provides a better conditioned system of equations. Readers are encouraged to eliminate this translation and observe the result.

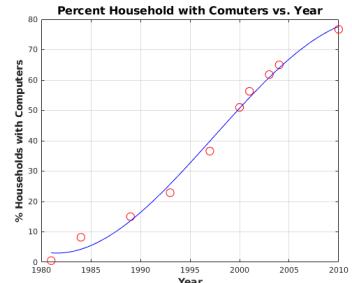


Figure 104: A 3rd-order polynomial fit using `polyval()` and `polyfit ()`.

4. A handful of higher-order and spline interpolants:

- (a) 'cubic'
 - (b) 'pchip' and
 - (c) 'spline'

to list a few.

Readers seeking more details should, of course, consult the MATLAB documentation. An example usage is shown below.

Example: The following data shows the power of a diesel engine at different engine speeds.

Engine RPM	1200	1500	2000	2500	3000	3250	3500	3750	4000	4400
Engine Power (hp)	65	130	185	225	255	266	275	272	260	230

Use MATLAB to create an interpolation of the data. Plot the estimated engine power from 1200 to 4400 RPM. Estimate the engine power at 2300 RPM.

The MATLAB code needed to do this is presented in the listing below.

```

clear
clc
close 'all'

x = [1200, 1500, 2000, 2500, 3000, ...
      3250, 3500, 3750, 4000, 4400];
y = [65, 130, 185, 225, 255, 266, ...
      275, 272, 260, 230];

method = 'pchip';

f_int = @(xi) interp1(x,y,xi,method);

xMin = min(x); xMax = max(x); Nx = 1000;
X_space = linspace(xMin, xMax, Nx);

figure(4)
plot(x,y, 'ro', ...
      X_space, f_int(X_space), '-b', ...
      'linewidth', 3)
title('MATLAB Interpolation', 'fontsize', 16, ...
      'fontweight', 'bold');
xlabel('Engine Speed [RPM]', 'fontsize', 14, 'fontweight', 'bold');
ylabel('Power [hp]', 'fontsize', 14, 'fontweight', 'bold');
grid on;
set(gca, 'fontsize', 12, 'fontweight', 'bold');
axis([xMin xMax 0.5*min(y) 1.25*max(y)]);

x1 = 2300;
fprintf('Estimated hp at %d rpm: %g \n', ...
      x1, f_int(x1));

```

A plot of the interpolation is shown in Figure 105. The estimated engine power at 2300 RPM is 210.43 horsepower using this interpolation method. Readers are encouraged to experiment with alternative interpolation methods to see how each performs.

Interpolation in 2D and 3D

It is often necessary to carry out interpolation on surfaces and in volumes. Perhaps the most important application is in data visualization. A numeric simulation based on the finite difference or finite element method can be used, for example, to find the temperature on a surface or a volume. The temperature, being a function, is usually represented as a vector. In many formulations the temperature vector corresponds to the temperature at discrete points within the surface or volume. What we *want* as engineers is the ability to see a smooth representation of the temperature throughout the domain; perhaps we also carry out some kind of calculation based on the integral or derivatives of that temperature field. The way much of this is done, is via interpolation. We create an interpolant to represent the function throughout the domain and it is the interpolant that is subjected to the differentiation or integration processes. Data *post processing* of this sort is an essential activity that allows us to gain insight from our numerical solutions.

Consider a rectangular domain discretized as part of the finite element method shown in Figure 106. When the analysis is completed, we want to represent the temperature throughout the domain as a smooth function. As is shown in the MATLAB listing below, we can use the built-in function—`scatteredInterpolant()`—to create the smooth interpolated representation shown in Figure 107.

```

1 clear
2 clc
3 close 'all'
4 % load data from FEM analysis
5 load('fem_data.mat');
6 % create an interpolant over the 2D domain
7 TempField = scatteredInterpolant(gcoord(:,1),gcoord(:,2),T);
8 xMin = min(gcoord(:,1)); xMax = max(gcoord(:,1));
9 yMin = min(gcoord(:,2)); yMax = max(gcoord(:,2));
10 % make a smooth plot of temperature over the domain
11 Nx = 200; Ny = 200;
12 X = linspace(xMin,xMax,Nx); Y = linspace(yMin,yMax,Ny);
13 [XX,YY] = meshgrid(X,Y);
14 figure(1)
15 surf(XX,YY,TempField(XX,YY), 'edgecolor', 'none');
16 title('Temperature Field', 'fontSize', 16, ...
17     'fontWeight', 'bold')
18 xlabel('X', 'fontSize', 14, 'fontWeight', 'bold');
19 ylabel('Y', 'FontSize', 14, 'FontWeight', 'bold');
20 zlabel('T ^{\circ}C', 'FontSize', 14, 'FontWeight', 'bold');
```

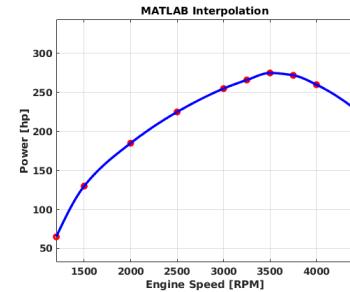


Figure 105: Interpolated curve of engine power versus engine speed.

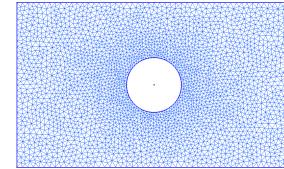


Figure 106: A discretized domain for a finite element method.

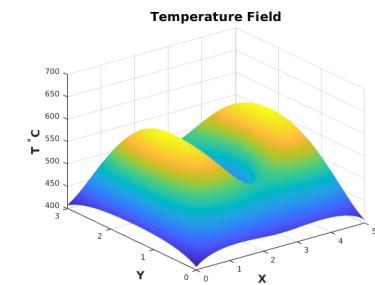


Figure 107: Interpolated temperature data.

Part X

Numeric Differentiation and Integration

Lecture 17 - Numeric Differentiation with Finite Difference Formulas

Objectives

The objectives of this lecture are to:

- Derive basic differentiation formulas using Taylor series expansions.
- Express the differentiation operation in matrix form.

Introduction

Numeric differentiation plays an important role in scientific computing. In the last lecture, we learned to use interpolation to visualize the temperature distribution on a surface as computed in a FEM-based analysis. Suppose we wanted also to calculate the heat flux along a boundary of the domain? As was discussed in the analytic methods portion of this text, the heat flux is related to the derivative of the temperature field:

$$q'' = -k\nabla T$$

where k is the thermal conductivity. In one spatial dimension, this simplifies to: $q'' = -k dT/dx$. In this lecture we will discuss methods for estimating derivatives of a function, $f(x)$, where the function is represented as a vector. This family of methods is referred to as *finite difference* methods.

First Derivative

Consider a function defined on a discrete grid as shown in Figure 108. For simplicity of the following analysis, we will assume that the grid points are all equally spaced: $\Delta x = h$. Suppose we know the value of the function, $f(x_i)$, at all grid points, x_i , and we wish to

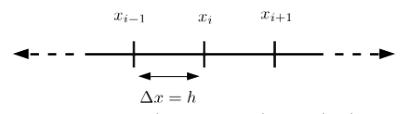


Figure 108: A discrete grid on which a function may be defined.

estimate the first derivative, $f'(x_i)$. From the Taylor series expansion we have:

$$f(x_{i+1}) = f(x_i) + \overbrace{f'(x_i)(x_{i+1} - x_i)}^h + \frac{f''(x_i)}{2!} \overbrace{(x_{i+1} - x_i)^2}^{h^2} + \dots + \frac{f^{(n)}(x_i)}{n!} \overbrace{(x_{i+1} - x_i)^n}^{h^n} + \dots$$

If we solve for $f'(x_i)$, we obtain a two-point *forward difference* formula:

$$\begin{aligned} f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{h} - \text{higher order terms} \\ f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(\xi)}{2!} h \end{aligned}$$

in which it can be shown that the higher order terms are equal to $\frac{f''(\xi)}{2!} h$ with $\xi \in [x_i, x_{i+1}]$. We can more generically characterize that error term using asymptotic notation: $\mathcal{O}(h)$.

We can similarly define a two-point backward difference formula:

$$\begin{aligned} f(x_{i-1}) &= f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!} h^2 - \dots \\ \Rightarrow f'(x_i) &= \frac{f(x_i) - f(x_{i-1})}{h} + \mathcal{O}(h) \end{aligned}$$

These equations are simple to use and effective, however they have a significant drawback in that the error term is proportional to h . For every extra decimal place we need to gain in accuracy, we must reduce h by a factor of 10. This adds up quickly. The good news is that we can do better.

For first derivative formulas, we can try a centered difference scheme:

$$\begin{aligned} f(x_{i+1}) &= f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!} h^2 + \mathcal{O}(h^3) \\ f(x_{i-1}) &= f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!} h^2 - \mathcal{O}(h^3) \\ f'(x_i) &= \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + \underbrace{\frac{\mathcal{O}(h^3)}{2h}}_{\mathcal{O}(h^2)} \end{aligned}$$

Here we subtract the first equation from the second and solve for $f'(x_i)$.

This results in second-order convergence. We can also get second-order convergence if we use more data points in the derivation. We will skip the messy algebraic details but three-point, second-order forward and backward differentiation formulas are given in Equation 169, and Equation 170.

$$f'(x_i) = \frac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2h} \quad (169)$$

$$f'(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 3f(x_i)}{2h} \quad (170)$$

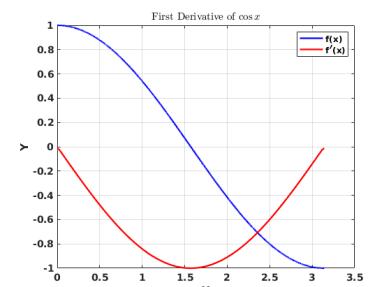
Example: Use the 2- and 3-point finite difference formulas to numerically differentiate $\cos x$.

In this listing, we carry out the differentiation with 2-point difference methods.

```

1 clear
2 clc
3 close 'all'
4
5 f = @(x) cos(x);
6
7 n = 7;
8 Nx = 2^n;
9 xMin = 0; xMax = pi;
10 X = linspace(xMin,xMax,Nx);
11 h = X(2) - X(1);
12
13 % initialize the derivative array
14 df_numeric = nan(1,Nx);
15
16 % use fwd difference at left end
17 df_numeric(1) = (1/h)*(f(X(2))- f(X(1)));
18
19 % use backward difference at right end
20 df_numeric(end) = (1/h)*(f(X(end)) - f(X(end-1)));
21
22 % use centered difference everywhere else
23 i = 2:(Nx-1); ip = i+1; im = i-1; ❶
24 df_numeric(2:(end-1)) = (1/(2*h))*(f(X(ip))-f(X(im)));
25
26 % plot the function and its derivative
27 Xp = linspace(xMin,xMax,10000);
28 figure(1)
29 plot(Xp,f(Xp),'-b',...
30 X,df_numeric,'-r','linewidth',2);
31 grid on;
32 title('First Derivative of $\cos{x}$',...
33 'fontsize',14,...
34 'fontweight','bold','Interpreter','latex');
35 xlabel('X','fontsize',12,'fontweight','bold');
36 ylabel('Y','fontsize',12,'fontweight','bold');
37 legend('f(x)', 'f'(x)');
38 set(gca,'fontsize',12,'fontweight','bold');
```

❶ here we use these vectors to index df_numeric and f in a “vectorized” fashion rather than one equation at a time.



In the next listing a method for carrying out a convergence analysis is shown.

```

39 %% Get Convergence Rate
40 N = 5:15;
41 rel_err = nan(1,length(N));
42 h_err = nan(1,length(N));
43
44 for s = 1:length(N)
45 Nx = 2^N(s);
46 xMin = 0; xMax = pi;
```

Figure 109: A plot of $\cos x$ and its first derivative calculated numerically.

```

X = linspace(xMin,xMax,Nx);
h = X(2) - X(1);
h_err(s) = h;

df_numeric = nan(1,Nx);
% use fwd difference at left end
df_numeric(1) = (1/h)*(f(X(2))-f(X(1)));
% use backward difference at right end
df_numeric(end) = (1/h)*(f(X(end)) - f(X(end-1)));
% use centered difference everywhere else
i = 2:(Nx-1); ip = i+1; im = i-1;
df_numeric(i) = (1/(2*h))*(f(X(ip))-f(X(im)));

% estimate the relative error
x_err = 1:Nx; % include the end points
%x_err = 2:(Nx-1); % exclude the end points

rel_err(s) = norm(df_numeric(x_err) - df(X(x_err)),2) ...
/norm(df(X(x_err)),2);
end

% add gauge lines for convergence
c1 = 0;
h1 = h_err + c1;

c2 = 0;
h2 = h_err.^2 + c2;

figure(2)
loglog(h_err,rel_err,'-b',...
    h_err,h1,'--r',...
    h_err,h2,'--g','linewidth',3);
title('Convergence Behavior','fontsize',14,...
    'fontweight','bold');
xlabel('h','fontsize',12,'fontweight','bold');
ylabel('Relative Error','fontsize',12,'fontweight','bold');
grid on
set(gca,'fontsize',10,'fontweight','bold');
legend('Estimate','h^1 convergence','h^2 convergence',...
    'location','best');

```

We use the 2nd-order accurate centered difference equation through most of the domain but, as Figure 110 shows, the first-order accurate 2-point formulas at the end-points are enough to spoil convergence.

If we use the 2nd-order 3-point forward and backward differentiation formulas, we can get convergence at 2nd-order as shown in Figure 111.

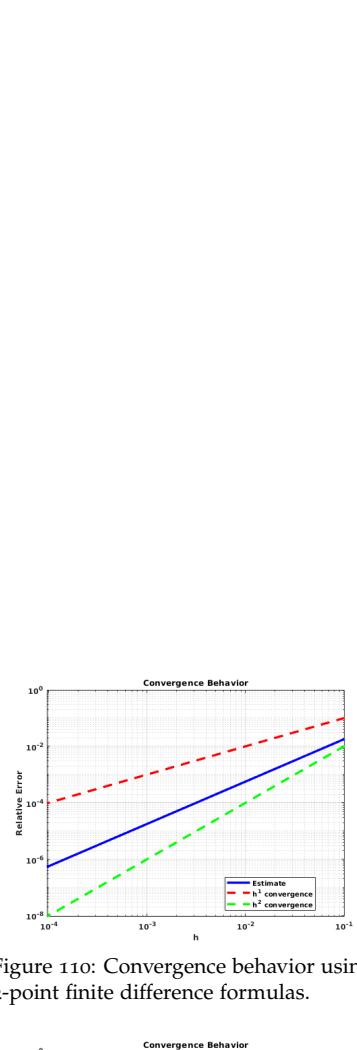


Figure 110: Convergence behavior using 2-point finite difference formulas.

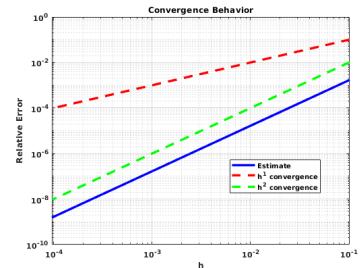


Figure 111: Convergence behavior using 2nd-order finite difference equations throughout the domain.

Second Derivative Formulas

We can use finite difference equations to approximate the second derivative (and higher-order derivatives) as well. As with first-order differentiation, we derive the formula from Taylor-series expansion.

$$\begin{aligned}f(x_{i+1}) &= f(x_i) + f'(x_i)h + \frac{f''(x_i)h^2}{2!} + \frac{f'''(x_i)h^3}{3!} + \dots \\f(x_{i-1}) &= f(x_i) - f'(x_i)h + \frac{f''(x_i)h^2}{2!} - \frac{f'''(x_i)h^3}{3!} + \dots\end{aligned}$$

If we add the two equations above and solve for $f''(x_i)$, we get:

$$f''(x_i) = \frac{2f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2} + \mathcal{O}(h^2) \quad (171)$$

Through a similar process, second-order accurate forward- and backward-differentiation formulas can be derived for use at domain end-points:

$$f'' = \frac{2f(x_i) - 5f(x_{i+1}) + 4f(x_{i+2}) - f(x_{i+3})}{h^2} \quad (172)$$

$$f'' = \frac{-f(x_{i-3}) + 4f(x_{i-2}) - 5f(x_{i-1}) + 2f(x_i)}{h^2} \quad (173)$$

Representation as a Matrix

It is convenient to represent finite difference equations as a matrix. For example, to take the first derivative of a function, $f(x)$, that is represented by a vector f that samples $f(x)$ at uniformly spaced points, the 2nd-order finite difference equations would be:

$$\begin{aligned}\frac{1}{2h} [-3f(x_1) + 4f(x_2) - f(x_3)] &= f'(x_1) \\ \frac{1}{2h} [-f(x_1) + f(x_3)] &= f'(x_2) \\ \frac{1}{2h} [-f(x_2) + f(x_4)] &= f'(x_3) \\ &\vdots &=& \vdots \\ \frac{1}{2h} [-f(x_{n-2}) + f(x_n)] &= f'(x_{n-1}) \\ \frac{1}{2h} [f(x_{n-2}) - 4f(x_{n-1}) + 3f(x_n)] &= f'(x_n)\end{aligned}$$

where h is the distance between the points, $h = x_i - x_{i-1}$. These equations can be represented in matrix-vector notation as:

$$\frac{1}{2h} \begin{bmatrix} -3 & 4 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & \cdots & \cdots & 0 & -1 & 0 & 1 \\ 0 & 0 & \cdots & \cdots & 0 & -1 & -4 & 3 \end{bmatrix} \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ \vdots \\ f(x_{n-2}) \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix} = \begin{bmatrix} f'(x_1) \\ f'(x_2) \\ f'(x_3) \\ f'(x_4) \\ \vdots \\ f'(x_{n-2}) \\ f'(x_{n-1}) \\ f'(x_n) \end{bmatrix}$$

Or, more concisely:

$$Af = f'$$

where A is a sparse $n \times n$ matrix of the coefficients. The matrix A should be thought of as a discrete representation of the differentiation operator. Whereas in calculus class one might write:

$$\frac{d}{dx}f(x) = f'(x)$$

The discrete version of this is given in the equation above. The functions have been replaced by vectors and the differential operator has been replaced by a matrix which their respective discrete equivalents. Of course, the matrix/vector formulation is an approximation, but the approximation improves as h is reduced.

Example: Consider the boundary value problem below:

$$\begin{aligned} \frac{d^2f}{dx^2} + \lambda^2 f &= 0, \quad \lambda > 0, \quad 0 < x < \pi \\ f(0) &= 0, \quad f(\pi) = 0 \end{aligned}$$

The general solution to the differential equation is:

$$f(x) = c_1 \cos(\lambda x) + c_2 \sin(\lambda x)$$

The boundary condition at $x = 0$ gives us:

$$f(0) = c_1 \cos(0) + c_2 \sin(0) = 0 \Rightarrow c_1 = 0$$

For the boundary condition at $x = \pi$ we have:

$$f(\pi) = c_2 \sin(\lambda\pi) = 0$$

As usual, we are looking for non-trivial solutions, so rather than set $c_2 = 0$, we look for values of λ such that $\sin(\lambda\pi) = 0$. This will be true when $\lambda\pi = n\pi$ where n is a positive integer. Thus, $\lambda = n$, so the eigenvalues are $\lambda^2 = n^2$ for $n = 1, 2, 3, \dots$, and the eigenfunctions are:

$$f_n(x) = \sin nx \quad (174)$$

We can solve the same eigenvalue problem using differentiation matrices. Applying the finite difference equations given in Equations 171, 172 and 173, we can assemble a matrix representation of the differential operator:

$$\begin{aligned} \frac{d^2 f}{dx^2} &= -\lambda^2 f \\ Af &= -\lambda^2 f \end{aligned}$$

This is accomplished in the code block below:

```
% discretize the spatial domain
N = 9;
Nx = 2^N;
xMin = 0; xMax = pi;
X = linspace(xMin,xMax,Nx);
h = X(2) - X(1);

% Create a matrix for 2nd derivative operator
A = zeros(Nx,Nx);

% set coefficients for first equation
A(1,1) = 2/h^2; A(1,2) = -5/h^2;
A(1,3) = 4/h^2; A(1,4) = -1/h^2;

% set coefficients for all interior equations
for m=2:(Nx-1)
    A(m,m) = -2/h^2; A(m,m-1) = 1/h^2;
    A(m,m+1) = 1/h^2;
end

% set coefficients for last equation
A(Nx,Nx) = 2/h^2; A(Nx,Nx-1) = -5/h^2;
A(Nx,Nx-2) = 4/h^2; A(Nx,Nx-3) = -1/h^2;
```

Note: We exclude the case $n = 0$ since we stipulated that $\lambda > 0$.

Note: As can be seen, it is really more correct to say that the eigenvalues are $-\lambda^2 = n^2$ or $-1, -4, -9, \dots$

We need to apply the boundary conditions to the matrix A . One way of accomplishing this is illustrated in the next code block.

```
% Set boundary conditions
A(1,:) = 0; A(:,1) = 0; A(1,1) = 1; %❷
A(end,:) = 0; A(:,end) = 0; A(end,end) = 1;
```

We will use the MATLAB built-in function `eigs` to find the eigenvalues and eigenvectors of A .

❷ Here we set the first and last diagonal of A to 1; the first and last row and column, other than the diagonal, is set to zero. Effectively this eliminates the first and last equation of A from the matrix. This method of applying the Dirichlet boundary condition has the feature that the symmetry of A is preserved.

```

% Get the four smallest eigenvalues and eigenvectors of A
[V,D] = eigs(A(2:(end-1),2:(end-1)),4,'smallestabs');

% reduce the X domain to exclude the boundaries
X_red = X(2:(end-1));
figure(5)

subplot(4,1,1)
plot(X_red,(V(:,1))/norm(V(:,1),2),'.b')
title('First Four Eigenvectors of A',...
'FontSize',14,'FontWeight','bold');
subplot(4,1,2)
plot(X_red,(V(:,2))/norm(V(:,2),2),'.b')

subplot(4,1,3)
plot(X_red,(V(:,3))/norm(V(:,3),2),'.b')

subplot(4,1,4)
plot(X_red,(V(:,4))/norm(V(:,4),2),'.b')
xlabel('X','FontSize',12,'FontWeight','bold');

% Display the eigenvalues
fprintf('D = \n');
disp(D);

```

The first four eigenvectors of A are plotted in Figure 112. While not a rigorous proof, you can check and see that the eigenvectors are orthogonal¹ which is analogous to the orthogonality of the eigenfunctions given in Equation 174. The eigenvalues are shown on the main diagonal of D in Figure 113.

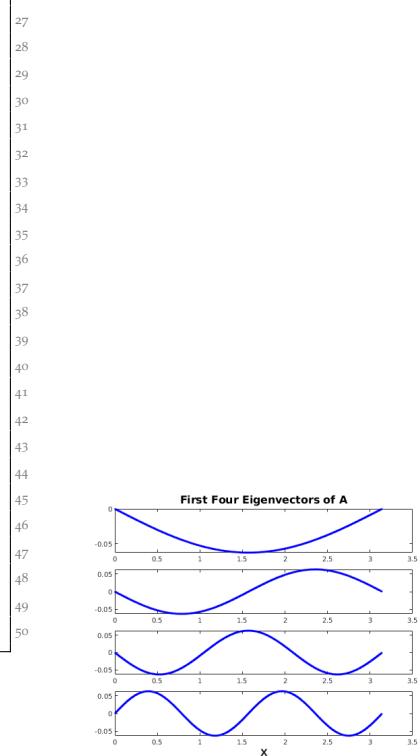


Figure 112: A plot of the first four eigenvectors of A .

$$D = \begin{matrix} -1.0000 & 0 & 0 & 0 \\ 0 & -3.9999 & 0 & 0 \\ 0 & 0 & -8.9997 & 0 \\ 0 & 0 & 0 & -15.9992 \end{matrix}$$

Figure 113: The matrix D with the first 4 eigenvalues of A .

¹ It is worth admitting here that the eigenvectors of *all* symmetric matrices are orthogonal. Since A is symmetric, its eigenvectors are orthogonal. Still there is a deeper connection – the differential operator that A represents is, in some way, *symmetric* and that is reflected in the symmetry of A .

Lecture 18 - Numeric Differentiation with Lagrange Polynomials

Objectives

The objectives of this lecture are to:

- Review interpolation with Lagrange Polynomials
- Describe and demonstrate numeric differentiation with Lagrange Polynomials
- Illustrate use of non-uniform sample points

Interpolation with Lagrange Polynomials

As readers may recall from Lecture 15, we can approximate a function with Lagrange Polynomials using Equation 168 which is copied here for convenience:

$$f_{\text{interp}}(x) = \sum_{i=1}^n y_i L_i(x) = \sum_{i=1}^n y_i \underbrace{\prod_{\substack{j=1 \\ j \neq i}}^n \frac{(x - x_j)}{(x_i - x_j)}}_{\text{Lagrange function}}$$

where x_i are the points where the function is sampled and $y_i = f(x_i)$. This is an interpolation which implies, at a minimum, that the interpolant $f_{\text{interp}}(x)$ is equal to $f(x)$ at the interpolating points. What happens *between the points*, however, is another matter entirely.

Differentiation of Lagrange Interpolant

The strategy we will explore in this lecture is straight-forward. In order to estimate the derivative of a function, we will first approximate the function with a polynomial (Lagrange) interpolant; then we will take the derivative of the interpolant.

To illustrate this method, we will start with a relatively simple 3-point Lagrange interpolation:

$$f_{\text{interp}}(x) = \frac{(x - x_2)(x - x_3)y_1}{(x_1 - x_2)(x_1 - x_3)} + \frac{(x - x_1)(x - x_3)y_2}{(x_2 - x_1)(x_2 - x_3)} + \frac{(x - x_1)(x - x_2)y_3}{(x_3 - x_1)(x_3 - x_2)}$$

The derivative of this function is not too difficult, but it is somewhat messy:

$$\begin{aligned} \frac{df_{\text{interp}}}{dx} &= y_1 \left[\frac{1}{x_1 - x_2} \left(\frac{x - x_3}{x_1 - x_3} \right) + \frac{1}{x_1 - x_3} \left(\frac{x - x_2}{x_1 - x_2} \right) \right] + \\ &\quad y_2 \left[\frac{1}{x_2 - x_1} \left(\frac{x - x_3}{x_2 - x_3} \right) + \frac{1}{x_2 - x_3} \left(\frac{x - x_1}{x_2 - x_1} \right) \right] + y_3 \left[\frac{1}{x_3 - x_1} \left(\frac{x - x_2}{x_3 - x_2} \right) + \frac{1}{x_3 - x_2} \left(\frac{x - x_1}{x_3 - x_1} \right) \right] \end{aligned}$$

We generalize this and encode in the notation used to describe Lagrange polynomials:

$$\begin{aligned} \frac{df_{\text{interp}}}{dx} &= \sum_{i=1}^n y_i \frac{d}{dx} L_i(x) = \sum_{i=1}^n y_i \frac{d}{dx} \prod_{\substack{j=1 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j} \quad (175) \\ &= \sum_{i=1}^n y_i \left[\sum_{\substack{k=1 \\ k \neq i}}^n \frac{1}{(x_i - x_k)} \prod_{\substack{j=1 \\ j \neq i \\ j \neq k}}^n \frac{(x - x_j)}{(x_i - x_j)} \right] \end{aligned}$$

This formulation is admittedly messy, but with some patient MATLAB coding, it can be implemented without undue difficulty. A sample implementation is provided in the listing below.

```

function dF = genLagrangeInterpDeriv(X,Y)
% function dF = genLagrangeInterpDeriv(X,Y) generates the
% derivative of a function using Lagrange Polynomial
% interpolation.
% Inputs
% X = x-values of the function
% Y = f(X) for some function
%
% Outputs
% dF - a function handle with the derivative of
% the Lagrange interpolant

n = length(X);
dF = @(x) o; ❶

for i = 1:n
    dLi = @(x) o;
    for k = 1:n
        if k ~= i
            dLp = @(x) 1; ❷
            for j = 1:n
                if ((j ~= i) && (j ~= k))
                    dLp = @(x) dLp(x).* ...
                        (x - X(j))./(X(i) - X(j));
                end
            end
        end
    end
end

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

❶ We initialize the derivative with Lagrange Interpolant with o - the additive identity. As we add terms in line 32 of the listing, we do not need to make a special case for i=1.

❷ Similarly here, we initialize the derivative of the Lagrange function with dLp = @(x) 1, which is the multiplicative identity, so a special case is not needed on line 23 and 24 where we include the next term of the product.

```

    end
end
dLi = @(x) dLi(x) + ...
(1./((X(i) - X(k))).*dLp(x);
end

end
dF = @(x) dF(x) + dLi(x)*Y(i);
end
end

```

As can be seen just from the MATLAB implementation, this method is fundamentally different from finite difference formulas. Using finite difference formulas, each equation only includes a few of the sample points while the Lagrange interpolant is a continuous function that includes *all* sample points. Recalling the discussion in Lecture 15 on Lagrange interpolation, you might expect that the quality of the numeric differentiation will depend not only on the number of sample points—where, in general, the quality of interpolation increases with more sample points—but also the *distribution* of the sample points throughout the domain.

Example: Use a Lagrange polynomials to numerically find the derivative of:

$$f(x) = \frac{1}{1 + 25x^2} \quad (176)$$

Analytically, the derivative is:

$$\frac{df}{dx} = \frac{-50x}{(1 + 25x^2)^2}$$

Using $n = 11$ uniformly spaced sample points, the function and its numerical derivative is shown in Figure 114. The numerical derivative is of very poor quality. For uniformly spaced points, however, the quality does not improve if n is increased. The result for $n = 21$ is shown in Figure 115. If anything, the numeric derivative is worse.

In contrast, if we use non-uniformly spaced Chebychev points, as we did in Lecture 15, the quality of the numeric derivative is markedly better as we see in Figure 116. Furthermore, if we increase the number of interpolation points, the quality of the numeric derivative improves still further as is shown in Figure 117.

When we study Finite Element Methods in future lectures, we will find numeric differentiation with Lagrange Polynomials using non-uniform sample points to be a very important and powerful tool.

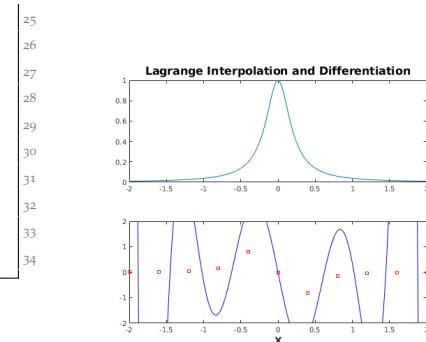


Figure 114: Numeric differentiation with $n = 11$ uniformly spaced points.

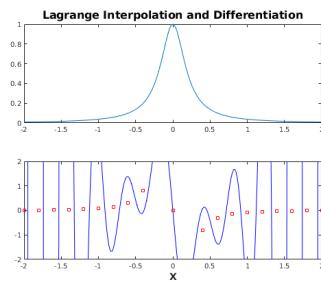


Figure 115: Numeric differentiation with $n = 21$ uniformly spaced points.

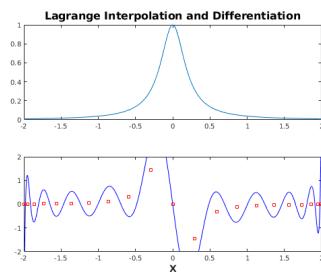


Figure 116: Numeric differentiation with $n = 21$ non-uniformly spaced points.

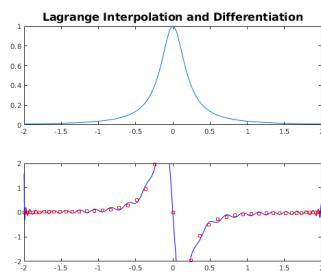


Figure 117: Numeric differentiation with $n = 51$ non-uniformly spaced points.

Assignment #6

1. Measurements of thermal conductivity, k (W/m-K), of silicon at various temperatures are given in the table below.

T[K]	50	100	150	200	400	600	800	1000
k [W/m-K]	28	9.1	4.0	2.7	1.1	0.6	0.4	0.3

The data is to be fitted with a function of the form $k = f(T)$. Determine which of the nonlinear equations presented in Lecture 14 can best fit the data and determine its coefficients. Make a plot that shows the data points (asterisk marker) and the equation (solid line).

2. The following data are given:

x	1	2.2	3.4	4.8	6	7
y	2	2.8	3	3.2	4	5

Write the polynomial in Lagrange form that passes through the points.

3. Values of enthalpy per unit mass, h , of an equilibrium Argon plasma (Ar , Ar^+ , Ar^{++} , Ar^{+++} ions and electrons) versus temperature are given in the table below.

$T \times 10^3$ [K]	5	7.5	10	12.5	15	17.5	20	22.5	25	27.5	30
h [MJ/kg]	3.3	7.5	41.8	51.8	61	101.1	132.9	145.5	171.4	225.8	260.9

Write a script that uses interpolation to calculate h at temperatures ranging from 5000 K to 30000 K in increments of 500 K. The program should generate a plot that shows the interpolated points and the data points from the table (use an asterisk marker).

- (a) For interpolation use Lagrange polynomials as demonstrated in Lecture 15.
- (b) For interpolation use MATLAB's built-in tool `interp1` with method='spline'.

4. Given the following data:

x	1.1	1.2	1.3	1.4	1.5
$f(x)$	0.6133	0.7822	0.9716	1.1814	1.4117

Find the first derivative, $f'(x)$, at the point $x = 1.3$.

- (a) Use the three-point forward-difference formula.
- (b) Use the three-point backward-difference formula.
- (c) Use the two-point centered difference formula.

5. Given the following data:

x	0.6	0.7	0.8	0.9	1.0
$f(x)$	5.2296	3.6155	2.7531	2.2717	2

Find the second derivative, $f''(x)$, at the point $x = 0.8$.

- (a) Use the three-point forward-difference formula.
- (b) Use the three-point backward-difference formula.
- (c) Use the two-point centered difference formula.

Lecture 19 - Numeric Integration with Newton-Cotes Formulas

Objectives

The objectives of this lecture are to:

- Describe and illustrate the Midpoint and Trapezoidal “rules” for numeric integration
- Demonstrate Simpson’s Rule
- Introduce and demonstrate *numerical quadrature* for deriving integration formulas

Newton-Cotes Integration Formulas

Suppose you have a function, $f(x)$, that you want to integrate but, for some reason, you do not know how. In addition to the more-or-less broad category of functions that you do not know how to integrate analytically, we can add the, perhaps, equally broad class of functions that simply cannot be integrate analytically at all by anyone. One strategy for dealing with this situation is to:

1. Replace $f(x)$ with some other function $g(x)$ that you *can* integrate.
2. If $g(x)$ is in some sense “close” to $f(x)$, you might hope that the integral of $g(x)$ would serve as a decent approximation to the integral of $f(x)$.

This is the strategy followed for the Newton-Cotes integration formulas. Each of the formulas in this class is distinguished from the others by the function $g(x)$ that the method used to approximate $f(x)$. What is common among these formulas is the manner in which $g(x)$ is forced to be “close” to $f(x)$ and it is this: for each of the Newton-Cotes integration formulas, the simple and easily integrated function $g(x)$ is forced to be exactly equal to $f(x)$ at some finite number of discrete points.

Midpoint Rule

Figure 118 gives a simple illustration of the midpoint rule. For the midpoint rule, the function $f(x)$ is approximated as a piece-wise constant. Each piece of the piece-wise constant function is set to be equal to $f(x)$ at the midpoint of each sub-interval.¹ It is not necessary that the sub-intervals be the same width but, in order to ease the implementation and to be consistent with common theoretical analysis, we will assume the sub-intervals are of equal length, which we will denote: $(x_i - x_{i-1}) = h$.

The integration formula is:

$$\int_a^b f(x) dx \approx \int_a^b g(x) dx = \int_{x_0}^{x_1} f\left(\frac{x_1 + x_0}{2}\right) dx + \cdots + \int_{x_{n-1}}^{x_n} f\left(\frac{x_n + x_{n-1}}{2}\right) dx \\ = h \sum_{i=1}^n f(m_i), \quad m_i = \frac{x_i + x_{i-1}}{2}$$

A simple implementation of this formula into a MATLAB function is shown in the listing below.

```

function y = midpoint(f,xMin,xMax,N)
% function y = midpoint(f,a,b,N)
% inputs:
% f -- function_handle. Handle to the function to be integrated
% xMin -- scalar. Lower bound of integration
% xMax -- scalar. Upper bound of integration
% N -- scalar. Number of subdivisions
% output:
% y -- scalar. Approximate of the integral of f(x) from a to b.
xS = linspace(xMin,xMax,N+1);
xMid = (1/2)*(xS(1:(end-1))+xS(2:end));
h = xS(2)-xS(1);
y = h*sum(f(xMid));
end

```

As is shown in Figure 119, as the number of sub-intervals is increased, and thus h decreases, the integration error is reduced. For this method, the global error is $\mathcal{O}(h^2)$; meaning that the error can be bounded above by $C_0 h^2$ where C_0 is a constant. With this notation, by indicating the midpoint's global error is $\mathcal{O}(h^2)$ we say that the midpoint method is *second order* convergent.

Trapezoidal Rule

A schematic of the trapezoidal rule is shown in Figure 120. The domain $[a, b]$ is broken into n sub-intervals and the function $f(x)$ is approximated by a linear function within each sub-interval; the linear function $g(x)$ matches $f(x)$ at the end-points. As promised, integrating each of the simple functions within the domain is easy; each

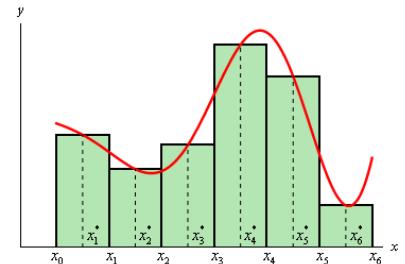


Figure 118: Schematic of midpoint rule.

¹ These methods are only applicable for definite integrals over $x \in (a, b)$. Also, unless otherwise noted, you should assume a and b are finite.

Note: This implementation doesn't allow the user to break the domain into unequal sub-intervals. Think about how you could change the function—not just the body, but the inputs also—to allow non-uniform sub-interval sizes. What benefits might such a feature bring?

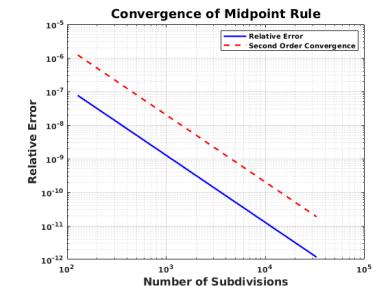


Figure 119: Convergence of the midpoint rule.

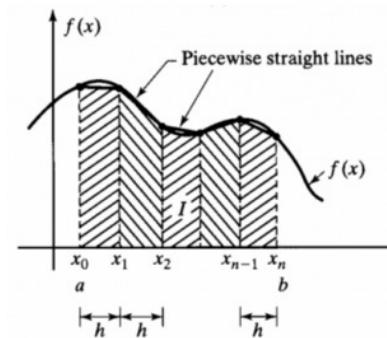


Figure 120: Schematic of the trapezoidal rule with equal sub-interval sizes.

sub-interval is a trapezoid whose area can be calculated with a simple formula:

$$\int_a^b f(x) dx \approx \int_a^b g(x) dx = \int_{x_0}^{x_1} \frac{f(x_0) + f(x_1)}{2} dx + \dots + \int_{x_{n-1}}^{x_n} \frac{f(x_{n-1}) + f(x_n)}{2} dx$$

As with the midpoint rule we will use equally-spaced sub-intervals: $x_i - x_{i-1} = h$, $i = 1, 2, \dots, n$ which results in the final form given in Equation 177.

$$\int_a^b f(x) dx \approx h \left[\frac{f(x_0 = a)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(x_n = b)}{2} \right] \quad (177)$$

Using this equation, we find that we can obtain second order convergence as shown in Figure 121.

Simpson's Rule

In much the same way as with the midpoint and trapezoidal rules, Simpson's rule replaces $f(x)$ with a function that can be easily integrated; in this case a quadratic function. The method is schematically shown in Figure 122. We partition the domain into an even number of sub-intervals (odd number of discrete points) each of equal length. The function $g(x)$ is defined as:

$$g(x) = c_0 + c_1 x + c_2 x^2$$

We make $g(x)$ "close" to $f(x)$ by requiring $g(x_i) = f(x_i)$ at each partitioning point, x_i . With some tedious algebra and calculus it can be shown that a unique function $g(x)$ meeting these conditions can be found and that its integral over a subdomain is:

$$\int_{x_{i-1}}^{x_{i+1}} f(x) dx \approx \int_{x_{i-1}}^{x_{i+1}} g(x) dx = \frac{h}{3} [f(x_{i-1}) + 4f(x_i) + f(x_{i+1})]$$

where $h = x_i - x_{i-1}$. If we repeat this formula for all of the sub-intervals in $[a, b]$ we have the composite Simpson's rule formula as shown in Equation 178.

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(a) + 4 \sum_{i=2,4,\dots}^{n-1} f(x_i) + 2 \sum_{j=3,5,\dots}^n f(x_j) + f(b) \right] \quad (178)$$

An alternative derivation for Simpson's rule can be constructed by combining the midpoint and trapezoidal rules. Careful inspection of Figure 119 and Figure 121 reveals that the midpoint rule is slightly more accurate than the trapezoidal rule. In fact, if you use both methods to integrate any quadratic polynomial using a single interval you will find that the error in the midpoint rule is *exactly half* and *opposite*

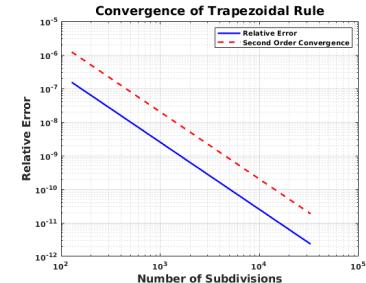


Figure 121: Convergence behavior of the trapezoidal rule.

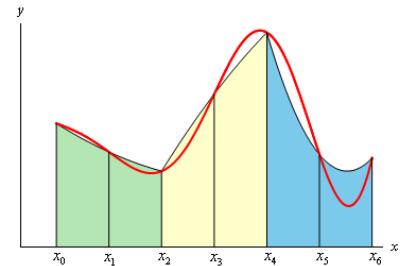


Figure 122: Schematic of Simpson's rule.

in sign of the error when using the trapezoidal rule.² This suggests that one might get better accuracy if one combined the two methods: multiplying the midpoint rule result by $2/3$ and the trapezoidal rule result by $1/3$ and adding the results together, the error for any quadratic should go to zero! If you follow this prescription:

² Try it!!

$$\begin{aligned}\int_a^b f(x) dx &\approx \frac{2}{3} \text{midpoint rule} + \frac{1}{3} \text{trapezoidal rule} \\ &= \frac{2}{3}(2h) \left[f\left(\frac{a+b}{2}\right) \right] + \frac{1}{3}(2h) \left[\frac{f(a) + f(b)}{2} \right] \\ &= \frac{h}{3} [f(a) + 4f(a+b/2) + f(b)]\end{aligned}$$

The last line of which is just Simpson's rule on a single interval.³

The convergence behavior of Simpson's rule is shown in Figure 123. One thing that is not at all obvious from the previous discussion is: *Why is Simpson's rule 4th-order convergent?* In particular, it is easy to verify that any *cubic* polynomial will be integrated *exactly* using Simpson's rule. A logical way to see why this happens is to reformulate Simpson's rule as a *numerical quadrature* rule. Understanding this way of thinking will be useful for understanding the Gauss quadrature methods described in the next lecture.

³ The factor $2h$ and h appear because the original single interval for the midpoint and trapezoidal rules was formally subdivided to conform with the notation used in Simpson's rule.

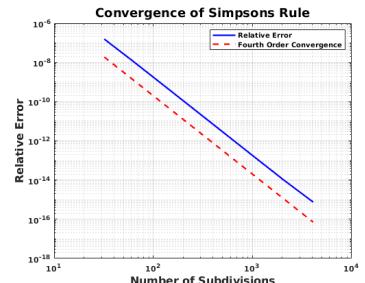


Figure 123: Convergence behavior of Simpson's rule.

Numerical Quadrature

In using numerical quadrature, we choose *sample points* and *weights* such that functions of increasingly higher order are integrated exactly. We will show that Simpson's rule is equivalent to a numerical quadrature formula that allows us to integrate third-order polynomials exactly.⁴

$$\begin{aligned}\int_0^1 f(x) dx &\approx \int_a^b g(x) dx = \sum_{n=1}^3 w_i g(x_i) \\ g(x) &= c_1 + c_1 x + c_2 x^2 + c_3 x^3\end{aligned}$$

⁴ Without loss of generality, we will assume that the interval of integration is $[0, 1]$. We can get back to the notation of Simpson's rule by multiplying the result by $2h$ since Simpson's rule must have, at a minimum, 2 sub-intervals which we will assume to be of equal length h .

Where w_i are the weights, and x_i are the sample points. Our quadrature rule must satisfy:

$$\sum_{i=1}^3 w_i g(x_i) = \overbrace{c_0 \int_0^1 1 dx + c_1 \int_0^1 x dx + c_2 \int_0^1 x^2 dx + c_3 \int_0^1 x^3 dx}^{\int_0^1 g(x) dx}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ x_1^2 & x_2^2 & x_3^2 \\ x_1^3 & x_2^3 & x_3^3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} \int_0^1 1 dx \\ \int_0^1 x dx \\ \int_0^1 x^2 dx \\ \int_0^1 x^3 dx \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \end{bmatrix}$$

Which results in the following nonlinear system of equations:

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ x_1^2 & x_2^2 & x_3^2 \\ x_1^3 & x_2^3 & x_3^3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \end{bmatrix} \quad (179)$$

In principle, the sample points and the weights are all unknown. Let us simplify the problem and move a step closer to Simpson's rule by stipulating, at least, that $x_1 = 0$ and $x_3 = 1$. This then leaves the middle sample point, x_2 , and all of the weights unknown.

We can analyze this using the MATLAB built-in function fsolve.

```

clear
clc
close all

% naive initial guesses
% xW(1) = x_2
% xW(2:4) = w_1, w_2, w_3
xW = [0.25 .25 0.25 0.25];

w = fsolve(quadRule,xW);

fprintf('The middle sample point is: x = %g.\n',w(1));
fprintf('The weights are: \n');
format long
disp(w(2:end))
format short

```

```

%% Local function for
function w = quadRule(xW)

A = [1 1 1;
      0 xW(1) 1;
      0 xW(1)^2 1;
      0 xW(1)^3 1];

w = A*[xW(2);xW(3);xW(4)] - [1; 1/2; 1/3; 1/4];
end

```

```

21
22
23
24
25
26
27
28
29
30 The middle sample point is: x = 0.5.
31 The weights are:
      0.66666659238851   0.166666670382028

```

Output from this script is shown in Figure 124 indicating that, in order to exactly integrate a cubic polynomial, the sample point x_2 should be at the midpoint of the domain, and that the weights should be: $[1/6, 2/3, 1/6]$. When we shift the interval to $[a, b]$ which is of length $2h$, we recover Simpson's rule. This explains the observed convergence behavior.

Other quadrature formulas can be obtained using the exact same procedure. For example, a 3-point quadrature formula that will exactly integrate 5th-order polynomials and (presumably) exhibit 6th-order convergence can be found by solving the non-linear equations:

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ x_1^2 & x_2^2 & x_3^2 \\ x_1^3 & x_2^3 & x_3^3 \\ x_1^4 & x_2^4 & x_3^4 \\ x_1^5 & x_2^5 & x_3^5 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \\ 1/6 \end{bmatrix} \quad (180)$$

where all of the sample points and weights are unknown. This system of equations can be solved to obtain the sample points and weights that are shown in Table 16. Convergence behavior is shown in Figure 125.

For three sample points and three weights, exact integration of 5th order polynomials and the associated 6th order convergence is the best you can do. It turns out that the sample points and weights derived for this integrating rule are equivalent to those that will be derived in the next lecture, which is on Gauss quadrature, though the method used in the formulation is different.

Figure 124: MATLAB output to find middle sample point and all weights.

i	x_i	w_i
1	0.1127016653792583	0.2777777777777786
2	0.5	0.44444444444444493
3	1 - x_1	w_1

Table 16: Sample points and weights for a 3-point, 6th order convergent quadrature formula.

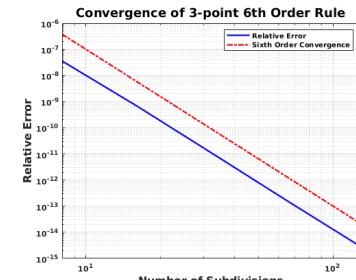


Figure 125: Convergence behavior of a 3-point, 6th order quadrature formula.

Lecture 20 - Gauss Quadrature

Objectives

The objectives of this lecture are to:

- Define Gauss-Legendre quadrature and illustrate its convergence properties.
- Provide a derivation to explain how and why Gauss quadrature works.

Gauss-Legendre Quadrature

Gauss-Legendre quadrature, from here-on-out referred to simply as Gauss quadrature, is similar to Newton-Cotes integration formulas in its overall algorithmic strategy:

- Approximate $f(x)$ by some other function, $g(x)$, that is in some way “close” to $f(x)$; and
- Integrate that function exactly.

The main differences are:

1. Gauss quadrature schemes use a different measure to determine if $g(x)$ is “close” to $f(x)$. In Newton-Cotes formulas, $g(x)$ was forced to be equal to $f(x)$ at pre-selected locations. Gauss quadrature measures closeness in an integral sense as shown in Equation 181.

$$\int_a^b [f(x) - g(x)] g(x) dx = 0 \quad (181)$$

If $g(x)$ is close to $f(x)$, then the integral in Equation 181, which we can think of as an inner product between $g(x)$ and the “error” $f(x) - g(x)$, will be small.

2. Gauss quadrature does not require $g(x)$ to match $f(x)$ at any pre-determined number of locations in the interval, but it does call for sampling $f(x)$ at a number of discrete points throughout the interval much like trapezoidal and Simpson’s rule does. The location of these sample points (often called *integration points* or *Gauss points*) is a key factor to the accuracy that Gauss quadrature can achieve.

In this lecture we will discuss two versions of the Gauss quadrature algorithm. First, we will describe the algorithm in a recipe-like way without any discussion regarding the theory or why it all works. After that is out of the way, a full derivation will be presented so that readers will know exactly how and why the method works.

The Short Version

In Gauss quadrature, the definite integral of a function $f(x)$ over an interval $[-1, 1]$ is approximated by Equation 182.

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^N w_i f(x_i) \quad (182)$$

Since we generally do not want to integrate functions only from $x \in [-1, 1]$, we will linearly map the desired domain, $[a, b]$, onto the reference domain, $t \in [-1, 1]$, as follows:

$$x(t) = \frac{1}{2}[(b-a)t + a + b]$$

$$dx = \frac{1}{2}(b-a) dt$$

Substituting this into the equation above gives us:

$$\int_a^b f(x) dx = \int_{-1}^1 f(x(t)) \frac{b-1}{2} dt \approx \sum_{i=1}^n w_i f(x(t_i)) \frac{b-a}{2} \quad (183)$$

Values for Gauss points and weights are commonly tabulated as shown in Figure 126. Engineers wanting to use Gauss quadrature incorporate those values into their code in some appropriate way and carry out the numeric integration.

The convergence of Gauss quadrature depends on the number of Gauss points used: order $2n - 1$ for n points. If you want faster convergence, simply add more points.¹ Figure 127 shows the convergence behavior of Gauss quadrature for calculating the integral:

$$\int_{-3}^3 e^{-x^2} dx$$

This convergence result was obtained using a single interval for the integral. It is, of course, possible to break the domain into multiple sub-domains and carry out the integral using Gauss quadrature for each sub-domain and then sum the result. In that case, convergence depends on the number of Gauss points used on each sub-domain. The error will be $\mathcal{O}(h^{2n-1})$ where h is the size of each sub-domain and n is the number of Gauss points used on each sub-domain.

Notice that at $x(t = -1) = a$, and $x(t = 1) = b$.

n (Number of points)	Coefficients C_i (weights)	Gauss points x_i
2	$C_1 = 1$ $C_2 = 1$	$x_1 = -0.57735027$ $x_2 = 0.57735027$
3	$C_1 = 0.5555556$ $C_2 = 0.8888889$ $C_3 = 0.5555556$	$x_1 = -0.77459667$ $x_2 = 0$ $x_3 = 0.77459667$
4	$C_1 = 0.3478548$ $C_2 = 0.6521452$ $C_3 = 0.6521452$ $C_4 = 0.3478548$	$x_1 = -0.86113631$ $x_2 = -0.33998104$ $x_3 = 0.33998104$ $x_4 = 0.86113631$
5	$C_1 = 0.2369269$ $C_2 = 0.4786287$ $C_3 = 0.5688889$ $C_4 = 0.4786287$ $C_5 = 0.2369269$	$x_1 = -0.90617985$ $x_2 = -0.53846931$ $x_3 = 0$ $x_4 = 0.53846931$ $x_5 = 0.90617985$
6	$C_1 = 0.1713245$ $C_2 = 0.3607616$ $C_3 = 0.4679139$ $C_4 = 0.4679139$ $C_5 = 0.3607616$ $C_6 = 0.1713245$	$x_1 = -0.93246951$ $x_2 = -0.66120938$ $x_3 = -0.23861919$ $x_4 = 0.23861919$ $x_5 = 0.66120938$ $x_6 = 0.93246951$

Figure 126: Published Gauss points and weights up to $n = 6$.

¹ This is referred to as *exponential convergence*.

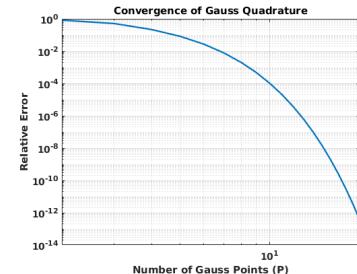


Figure 127: Convergence behavior of Gauss quadrature.

The Long Version

For Gauss quadrature, we approximate the integral of $f(x)$ as follows:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n w_i g(x_i)$$

where, as before, w_i are the weights and x_i are the sample points. We are free to choose these weights and sample points in any way we wish, giving us a total of $2n$ degrees of freedom.

With these $2n$ degrees of freedom, we might hope to *exactly* integrate any polynomial up to degree $2n - 1$ much like in the trapezoidal and Simpson's rule. The key is to choose the sample points and weights carefully so that we can achieve this goal.

For Gauss-Legendre quadrature, we will choose the functions $g(x)$ to be Legendre polynomials.² Legendre polynomials were first introduced in Lecture 9 of the analytical methods portion of this text. As a review, Legendre polynomials, $P_n(x)$, are solutions to Legendre's differential equation:

$$\frac{d}{dx} \left[\left(1 - x^2\right) \frac{d}{dx} P_n(x) \right] + n(n+1)P_n(x) = 0, \quad n = 0, 1, 2, \dots \quad (184)$$

The first two Legendre polynomials are:

$$P_0(x) = 1 \quad P_1(x) = 1$$

and subsequent polynomials can be found using a 3-term recurrence relation:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \quad (185)$$

Legendre polynomials are chosen because they have the following important properties:

1. They are *orthogonal*; meaning the following equation holds:

$$\int_{-1}^1 P_i(x)P_j(x) dx = 0 \quad \text{if } i \neq j$$

This makes the representation of $f(x)$ with a linear combination of Legendre polynomials relatively easy.³

2. They are *complete*, meaning that any “reasonable” function within the interval $x \in [-1, 1]$ can be approximated to arbitrary accuracy by taking a large enough linear combination of Legendre polynomials.

² This is why our method is called *Gauss-Legendre* quadrature. There are several other variations of Gauss quadrature that use other sets of orthogonal polynomials but the theory presented in this lecture is typical.

³ The representation is “easy” in much the same way that it is easier to represent vectors in 3-dimensional space by using linear combinations of vectors $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$ as opposed to any other linear combination of independent but non-orthonormal vectors.

We approximate $f(x)$ as a linear combination of Legendre Polynomials:

$$\begin{aligned} f(x) &\approx \underbrace{c_0 P_0(x) + c_1 P_1(x) + \cdots + c_{n-1} P_{n-1}(x)}_{\text{low order terms}} \\ &\quad + \underbrace{c_n P_0(x) P_n(x) + c_{n+1} P_1(x) P_n(x) + \cdots + c_{2n-1} P_{n-1}(x) P_n(x)}_{\text{high order terms}} \\ &= g(x) \end{aligned}$$

We hope to integrate $g(x)$ exactly, just as with the Newton-Cotes formulas. As always, if $g(x)$ is “close” to $f(x)$, the integral will be a good approximation to the integral of $f(x)$. We will see that this exact integration of $g(x)$ is possible because:

1. We will choose the sample points to be the n roots to the n -th order Legendre polynomial; and
2. The orthogonality property of Legendre polynomials.

Let us carry out the details of this integration:

$$\begin{aligned} \int_a^b f(x) dx &= \int_{-1}^1 f(x(t)) dt \frac{b-a}{2} \approx \int_{-1}^1 g(t) dt \\ \int_{-1}^1 g(t) dt &= c_0 \underbrace{\int_{-1}^1 P_0(t) dt}_{\text{low order terms}} + c_1 \underbrace{\int_{-1}^1 P_1(t) dt}_{\text{low order terms}} + \cdots + c_{n-1} \underbrace{\int_{-1}^1 P_{n-1}(t) dt}_{\text{low order terms}} \\ &\quad + \underbrace{c_n \int_{-1}^1 P_0(t) P_n(t) dt + c_{n+1} \int_{-1}^1 P_1(t) P_n(t) dt + \cdots + c_{2n-1} \int_{-1}^1 P_{n-1}(t) P_n(t) dt}_{\text{high order terms}} \rightarrow 0 \\ &= \sum_{i=0}^{n-1} w_i g(t_i) \\ &= c_0 \underbrace{\sum_{i=0}^{n-1} w_i P_0(t_i)}_{\text{low order terms}} + c_1 \underbrace{\sum_{i=0}^{n-1} w_i P_1(t_i)}_{\text{low order terms}} + \cdots + c_{n-1} \underbrace{\sum_{i=0}^{n-1} w_i P_{n-1}(t_i)}_{\text{low order terms}} + \\ &\quad + \underbrace{c_n \sum_{i=0}^{n-1} w_i P_0(t_i) P_n(t_i) + c_{n+1} \sum_{i=0}^{n-1} w_i P_1(t_i) P_n(t_i) + \cdots + c_{2n-1} \sum_{i=0}^{n-1} w_i P_{n-1}(t_i) P_n(t_i)}_{\text{high order terms}} \rightarrow 0 \end{aligned}$$

The high order integral terms are zero due to the orthogonality property of Legendre polynomials; the high order Gauss quadrature terms go to zero since the sample points, t_i , are the roots of $P_n(t)$. This leaves us with a *linear* equation to solve for the weights:

$$\begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} \begin{bmatrix} P_0(t_0) & P_0(t_1) & \cdots & P_0(t_{n-1}) \\ P_1(t_0) & P_1(t_1) & \cdots & P_1(t_{n-1}) \\ \vdots & \vdots & \ddots & \vdots \\ P_{n-1}(t_0) & P_{n-1}(t_1) & \cdots & P_{n-1}(t_{n-1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} \begin{bmatrix} \int_{-1}^1 P_0(t) dt \\ \int_{-1}^1 P_1(t) dt \\ \vdots \\ \int_{-1}^1 P_{n-1}(t) dt \end{bmatrix}$$

which is equivalent to:

$$\begin{bmatrix} P_0(t_0) & P_0(t_1) & \cdots & P_0(t_{n-1}) \\ P_1(t_0) & P_1(t_1) & \cdots & P_1(t_{n-1}) \\ \vdots & \vdots & \ddots & \vdots \\ P_{n-1}(t_0) & P_{n-1}(t_1) & \cdots & P_{n-1}(t_{n-1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} \int_{-1}^1 P_0(t) dt \\ \int_{-1}^1 P_1(t) dt \\ \vdots \\ \int_{-1}^1 P_{n-1}(t) dt \end{bmatrix} \quad (186)$$

Each row of the matrix on the left hand side of Equation 186 consists of the 0^{th} - through $(n-1)^{\text{th}}$ -order Legendre polynomials sampled at the integration points⁴ and are easy to calculate once you know the values for t_i . The vector on the right hand side is can be shown to be:

$[2, 0, 0, \dots, 0]^T$.⁵ Readers who carefully followed the discussion on quadrature formulas in Lecture 19 will recognize Equation 186. The difference in this case is that, since we have already specified all of the sample points and those sample points are roots of $P_n(t)$, the set of equations is now linear and a unique solution can be found using standard methods of linear algebra.

A MATLAB implementation of the full Gauss quadrature algorithm is shown in the listings below.

```
function [intF, xgl, wgl] = GaussQuad1D(F,a,b,P)
%GaussQuad1D(f,a,b,P) performs P-point Gaussian quadrature
%of function F over interval [a,b]
% input: F - function handle
%         a - lower limit of integration
%         b - upper limit of integration
%         P - # of Gauss Points to use in integration
% output: intF - numeric integral of F
%          xgl - vector of gauss points
%          wgl - vector of weights
%
%
%% Generate Legendre Polynomials of Order o through P
% Store handles to these functions in a cell array.
isQuadSet = false; % flag for special exit of quadrature scheme.
Pn = cell(P+1,1); ❶
Pn{1} = @(x) 1;
Pn{2} = @(x) x;
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

⁴ Which, as a reminder, are equal to the roots of $P_n(t)$

⁵ The first one is easy: $\int_{-1}^1 1 dt = 2$. You can convince yourself that the others must all be zero by expressing each Legendre polynomial as: $P_i(t) = P_0(t)P_i(t)$, since $P_0(t) = 1$. The integrals are now $\int_{-1}^1 P_0(t)P_1(t) dt$, $\int_{-1}^1 P_0(t)P_2(t) dt$, and so on; all of which are equal to zero due to the orthogonality property of the Legendre polynomials.

❶ The Legendre polynomials will be represented as function handles. In MATLAB, a cell array is the appropriate data structure to hold an array of function handles.

```

if P == 0
    error('P must be greater than 0');
elseif P == 1
    xgl = 0; % for P == 1, GQ reduces to midpoint rule
    wgl = 2;
    isQuadSet = true; ②
else
    for n = 2:P
        % use recurrence relation to generate higher
        % order Legendre Polynomials ("Pn functions")
        Pn{n+1} = @(x) ...
            (2*(n-1)+1)*x.*Pn{n}(x)./((n-1)+1) ...
            - (n-1)*Pn{n-1}(x)./((n-1)+1);
    end
end

```

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

② In this case, we need not carry out the rest of the work to find the Gauss points and weights so we set the logical variable isQuadSet to true.

At this point the cell array of Legendre polynomials has been created and we are ready to set up the system of equations so we can solve for the Gauss points (roots of the P -th order Legendre polynomial) and weights and evaluate the quadrature formula.

```

if ~isQuadSet
    %% Compute Roots to the Pth order
    %% Legendre Polynomial

    Tch = @(n) cos(((2*(1:n)) - 1)*pi./(2*n)); ③
    xEst = Tch(P);

    xgl = nan(1,P);
    for r = 1:P
        xgl(r) = fzero(Pn{P+1}, xEst(r)); ④
    end

    if P == 1
        A = xgl(1);
    else
        A = nan(P,P);
        for n = 0:(P-1)
            A((n+1), :) = Pn{n+1}(xgl); ⑤
        end
    end

    k = zeros(P,1); k(1) = 2; ⑥
    wgl = A\k; ⑦
end

%% Change Variables to Scale Interval to [-1,1]
xT = @(t) ((b-a)*t + a + b)/2;
Jac = (b - a)/2;

%% Perform the Integration
intF = F(xT(xgl))*wgl*Jac; ⑧
end

```

34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68

③ Here we use the Chebychev points as an approximation to the P roots of the P -th order Legendre polynomials.

④ Use fzero to find the root of the Legendre polynomial in the vicinity of the Chebychev point.

⑤ Sample the Legendre polynomials at the Gauss points and construct the matrix to solve for the weights.

⑥ Construct the vector on the right hand side of the linear system in Equation 186.

⑦ Solve the linear system, Equation 186, to obtain the weights.

⑧ This line carries out the formula in Equation 183.

Lecture 21 - Monte Carlo Methods for Numeric Integration

Objectives

The objectives of this lecture are to:

- Describe two simple algorithms for carrying out numerical integration with the Monte Carlo method.
- Provide some suggestions as to when different numerical integration algorithms are appropriate.

Introduction

Most every integration problem that you are likely to encounter in your time as an engineer can be solved by one of the methods described in the last two lectures—possibly with some extensions or modification.¹ Still, in the spirit of skinning the cat in as many ways as we can, and to help us acknowledge that some problems can be approached from many different ways, in this lecture we introduce Monte Carlo quadrature.

Monte Carlo quadrature involves solving the integral at hand by changing the problem into a game of chance. There are several textbooks that introduce this method along with Monte Carlo methods for solving a variety of other problems. For an engineer-friendly introduction, see chapter 42 and 43 of the excellent book by Farlow.² For a more in-depth treatment that includes important applications for nuclear engineers, see the text by Dunn.³

Hit or Miss Algorithm

The “Hit or Miss” algorithm is probably the simplest example of a Monte Carlo method for quadrature. The algorithm is as follows:

1. Establish a simple area—a “bounding box”—that bounds the function of interest.

¹ Notably, we have not yet discussed integration over more than one dimension and we have side-stepped issues such as infinite domains of integration or integrals over intervals in which the integrand has one or more singularities. We have ways of dealing with these problems that have been omitted in sympathy for your endurance. We walk before we run.

² Stanley J Farlow. *Partial Differential Equations for Scientists and Engineers*. Courier Corporation, 1993

³ William L Dunn and J Kenneth Shultz. *Exploring Monte Carlo Methods*. Elsevier, 2022

2. Take uniform random samples from points within the bounding box.
3. Determine the fraction of the random samples that fall “under” the function to be integrated.
4. Multiply that fraction by the area of the bounding box; this is your Monte Carlo estimate of the integral.

Example: Use the hit-or-miss method to estimate the following integral:

$$\int_{-3}^3 e^{-x^2} dx$$

1. The function has a maximum value at $x = 0$: $e^{-0^2} = 1$. A logical bounding box would therefore be: $x \in [-3, 3]$, $y \in [0, 1]$. The area of the bounding box is 6.
2. Figure 128 shows a sample of 1000 uniformly distributed points from within this bounding box.
3. Use MATLAB to determine if $y_{\text{sample}} < f(x_{\text{sample}})$. If it is, then the point lies “under” the curve to be integrated. Do this for each sample point and calculate the total fraction under the curve.
4. Multiply the fraction by 6 (area of the bounding box) to obtain the estimated integral.

MATLAB code to carry out such a calculation is shown in the listing below.

```

clear
clc
close all

%% Parameters
f = @(x) exp(-x.^2);
a = -3; b = 3;
fMax = 1; fMin = 0;
N = 1e6; % number of samples

%%

% Compute area of bounding box
Abox = (b-a)*(fMax-fMin);

% Get N uniformly distributed random numbers
% from within the bounding box
x_s = a + (b-a)*rand(N,1);
y_s = fMin + (fMax-fMin)*rand(N,1);

% Determine how many points fall under f(x)
hit = sum(y_s <= f(x_s));

```

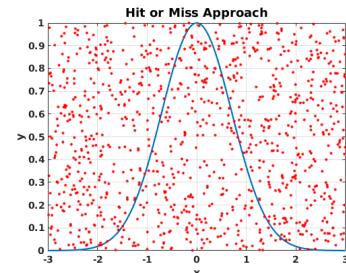


Figure 128: Sample of 1000 uniformly distributed random points from within the box $[-3, 3] \times [0, 1]$.

```
% Calculate ratio of "Hits"
hit_frac = hit/N;

% Estimated integral
MCInt = hit_frac*Abox;

fprintf('Hit or Miss estimated integral = %g \n',MCInt);
```

Naturally, we expect that our approximation of the integral will improve if we take more random samples. This turns out to be true and the convergence behavior is illustrated in Figure 129. Notice that the rate of convergence is quite slow with “half”-order convergence; to reduce error by a factor of 10, you need to do 100 times as much work.

Mean Value Algorithm

One simple optimization will allow us to cut the amount of work we need to do in half. We know from calculus that there is a relationship between the definite integral of a function and its mean value over the interval $x \in [a, b]$.

$$f_{\text{AVG}} = \frac{1}{b-a} \int_a^b f(x) dx \quad (187)$$

We can turn this into a Monte Carlo algorithm by turning the equation around:

1. Sample $f(x)$ uniformly at N random locations within the interval $[a, b]$
2. Approximate f_{AVG} as:

$$f_{\text{AVG}} = \frac{1}{N} \sum_{i=1}^N f(x_i)$$

$$\int_a^b f(x) dx \approx \frac{b-a}{N} \sum_{i=1}^N f(x_i)$$

The nice thing about the Mean Value algorithm is that it requires only half as many random numbers as the Hit or Miss algorithm so it runs roughly twice as fast. The bad part is that it does nothing to improve upon the slow rate of convergence of the Monte Carlo methods.

Good Features of Monte Carlo Integration

For simple one-dimensional integrals of smooth “well-behaved” functions, Monte Carlo is not competitive with the various Newton-Cotes

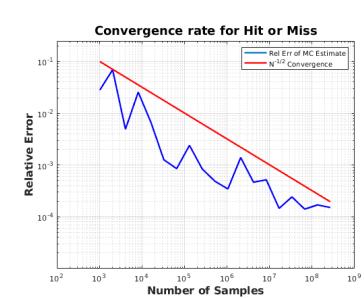


Figure 129: Convergence behavior of the Monte Carlo numeric integration algorithm.

or Gaussian quadrature schemes discussed in previous lectures.

Monte Carlo methods do, however, have some very nice properties which are enough to justify having them in your tool-bag of quadrature methods.

1. **Monte Carlo quadrature works well for non-smooth, non-“well-behaved” functions.** Consider the “wiggly” function shown in Equation 188 and plotted in Figure 130.

$$f(t) = \sin^2\left(\frac{1}{t}\right) \quad (188)$$

Newton-Cotes and Gaussian quadrature algorithms perform very poorly on functions like this since the spacing of sample points must be very tight if the oscillatory behavior of the function is to be captured. The accuracy of Monte Carlo methods, on the other hand, do not depend on being able to construct a suitable approximation of the integrand.

2. **The rate of convergence is independent of the dimensions of integration.** We have not discussed 2D, 3D or more generally N-dimensional integration in any of the last three lectures. Still, these are important problems both for you as well as your friends in other disciplines. For Monte Carlo quadrature, the integration error is always proportional to $1/\sqrt{n}$. The convergence of Newton-Cotes or Gaussian quadrature is actually dependent on the dimensionality. For example, the error the trapezoidal rule is actually proportional to $1/(n^{2/D})$ where D is the number of spatial dimensions. This “curse of dimensionality” is especially important, for example, for certain problems of statistical physics where the integration must be performed over billions of dimensions; each “dimension” representing an interaction between a pair of bodies. In this case, use of any method *other* than Monte Carlo is a practical impossibility.
3. **Monte Carlo methods are very easy to parallelize.** This is important in a day and age when nearly every computer has multiple cores. Devising algorithms that can efficiently use large parallel computers is a difficult task.⁴ Monte Carlo methods fit well in a computational ecosystem adaptable to supercomputers.

Which Method to Use?

Given the variety of integration method discussed over the last three lectures, you might be wondering why we do not simply pick the best one and always use it. The reason, of course, is that which one is best depends on your problem.

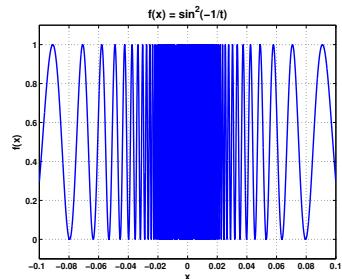


Figure 130: An example of a function that Newton-Cotes and Gauss quadrature do not integrate well.

⁴ And, sadly, a task that is not addressed in this text.

Guidelines:

1. For integrations where you must sample the functions at regular intervals—e.g. if the function you are integrating is known to you only through the data provided by lab instrumentation—then the trapezoidal or Simpson's rule may be the only choices that are reasonable. Pick one.
2. For low-dimensional integrations where the function is reasonably smooth and you can sample the function in any place you wish, use Gauss quadrature. Finite element methods that we will discuss in future lectures use variations of Gauss quadrature extensively because they achieve high accuracy with relatively little work.
3. For high-dimension and/or highly non-smooth functions that you can sample in any place you wish, use Monte Carlo methods.

Lecture 22 - Numerical Quadrature with MATLAB Built-in Functions

Objectives

The objectives of this lecture are to:

- Describe and demonstrate the use of `trapz`.
- Describe and demonstrate the use of `integral`, `integral2`, and `integral3`.

If you are doing your computing in a MATLAB environment, it makes sense to use MATLAB built-in functions for essentially all applications. In this lecture we will explore the essential quadrature tools available in MATLAB.

Integration with TRAPZ

This is the function to use when you must integrate a function from sampled data. Consider the example below.

Example: The flow rate Q , m^3/s , in the channel shown in Figure 131 can be calculated by:

$$Q = \int_0^b v(y) dy$$

where $v(y)$ is the water speed in m/s and $h = 5$ is the overall height of the water. The water speed at different heights are given in Table 17.

MATLAB's primary tool for numeric integration in cases like this is `trapz` which, as the name makes plain, is based on the trapezoidal rule. The call syntax is:

1. $Q = \text{trapz}(Y)$. This form assumes that the data provided in Y is uniformly spaced with unit interval between data points.
2. $Q = \text{trapz}(X,Y)$. With this syntax the user supplies the location of the data points. They can non-uniformly spaced with any interval.

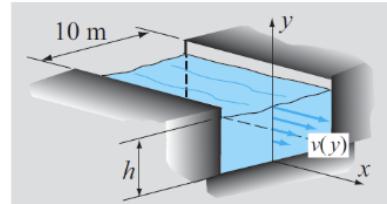


Figure 131: Example water channel dimensions.

y [m]	0	0.3	0.5	1.0	1.5
x [m/s]	0	0.4	0.5	0.56	0.6
y [m]	2	2.5	3	4	5
x [m/s]	0.63	0.66	0.68	0.71	0.74

Table 17: Water speed data taken at various channel depths.

This is, of course, the form that we should use for this example problem.

3. $Q = \text{trapz}(X,Y,\text{dim})$. This form allows for integrals in multiple dimensions. Readers are encouraged to consult the MATLAB documentation for more details.

A straight-forward MATLAB implementation is provided in the listing below.

```
clear
clc
close 'all'

y = [0 0.3 0.5 1 1.5 ...
      2 2.5 3 4 5]; % m, depth
v = [0 0.4 0.5 0.56 0.6 ...
      0.63 0.66 0.68 0.71 0.74];% m/s, speed

h = 10; % m, width of channel.

Q = h*trapz(y,v); % m^3/s, volumetric flow rate
fprintf('Q = %g m^3/s\n',Q);
```

1
2
3
4
5
6
7
8
9
10
11
12
13

Integration with INTEGRAL

The MATLAB function `integral` should be used for numeric integration in one dimension. This function uses adaptive integration over intervals with Gauss quadrature.¹ Interested readers are encouraged to locate and read the MATLAB file in which this built-in function is implemented.² As with other numeric integration algorithms the integral must be carried out over definite bounds but *improper* integrals can be handled automatically.³

Example: Evaluate the the following integral using MATLAB's built-in function `integral`.

$$I = \int_0^\pi \sin^2 x \, dx$$

The basic call syntax is shown in the listing below.

```
clear
clc
close 'all'

f = @(x) sin(x).^2;
a = 0; b = pi;
int_f = integral(f,a,b);
fprintf('I = %g \n',int_f);
```

1
2
3
4
5
6
7
8

As with many of MATLAB's built-in function, a user can further control behavior of the function by specifying name/value pairs for optional arguments, $I = \text{integral}(\text{fun},\text{a},\text{b},\text{Name},\text{Value})$. Two important name/value pairs are:

¹ Lawrence F Shampine. Vectorized adaptive quadrature in MATLAB. *Journal of Computational and Applied Mathematics*, 211(2):131–140, 2008

² As of release 2023b, the function is implemented in MATLAB code and thus the details, in principle, are accessible to users.

³ **Reminder:** an integral is said to be improper when either or both limits of integration are infinite or when the integrand diverges to infinity at one or more points in the range of integration.

1. '`RelTol`'. The corresponding argument should be a non-negative real number; the default value is 10^{-6} . The algorithm estimates the relative error of the computed integral— $|q - Q|/|Q|$, where q is the computed result and Q is the (unknown) exact value.
2. '`AbsTol`'. The corresponding argument should be a non-negative real number; the default value is 10^{-10} .

Note that MATLAB uses both the absolute and relative error tolerance as stopping criteria in the integral implementation. You should generally specify both absolute and relative tolerances if it is important that you obtain greater precision in your results.

Multiple Integrals

In this course we will not go through the details of multi-dimensional numeric integration. MATLAB provides facilitates integration in 2- or 3-dimensions with the built-in functions `integral2` and `integral3`.⁴

Example: Evaluate the integral:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \sqrt{x^2 + y^2 + z^2} e^{-(x^2+y^2+z^2)} dx dy dz$$

Using `integral3`:

```
clear
clc
close 'all'

fun = @(x,y,z) sqrt(x.^2+y.^2+z.^2).*exp(-(x.^2+y.^2+z.^2));
I = integral3(fun,-inf,inf,-inf,inf,-inf,inf);

I_exact = 2*pi;

I_rel_error = abs(I - I_exact)/abs(I_exact);
fprintf('I = %g \n',I);
fprintf('relative error: %g \n',I_rel_error);
```

⁴ Evidently there are no points for creativity in function naming at MathWorks.

It cannot be discerned by this example, but the last six arguments comprise the lower and upper bounds of integration in each dimension—lower and upper bounds in the first dimension, followed by the lower and upper bounds in the second dimension, then the lower and upper bounds in the third dimension. The ordering of dimensions—e.g., in this case, x , y , and z —are the same as the ordering of arguments in the function used for the integrand.

One important detail to remember is that, for integration in polar, cylindrical, or spherical coordinates, the differential volume element is *not* included. This is illustrated in the example below.

Example: Under the assumptions of single-group diffusion theory, the flux, $\phi(r, z)$, in a homogeneous, bare, cylindrical (nuclear) reactor

of radius R and height H is given by:

$$\phi(r, z) = C J_0 \left(\frac{2.405r}{R} \right) \cos \left(\frac{\pi z}{H} \right)$$

Where J_0 is the Bessel function of the first kind of order zero and C is a constant. One parameter of interest is the ratio of the peak to average flux in the reactor. The average flux is:

$$\begin{aligned}\phi_{\text{avg}} &= \frac{\int \phi dV}{V} \\ &= \frac{\int_{-H/2}^{H/2} \int_0^R \int_0^{2\pi} \phi(r, z) r d\theta dr dz}{\pi R^2 H} \\ &= \frac{2\pi \int_{-H/2}^{H/2} \int_0^R \phi(r, z) r dr dz}{\pi R^2 H}\end{aligned}$$

and the peak flux ϕ_{peak} is equal to C . With this information, we can use MATLAB to calculate the ratio of peak to average flux:

```
clear
clc
close 'all'

R = 3; H = 6;
C = 1;
P = @(r,z) C*besselj(0,2.405*r/R).*cos(pi*z/H);

totP = 2*pi*integral2(@(r,z) P(r,z).*r,o,R,-H/2,H/2); ❶
V = pi*(R.^2)*H;
AvgP = totP/V;
fprintf('Peak/Avg power = %g \n',C./AvgP);
```

Note the extra factor of r for the differential volume element: $r d\theta dr dz$.

❶ Here the extra factor of r is for the differential volume. Hopefully it is clear that MATLAB does not do this automatically; it is incumbent upon the user to include details like this.

The output of this script is approximately 3.6387 which is a standard result in reactor physics based on diffusion theory.

Assignment #7

1. Write a MATLAB user-defined function that determines the first derivative of a function that is given by a set of discrete points with equal spacing. For the function name, use $yd = \text{FirstDeriv}(x,y)$. The input arguments x and y are vectors with the coordinates of the points and the output argument yd is a vector with the values of the derivative at each point. At the first and last points, the function should calculate the derivative with three-point forward and backward difference formulas, respectively. At all other points, FirstDeriv should use the two-point centered difference formula. Use FirstDeriv to calculate the derivative of the function given by the following tabulated data:

x	1.1	1.2	1.3	1.4	1.5
$f(x)$	0.6133	0.7822	0.9716	1.1814	1.4117

Output the value of df/dx at $x = 1.3$.

2. Write a MATLAB user-defined function that calculates the second derivative of a function that is given by a set of discrete data points with equal spacing. For the function name and arguments use $ydd = \text{SecDeriv}(x,y)$, where the input arguments x and y are vectors with the coordinates of the points, and ydd is a vector with the values of the second derivative at each point. For calculating the second derivative, the function SecDeriv should use the finite difference formulas that have a truncation error of $\mathcal{O}(h^2)$. Use SecDeriv to calculate the second derivative of the function that is given by the tabulated data below:

x	-1	-0.5	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5
$f(x)$	-3.632	-0.3935	1	0.6487	-1.282	-4.518	-8.611	-12.82	-15.91	-15.88	-9.402	9.017

Output the value of d^2f/dx^2 at $x = 2.5$.

3. A radar station is tracking the motion of an aircraft. The recorded distance to the aircraft, r , and the angle θ during a period of 60 seconds is given in the table below. The magnitude of the instantaneous velocity and acceleration of the aircraft can be calculated by:

$$v = \sqrt{\left(\frac{dr}{dt}\right)^2 + \left(r \frac{d\theta}{dt}\right)^2}, \quad a = \sqrt{\left[\frac{d^2r}{dt^2} - r \left(\frac{d\theta}{dt}\right)^2\right]^2 + \left[r \frac{d^2\theta}{dt^2} + 2 \frac{dr}{dt} \frac{d\theta}{dt}\right]^2}$$

Determine the magnitudes of the velocity and acceleration at the times given in the table. Plot the velocity and acceleration versus time on the same plot with two different y-axes using the built-in MATLAB tool yyaxis. Calculate the derivatives using the FirstDeriv and SecDeriv functions you developed for the previous two problems.

t [s]	0	4	8	12	16	20	24	28
r [km]	18.803	18.861	18.946	19.042	19.148	19.260	19.376	19.495
θ [rad]	0.7854	0.7792	0.7701	0.7594	0.7477	0.7350	0.7215	0.7073
t [s]	32	36	40	44	48	52	56	60
r [km]	19.617	19.741	19.865	19.990	20.115	20.239	20.362	20.484
θ [rad]	0.6925	0.6771	0.6612	0.6448	0.6280	0.6107	0.5931	0.5750

3. Write a user-defined MATLAB function to integrate a function $f(x)$ that is given in a set of n discrete points using the trapezoidal rule. Your function should *not* require that the points be evenly spaced. For the function name and arguments use `I=IntPointsTrap(x, y)`, where the input arguments x and y are vectors with the values of x and the corresponding values of $f(x)$, respectively. The output argument I is the value of the integral. Use the function to estimate the surface area and volume of a wine barrel as illustrated in Figure 132. The diameter of the barrel is measured at the points provided in the table below. The surface area, S , and volume, V , can be determined by:

$$S = 2\pi \int_0^L r \, dz \quad \text{and} \quad V = \pi \int_0^L r^2 \, dz$$

Your script should provide a properly formatted output of the surface area and volume of the barrel.

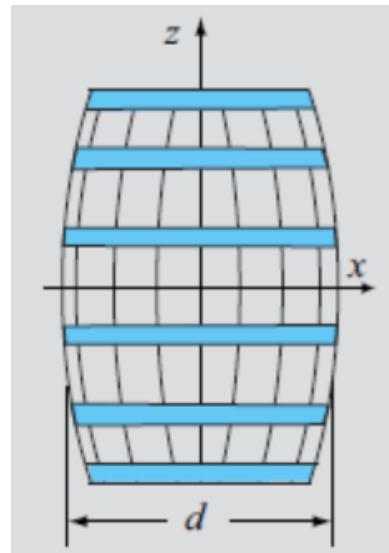


Figure 132: Schematic of wine barrel.

4. Write a user-defined MATLAB function that uses Simpson's rule to integrate a function $f(x)$ that is given in a set of n evenly spaced points. For the function name and arguments use `I=SimpsonPoints(x, y)`, where the input arguments x and y are vectors with the values of x and the corresponding values of $f(x)$, respectively. The output argument I is the value of the integral. Use the function to compute $\int_0^{1.8} f(x) dx$ with the tabulated data below:

x	0	0.3	0.6	0.9	1.2	1.5	1.8
$f(x)$	0.5	0.6	0.8	1.3	2	3.2	4.8

Your script should provide a properly formatted output of the value of the integral.

5. Write a user-defined MATLAB function to integrate a function, $f(x)$, in the domain $x \in [a, b]$, with five-point Gauss quadrature. For the function name and arguments use `I=GaussQuad5ab(fun,a,b)`, where `fun` is a handle to the function that is being integrated, `a` and `b` are the lower and upper bounds of the integral, and `I` is the value of the integral. Use the function to calculate the following integral:

$$\int_0^3 e^{-x^2} dx$$

Your script should provide a properly formatted output of the value of the integral.

Part XI

Solving Initial Value Problems

Lecture 23 - ODE Review and Euler's Method for IVPs

Objectives

The objectives of this lecture are to:

- Review some terminology and concepts regarding differential equations.
- Describe Euler's method for 1st order initial value problems (IVPs) and demonstrate convergence behavior.
- Describe and demonstrate the modified Euler's method.

Ordinary Differential Equations (ODEs) Review

It almost seems appropriate to begin this section with a brief apology. Elementary differential equations, such as what most undergraduate engineering majors study, are more fully reviewed in lectures 1 through 6 in the analytical methods portion of this text. I intend in this section only a brief reminder of selected categories by which a given ordinary differential equation can be classified. So from that perspective, the discussion I am about to begin will be redundant and I apologize for that.

In addition to being redundant, the classification schemes I will present—which inherently smack of pedantry—will turn out to be of limited relevance in the context of the *numerical* methods we are about to study; for most classes of ODEs many numerical methods will suffice and differ only in their complexity and convergence properties. Nonetheless, when learning these new methods, it will be important to select problems that we *can* solve analytically so we know whether or not the solution our numerical method produces is correct. Thus we will review our classification of ODEs so we can discriminate between problems we can and cannot solve analytically; we will choose the former for our test cases to validate implementation of the numerical method.

Classification of Differential Equations

In the opening six lectures of this text, a number of classification schemes were used to characterize an ODE and determine which method should be used in finding its solution.

1. *type*. An equation where the solution that we seek is only a function of a single independent variable and only ordinary derivatives are involved is called an *ordinary* differential equations. Otherwise, if there are multiple independent variables and partial derivatives, it is called a *partial* differential equation.
2. *separability*. An ODE is separable if it can be written in the form:

$$\frac{du}{dx} = f(u)g(x)$$

If it is, then the equation is formally *separated* and integrated to find a solution.¹

3. *linearity*. An ODE is *linear* if each term in the ODE involving the *dependent variable* is linear—i.e. not raised to some power or included as an argument to a trigonometric function—and also the dependent variable is not multiplied by one of its derivatives.

From that description it should be apparent that the following ODEs are linear:

- (a) $u'' + u = 0$
- (b) $u' - \cos(x)u = x^2$

and that, by contrast:

- (a) $u'' - uu' = 0$;
- (b) $u' - u^2 = f(x)$

are both nonlinear. If an equation is nonlinear, we usually cannot solve it analytically unless it also happens to be separable.

4. *homogeneity*. An ODE is *homogeneous* if each non-zero term in the equation involves the dependent variable or one of its derivatives. Otherwise it is *non-homogeneous*. Thus:

- (a) $u' - xu = 0$; and
- (b) $uu'' = u' + 5u$

are homogeneous, while

- (a) $u^{\text{prime}} - 3x = 0$; and
- (b) $u'' - x^2u' = \sin(x)$

are non-homogeneous.

¹ Please see Lecture 2 of the Analytical Methods portion of this text for more details.

Examples: Classify the following ODEs by order, type, linearity, and homogeneity:

1. $\frac{d^2y}{dx^2} = e^{3x} - y$
2. $\frac{d^2y}{dt^2} + 5\left(\frac{dy}{dt}\right)^2 - 6y = 0$
3. $\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} + S(x, t)$

Answers:

1. 2nd-order, ordinary, linear, non-homogeneous
2. 2nd-order, ordinary, nonlinear, homogeneous
3. 2nd-order, partial, linear, non-homogeneous

Initial Value Problems (IVPs) and Boundary Value Problems (BVPs)

From now until further notice, we will restrict our attention to ordinary differential equations. As readers are most likely aware, in order to find a unique solution for, say, a 2nd-order ODE, two conditions—values for the dependent variable and/or its derivative—must also be provided. We will further classify a problem based on the manner in which these conditions are provided.

Initial Value Problem

An initial value problem is generally of the form:

$$a_n y^{(n)} + a_{n-1} y^{(n-1)} + \cdots + a_1 y' + a_0 y = g(x)$$

along with the following conditions:

$$y(x_0) = y_0, \quad y'(x_0) = y_1, \quad \dots, \quad y^{(n-1)}(x_0) = y_{n-1}$$

where x_0 is within the domain of interest. It is not required, but it is customary that x_0 is at one boundary of the domain, typically $x_0 = 0$.² The coefficients a_n can be constants, functions of the independent variable, or, in the general non-linear case, functions of both the dependent and independent variable. In the case where a_n are constant or functions only of the independent variable, where a_n and $g(x)$ are continuous, and $a_n \neq 0$ throughout the interval of interest, then a unique solution is assured.

For the lectures that follow, we will restrict our attention to initial value problems of the form given in Equation 189.

$$y' = f(x, y) \quad y(x_0) = y_0 \quad (189)$$

This formulation is sufficiently general to capture all problems of interest for this class.

Boundary Value Problem

For the purposes of this discussion, we will restrict ourselves to linear 2nd-order boundary value problems of the form given in Equation 190.

$$a_2(x)y'' + a_1(x)y' + a_0(x)y = g(x), \quad y(a) = y_1, \quad y(b) = y_2, \quad a \neq b \quad (190)$$

Initial value problems will be the subject of Lectures 23 - 27. Most numerical methods for IVPs can be broken down into *single-step* methods or *multi-step* methods. In this class, we will focus exclusively on single-step methods. We will study a range of different numerical methods for solving BVPs starting in Lecture 28 and continuing through the end of the course.³

² Of course, it is also very common that the independent variable for initial value problems is *time*, denoted by t and that $t_0 = 0$.

³ Alert readers may be wondering: "But what about initial boundary value problems?" I am sorry to say that we will not be addressing problems of that type in this class.

Euler's Explicit Method for 1st-Order IVPs

Consider the problem below:

$$y' = f(x, y), \quad y(x_0) = y_0 \quad (191)$$

The fact that $y(x)$ is differentiable, and therefore continuous, is implied in the problem statement. Such functions can be expressed in a Taylor series expansion as shown below.

$$y(x_{(i+1)}) = y(x_i) + y'(x_i)(x_{(i+1)} - x_i) + \frac{y''(x_i)}{2!}(x_{(i+1)} - x_i)^2 + \cdots + \frac{y^{(n)}(x_i)}{n!}(x_{(i+1)} - x_i)^n + \cdots$$

If we assume $x_{(i+1)} - x_i = h$ and truncate the expansion we arrive at the following expression:

$$y(x_{(i+1)}) = y(x_i) + y'(x_i)h + \frac{y''(\xi)}{2}h^2$$

where $\xi \in [x_i, x_{(i+1)}]$. Substituting in Equation 191, we get the basic theoretical result underpinning Euler's explicit method:

$$y(x_{(i+1)}) = y(x_i) + f(x, y)h + \frac{y''(\xi)}{2}h^2$$

which we shorten somewhat for notational simplicity as:

$$y_{i+1} = y_i + hf(x_i, y_i) + \underbrace{\frac{y''(\xi)}{2}h^2}_{\text{local truncation error}} \quad (192)$$

Note that the local truncation error is incurred on every interval and accumulates; since y_{i+1} is not exact, $f(x_{i+1}, y_{i+1})$ is different from $f(x_{i+1}, y_{i+1}^*)$, where y_{i+1}^* is the exact solution at x_{i+1} . This introduces additional error as the algorithm progresses. Consequently the *global* truncation error is $\mathcal{O}(h)$ and we say that the method is 1st-order convergent.⁴

Algorithm:

1. Discretize the independent variable, $x \in [a, b]$ into N intervals of equal size: $h = (x_{i+1} - x_i) = \frac{b-a}{N}$ ⁵
2. Set $y(x_0) = y_0$ where both x_0 and y_0 are given in the problem statement.
3. Set $x_{i+1} = x_i + h$
4. Evaluate: $y_{i+1} = y_i + hf(x_i, y_i)$
5. Repeat steps 3 and 4, N times.

⁴ This is an admittedly "hand-wavy" explanation even though it happens to be correct. Dissatisfied readers are encouraged to dig deeper in the references.

⁵ Often the problem statement for initial value problems will identify one end of the domain but leave the other end unspecified—e.g. $x > 0$. For practical purposes we will always assume a finite domain for the independent variable.

Let us illustrate the algorithm with a simple example.

Example: Consider the following IVP:

$$y' = \frac{x^2}{y}, \quad 0 < x < 2, \quad y(0) = 2$$

This first order, linear, homogeneous equation is separable and one can show⁶ that the solution is:

$$y_{\text{exact}}(x) = \sqrt{2/3x^3 + 4} \quad (193)$$

This algorithm is implemented in the MATLAB listing below. We discretize the domain into $N = 30$ intervals.

```

1 clear
2 clc
3 close 'all'

4 %% Define the problem to be solved.
5 f = @(x,y) (x.^2)./y;
6 y_exact = @(x) sqrt((2/3)*x.^3 + 4); ❶
7 xMin = 0; xMax = 2.0;

8 %% Euler Explicit Demonstration
9
10 N = 30;
11 x = linspace(xMin,xMax,N+1);
12 y_ns = nan(1,N+1); ❷
13 y_ns(1) = 2;
14 h = x(2)-x(1);
15 for t = 1:N ❸
16     y_ns(t+1) = y_ns(t)+f(x(t),y_ns(t))*h;
17 end

18 %% Plot the Solution
19 x_gold = linspace(xMin,xMax,1000); ❹
20
21 figure(1)
22 plot(x,y_ns,'-b',...
23       x_gold,y_exact(x_gold),'-r',...
24       'linewidth',3);
25 title("Solution with Euler's Method", 'fontsize',14,...
26       'fontweight','bold');
27 xlabel('X','fontsize',12,'fontweight','bold');
28 grid on;
29 set(gca,'fontsize',10,'fontweight','bold');
30 legend('Euler Explicit','Exact Solution');
31
32
33

```

❶ Make a handle to the exact solution so we can compare with our numeric solution. This is a strongly-advised safety tip.
❷ The variable y_{ns} represents the numeric solution. Here we pre-allocate an array to hold the numeric solution.
❸ With this loop we repeat the iteration $N = 30$ times.
❹ We will use a more refined set of points to plot the exact solution to ensure we are accurately representing its shape.

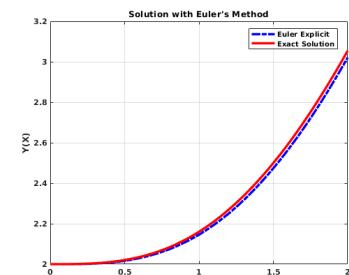


Figure 133: Approximate solution of example problem using Euler's explicit method with $N = 30$.

A plot of the numeric solution along with the exact solution is shown in Figure 133. The numeric solution clearly has errors which, of course, we can reduce if we increase N . The convergence behavior of Euler's explicit method is shown in Figure 134.

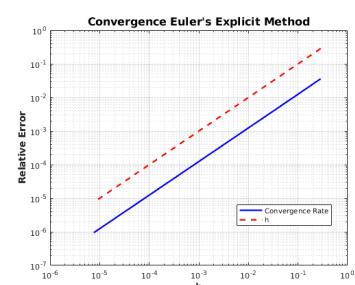


Figure 134: Convergence behavior of Euler's explicit method for example problem.

Stiff Equations

ONE DISADVANTAGE of Euler's explicit method for solving IVPs is that for some problems local truncation error, instead of building up steadily, amplify exponentially and the solution "blows up." As an example, consider the following IVP:

$$y' = -100[y - \cos(x)] - \sin(x), \quad y(1) = 1, \quad 0 < x < 2 \quad (194)$$

The reader is encouraged to verify that the exact solution to this problem is $y^*(x) = \cos(x)$. If we discretize the domain into $N = 95$ intervals and solve using Euler's explicit method, we get the result shown in Figure 135.

You should look at that figure and recognize that whatever is going wrong, it is more than just the accumulation of local truncation error, but is instead a sign of incipient instability. If you reduce N further the numeric result blows up and produces a meaningless result.

The example given in Equation 194 is referred to as a "stiff" differential equation. Even though the actual solution is quite smooth, a large number of intervals is required in order to obtain a reasonable numeric solution. Often it is the case, as it is now, that we can address the stiffness problem by simply increasing N and thereby reducing our step size. Eventually a threshold is passed where the numeric solution is stable and converges as expected. In some cases, however, evaluating $f(x, y)$ is computationally expensive and "simply" reducing the step size has unacceptable costs. A conceptually simple change to the algorithm can fix this problem.

Euler's Implicit Method for 1st-Order IVPs

This method relies on the same theoretical development as the explicit Euler method. The difference is that, when we solve for y_{i+1} , we will use $f(x_{i+1}, y_{i+1})$:

Algorithm:

1. Discretize the independent variable, $x \in [a, b]$ into N intervals of equal size: $h = (x_{i+1} - x_i) = \frac{b-a}{N}$.
2. Set $y(x_0) = y_0$ where both x_0 and y_0 are given in the problem statement.
3. Set $x_{i+1} = x_i + h$
4. Evaluate: $y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$

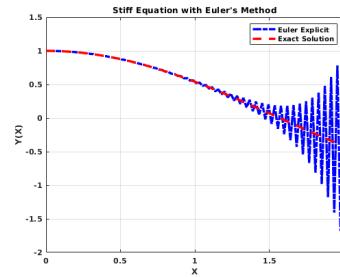


Figure 135: Result of attempting to solve a "stiff" differential equation with Euler's explicit method, $N = 95$.

5. Repeat steps 3 and 4, N times.

Note that step 4 of the algorithm requires solving a nonlinear equation (or system of equations). We have studied methods of solving nonlinear equations previously in this course and we know it is not a trivial problem. Also, this method is still only 1st-order convergent like its explicit sibling. The main benefit is that this algorithm will maintain stability even for stiff problems like the last example.

A straight-forward MATLAB implementation is provided in the listing below.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

clear
clc
close 'all'

%% Set up the Problem
f_stiff = @(x,y) -100*(y-cos(x))-sin(x);
f_stiff_exact = @(x,y) cos(x);

N = 10
x = linspace(xMin,xMax,N+1);
y_ns = nan(1,N+1);
y_ns(1) = 1;
h = x(2)-x(1);

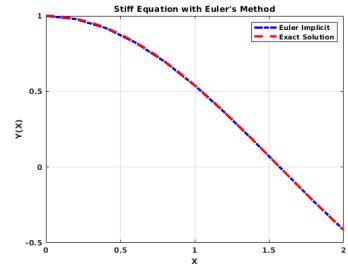
%% Solve using Implicit Euler's Method
options = optimoptions('fsolve','Display','none'); ⑤
for t = 1:N-1
    fe_fun = @(y) y - y_ns(t) - f_stiff(x(t+1),y)*h;
    y_ns(t+1) = fsolve(fe_fun,y_ns(t),options); ⑥
end

%% Plot the result
plot(x,y_ns,'-b',...
    x_gold,f_stiff_exact(x_gold),'--r',...
    'LineWidth',3);
title("Stiff Equation with Euler's Method",'FontSize',14,...
    'FontWeight','bold');
xlabel('X','FontSize',12,'FontWeight','bold');
grid on;
set(gca,'FontSize',10,'FontWeight','bold');
ylabel('Y(X)','FontSize',12,'FontWeight','bold');
grid on;
set(gca,'FontSize',10,'FontWeight','bold');
legend('Euler Implicit','Exact Solution');

```

⑤ Create an options structure that will suppress extraneous output of fsolve.

⑥ Call fsolve to find $y_{ns}(t+1)$.



The resulting solution is shown in Figure 136. The performance trade-offs are hard to measure in general. Depending on how many evaluations of $f(x,y)$ are required to solve the nonlinear equations and depending on how stiff the equation is, implicit Euler's method may be less attractive than simply increasing N and trying again with Euler's explicit method. This trade-off will be different for each problem.

Figure 136: Solution of a stiff IVP with Euler's implicit method with $N = 10$.

Modified Euler's Method

Whether you are using the implicit or explicit version of Euler's method, they both exhibit 1st-order convergence which is slow. Can we do better? The answer is: of course. The main assumption in Euler's method is that the slope remains constant throughout each interval. This is the major source of error and the modified Euler's method partially corrects for this error.

Algorithm:

1. Discretize the independent variable, $x \in [a, b]$ into N intervals of equal size: $h = (x_{i+1} - x_i) = \frac{b-a}{N}$.
2. Set $y(x_0) = y_0$ where both x_0 and y_0 are given in the problem statement.
3. Set $x_{i+1} = x_i + h$
4. Calculate $f(x_i, y_i)$
5. Estimate y_{i+1} using Euler's explicit method: $y_{i+1}^{\text{EE}} = y_i + hf(x_i, y_i)$
6. Calculate $f(x_{i+1}, y_{i+1}^{\text{EE}})$
7. Find better estimate of y_{i+1} by averaging the two slopes:

$$y_{i+1} = y_i + \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{\text{EE}})}{2}h$$

8. Repeat steps 3 through 7, N times.

One can show that this algorithm reduces local truncation error to $\mathcal{O}(h^3)$ and the corresponding global truncation error is $\mathcal{O}(h^2)$, thereby obtaining quadratic convergence. The listing below shows a MATLAB implementation of steps 3 through 7 and Figure 137 shows the 2nd-order convergence behavior.

```

for t = 1:N
    f_xy = f(x(t),y_ns(t));
    % initial Euler step
    y_ns_EU = y_ns(t) + f_xy*h;

    % Apply Mod Euler
    y_ns(t+1) = y_ns(t) + ...
        (f_xy + f(x(t+1),y_ns_EU))*h/2;
end

```

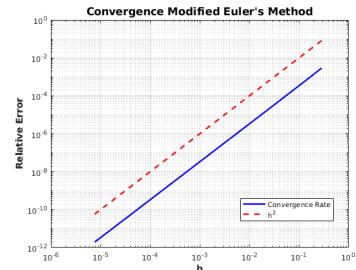


Figure 137: Quadratic convergence of the modified Euler method.

Lecture 24 - Solving Systems of 1st-Order IVPs

Objectives

The objectives of this lecture are to:

- Apply Euler's explicit method to solve systems of 1st-order IVPs.
- Show how to convert higher order IVPs into systems of 1st-order IVPs.

Systems of 1st-Order IVPs with Euler's Explicit Method

So far we have only addressed scalar first-order equations. There are many physical phenomena that, in order to properly describe the relevant physics, must be modeled as a system of equations. In this section we will consider the dynamics of the isotope xenon-135 in nuclear reactors.

Xenon-135 Background

Most nuclear reactors in the world today generate power through the fission of ^{235}U . Each fission results in the release of approximately 185 MeV of recoverable energy,¹ two or three neutrons, one or more of which is expected to also cause a fission thus propagating the chain reaction, and two fission products. Almost all of the resulting fission products are radioactive, needing to undergo a series of beta- and alpha-decay processes to achieve a nuclear configuration that is stable. This process is depicted in Figure 138.

Some of these fission products (and their subsequent decay offspring) have a significant impact on the nuclear chain reaction that goes on around them. One class of fission products very influential in this way are those that have a strong tendency to absorb neutrons without undergoing fission. These fission products are sometimes referred to as *poisons*, owing to the fact that neutron absorption "poisons" the chain reaction process by preventing the absorbed neutron from going on to cause fission of an atom of fuel.

¹ MeV stands for "mega-electron-volt", or 1×10^6 eV. 1eV is equivalent to 1.6×10^{-19} Joules. It takes a lot of fissions to produce a discernible amount of power.

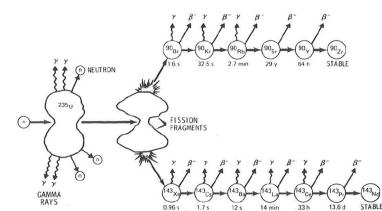


Figure 138: Schematic of representative fission product decay process.

The particular fission product decay chain that produces ^{135}Xe is illustrated in Figure 139. Tellurium-135 (^{135}Te) is produced directly from fission. It undergoes beta-decay to iodine-135 (^{135}I) with a 19-second half-life. ^{135}I in turn decays to xenon-135 (^{135}Xe) with a slower half-life of 6.6 hours. ^{135}Xe is also produced in significant quantities as a fission product. ^{135}Xe undergoes both beta-decay to cesium-135 (^{135}Cs) as well as neutron absorption—denoted with the (n, γ) symbol for radiative capture—to become ^{136}Xe which is stable but a very weak neutron absorber. ^{135}Cs decays to barium-135 (^{135}Ba) which is a strong neutron absorber but, since the half-life is 2.3 million years, not enough ^{135}Ba builds up in the core to have an impact on the kinetic behavior of the fission process.

Model of ^{135}Xe Concentration

We can model the atom density of ^{135}I and ^{135}Xe with a first-order system of differential equations.

$$\begin{aligned} \frac{dI}{dt} &= \underbrace{\gamma^{\text{Te}} \Sigma_f \phi}_{\text{production from fission}} - \underbrace{\lambda^{\text{I}} I}_{\text{loss from decay}} \\ \frac{d\text{Xe}}{dt} &= \underbrace{\gamma^{\text{Xe}} \Sigma_f \phi}_{\text{production from fission}} + \underbrace{\lambda^{\text{I}} I}_{\text{production from iodine decay}} - \underbrace{\lambda^{\text{Xe}} \text{Xe}}_{\text{loss from xenon decay}} - \underbrace{\text{Xe} \sigma_a \phi}_{\text{loss from neutron capture}} \end{aligned}$$

This system can be solved using Euler's explicit method more-or-less in the same way that a single equation can be solved with the following equations:

$$\begin{aligned} y(t) &= \begin{bmatrix} I(t) \\ \text{Xe}(t) \end{bmatrix} \\ y'(t) &= \begin{bmatrix} I(t) \\ \text{Xe}(t) \end{bmatrix}' = \begin{bmatrix} \gamma^{\text{Te}} \Sigma_f \phi - \lambda^{\text{I}} y(1, t) \\ \gamma^{\text{Xe}} \Sigma_f \phi + \lambda^{\text{I}} y(1, t) - \lambda^{\text{Xe}} y(2, t) - y(2, t) \sigma_a \phi \end{bmatrix} \end{aligned}$$

and the time-stepping formula would be:

$$y(:, t+1) = y(:, t) + y'(:, t) dt$$

MATLAB Implementation

An analysis of ^{135}Xe concentration through a typical power transient comprising a reactor start-up, a period of full power during which

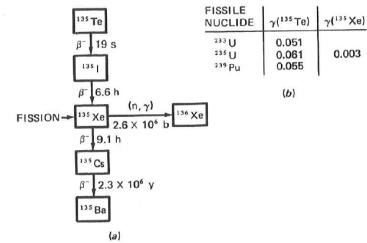


Figure 139: Fission product decay chain for generating xenon-135.

Owing to the short half-life of ^{135}Te we assume that it immediately decays and thus, effectively, is a direct production term for ^{135}I .

Nomenclature:

1. I - iodine-135 atom density
2. Xe - xenon-135 atom density
3. ϕ - neutron flux. Flux is proportional to reactor power; if flux is higher, reactor power is higher.
4. γ - fission yield. The fraction of fissions that result in a particular fission product.
5. λ - decay constant which is related to half-life: $\lambda = \ln(2)/t_{1/2}$
6. σ_a - microscopic absorption cross-section. Represents the probability that an incident neutron will be absorbed. Neutron "poisons" have large values of σ_a .

Note: Here we adopt the MATLAB syntax for vectors. We also use dt to indicate the step-size.

xenon levels approach equilibrium, and then a brief shutdown followed by a start-up 20 hours later is shown in the next few listings.

We start by clearing the environment and providing necessary nuclear data.

```

clear
clc
close 'all'

%% Nuclear Data
nominalFlux = 2e14; % n/cm^2-s
flux = @(t) nominalFlux*power_profile_Xe(t); ❶
Sigma_F = .0452; %1/cm
% Iodine-135 and Tellurium-135 Nuclear data
gamma_Te = 0.061;
lambda_I = 2.9173e-5; %1/s

% Xenon-135 Nuclear data
sigma_a_Xe = 2.6e6*10^(-24); % cm^2
gamma_Xe = 0.003;
lambda_Xe = 2.1185e-5; % 1/s

```

❶ We use a local function (defined below) to specify the time-dependent flux profile for the transient of interest.

Next we will set time intervals for the explicit Euler method and initialize data arrays.

```

%% Time discretization
tStart = 0; % sec - time start
tEnd = 160*3600; %sec - time end (160 hours)
numTs = 50000;
tSpace = linspace(tStart,tEnd,numTs);
dT = tSpace(2)-tSpace(1);

%% Construct data arrays and provide initial conditions
P = nan(2,numTs+1); % an extra column for the initial data
P(1,1) = 0; % initial I-135 concentration;
P(2,1) = 0; % initial Xe-135 concentration;
pfrac = nan(1,numTs); % power fraction

```

Now we are ready to commence time stepping.

```

%% Commence time stepping
for ts = 1:numTs

    if mod(ts,10000)==0 ❷
        fprintf('Commencing time step %i.\n',ts);
    end

    dP = nan(2,1);
    T = tSpace(ts);
    pfrac(ts) = power_profile_Xe(T);
    % update Iodine concentration
    dP(1) = gamma_Te*Sigma_F*flux(T) - lambda_I*P(1,ts); ❸
    % update Xenon concentration
    dP(2) = gamma_Xe*Sigma_F*flux(T) + lambda_I*P(1,ts) ...
        - P(2,ts)*sigma_a_Xe*flux(T) - lambda_Xe*P(2,ts);

    % update the total poison concentration.
    P(:,ts+1) = P(:,ts) + dP*dT;
end

```

❷ For scripts that require more than a few seconds to run, it is a good practice to provide some intermediate output to let the user know that "something is happening." At the same time, you also do not want to flood the command window with output. Here we choose to provide a progress update every 10000 time steps.

❸ The notation here is somewhat clunky but these lines effectively define y' .

Once the calculations are done, we will plot the results. For this transient analysis the output is shown in Figure 140.

```
%>% Plot your results
figure(1)
subplot(2,1,1)
plot(tSpace/3600,pfrac,'LineWidth',2);
axis([0 160 -0.1 1.1]);
title('Power Profile')
set(gca,'FontSize',14,'FontWeight','bold');
ylabel('Power Fraction',...
      'FontWeight','bold','FontSize',14);

subplot(2,1,2)
h = semilogy(tSpace/3600,P(2,1:(end-1)));
set(h,'LineWidth',2);
set(gca,'FontSize',14,'FontWeight','bold');
axis([0 160 2*10^14 2*10^16])
grid on
xlabel('Time (h)', 'FontWeight','bold','FontSize',14)
ylabel('Xenon-135 (at/cm^3)',...
      'FontWeight','bold','FontSize',14);
title('Xenon-135 Concentration','FontSize',14,...,
      'FontWeight','bold')
```

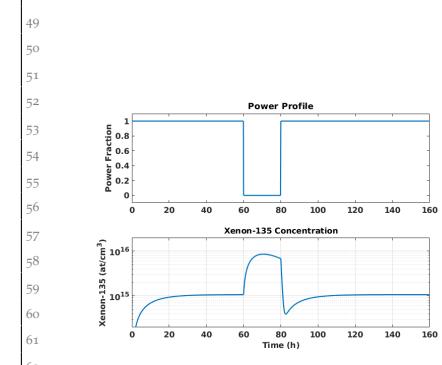


Figure 140: Xenon-135 concentration during a transient.

The power profile for the transient is encoded in a local function as shown below.

```
%>% Local Functions
function p = power_profile_Xe(t)
% returns a power level [0,1] giving the percent full power
tH = t/3600; % convert seconds to hours.
if tH < 60
    p = 1;
elseif tH>=60 && tH<80
    p = 0;
else
    p = 1;
end
end
```

Convert High-Order IVPs into a System of 1st-Order IVPs

Now that we have introduced a few methods for numerically solving initial value problems, some readers may be starting to wonder when we will include methods tailored to solve higher order IVPs. The (possibly) surprising answer is that we will *not* present new methods for solving higher-order IVPs. Instead, *all* of the numerical methods we present for IVPs will be for 1st-order IVPs; higher-order IVPs will simply be re-stated in terms of a *system* of 1st-order IVPs.

Consider the following 2nd-order IVP that captures Newton's laws of motion for a rocket test facility, idealized as a simple spring-mass

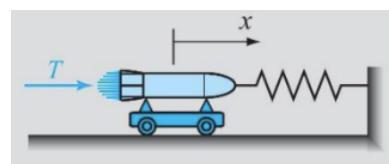


Figure 141: A rocket test facility idealized as a spring-mass system.

system as shown in Figure 141.

$$m(t) \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = T$$

$$m(t) = m_0 - mt, \quad x_0 = 0, \quad v_0 = x'(0) = 0$$

In this expression, c is a constant damping coefficient and k the spring constant. Since the rocket consumes fuel while operating, the mass of the rocket is a function of time as shown. Lastly, initial displacement, x_0 , and velocity, x' , are given. By inspection we can see that so long as $m(t) \neq 0$ this IVP will have a unique solution.

We will convert this 2nd-order IVP to a system of two 1st-order IVPs as follows:

1. Write the governing equation in the form shown below:

$$x'' = -\frac{c}{m}x' - \frac{k}{m}x + \frac{T}{m}$$

2. Define a new vector of dependent variables:

$$w = \begin{bmatrix} x \\ x' \end{bmatrix}$$

3. Write higher-order equation in terms of new vector of dependent variables:

$$w' = \begin{bmatrix} x \\ x' \end{bmatrix}' = \begin{bmatrix} x' \\ x'' \end{bmatrix} = \begin{bmatrix} w(2) \\ -\frac{c}{m}w(2) - \frac{k}{m}w(1) + \frac{T}{m} \end{bmatrix}$$

4. Solve for w with your favorite method.

A MATLAB script to solve this problem is shown in the listings below. We start by clearing out the environment and defining necessary variables.

```

clear
clc
close 'all'

%% Example #2
xo = 0; % initial displacement
vo = 0; % initial velocity

tMax = 5; % seconds
nT = 100000;
T = linspace(0,tMax,nT);
w = nan(2,nT);
w(:,1) = [xo;vo]; % set initial conditions
dt = T(2)-T(1);

```

We will use Euler's explicit method to solve for $w(t)$. The results are plotted and shown in Figure 142.

```

16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
for t = 1:(nT-1)
    % say something comforting about program progress
    if mod(t,10000)==0
        fprintf('Commencing time step %i.\n',t);
    end
    dw = ex2(T(t),w(:,t));
    w(:,t+1) = w(:,t) + dw*dt;
end

x = w(1,:); %ft , displacement
v = w(2,:); %ft/s, velocity

%% plot the results
figure(2)
plot(T,x,'-c','LineWidth',2);
xlabel('Time [sec]', 'FontSize',14, 'FontWeight','bold');
ylabel('Displacement [ft]', 'FontSize',14, ...
    'FontWeight','bold');
title('Rocket Displacement vs. Time',...
    'FontSize',16, 'FontWeight','bold');
grid on
set(gca, 'FontSize',10, 'FontWeight','bold');

```

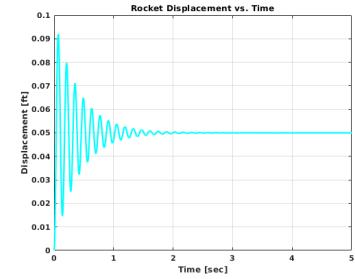


Figure 142: Displacement of the rocket during a simulated powered test.

The equation defining $w'(t)$ is implemented as a local function.

```

38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
function dw = ex2(t,w) ④
% define m(t)
mo = 100; % slugs, initial mass of fuel
m_dot = 1; % slug/s, fuel burn rate
m = @(t) mo - m_dot*t;% slugs, fuel mass

k = 2e5; % lb/ft, spring coefficient
c = 500; % slug/s, damping coefficient
T = 10000; % lb, thrust

% initialize and compute the output
dw = nan(2,1);      ⑤
dw(1) = w(2);
dw(2) = T./m(t) - c*w(2)./m(t) - k*w(1)./m(t);
end

```

④ For this equation, w' is dependent on both t and w . Equations of this type are called *non-autonomous*. Even if the equation is *autonomous*—i.e. not a function of the independent variable—you should still define your local function as $w' = f(t, w)$. This is because MATLAB's built-in tools for solving IVPs expect it to be defined that way.

⑤ The semantics of our main solver routine above expect that the output will be in the form of a column vector.

Lecture 25 - Solving IVPs with Runge-Kutta Methods

Objectives

The objectives of this lecture are to:

- Qualitatively motivate and describe the derivation of Runge-Kutta methods.
- Do some example problems.

Runge-Kutta Methods

Runge-Kutta (RK) methods are a family of single-step numerical methods for solving first order initial value problems.

$$\begin{aligned}y' &= f(t, y) \\y(0) &= y_0\end{aligned}$$

The basic idea stems from that of Euler's methods where we approximated y_{n+1} based on y_n , the slope, $y' = f(t, y)$, and the step size, h .

$$y_{n+1} = y_n + f(t, y)h$$

This can be expressed more exactly in terms of integrals:

$$y_{n+1} = y_n + \int y' dt = y_n + \int_{t_n}^{t_{n+1}} f(t) dt$$

In your earlier classes in ordinary differential equations, this method of integrating to "un-do" the derivative is a standard solution method for separable initial value problems:

$$\frac{dy}{dt} = y' = \frac{h(t)}{g(y)} \Rightarrow \int g(y) dy = \int h(t) dt$$

For the simple case where $g(y) = 1$ and we integrate over one time step we get:

$$\int_{y_n}^{y_{n+1}} 1 dy \rightarrow y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t) dt$$

Let us generalize a bit further and consider first order IVPs in the form:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t, y) dt \quad (195)$$

and propose that we use *quadrature* instead of exact integration. Now we can re-write Equation 195 as:

$$y_{n+1} = y_n + h \sum_{i=1}^s b_i f(t_n + c_i h, y(t_n + c_i h)) \quad (196)$$

where h is like the scaling term, $b - a/2$, in Gauss quadrature, b_i are the weights, and c_i are the sample points. We constrain $c_i \in [0, 1]$ so that $t_n \leq (t_n + c_i h) \leq t_{n+1}$.

ONE PROBLEM WITH this approach is that we do not know the value of $y(t_n + c_j h)$. In RK methods, we will approximate these points between y_n and y_{n+1} as follows:

$$\begin{aligned} \xi_v &= y_n + h \sum_{i=1}^s a_{v,i} f(t_n + c_i h, \xi_i) \\ y_{n+1} &= y_n + h \sum_{i=1}^s b_i f(t_n + c_i h, \xi_i) \end{aligned}$$

The number of sample points, s , is referred to as the number of *stages* and the elements $a_{v,i}$ are customarily arranged into a square matrix, called the *RK matrix*. As an example, for a 2-stage system, we can write out these equations fully as:

$$\begin{aligned} \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} &= y_n + h \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} f(t_n + c_1 h, \xi_1) \\ f(t_n + c_2 h, \xi_2) \end{bmatrix} \\ y_{n+1} &= y_n + h \begin{bmatrix} b_1 & b_2 \end{bmatrix} \begin{bmatrix} f(t_n + c_1 h, \xi_1) \\ f(t_n + c_2 h, \xi_2) \end{bmatrix} \end{aligned}$$

For an *explicit* RK method, the RK matrix is strictly lower triangular. The values of the RK weights, RK nodes, and entries in the RK matrix are customarily organized as a Butcher Tableau¹ as illustrated in Figure 143.

In this class we will not derive any RK methods from scratch. Still, we can make some general observations about RK methods:

1. The order of convergence for explicit RK methods is equal to the number of stages for $s \leq 4$. RK methods with greater than 4th-order convergence have been derived, but in those cases the number of stages, s , is greater than the order of convergence.

Note: With this notation, $y_n = y(t_n)$ and $y_{n+1} = y(t_{n+1})$.

Note: To prevent confusion with Gauss quadrature, from now on, we will refer to c_i as the *RK nodes* and b_i as the *RK weights*.

Note: Notice that the top equation is, in general, non-linear and must be solved iteratively using one of the methods we learned for non-linear systems of equations. If the RK matrix is strictly lower triangular—i.e. only non-zero below the main diagonal—then the values for ξ_i can be solved without iteration.

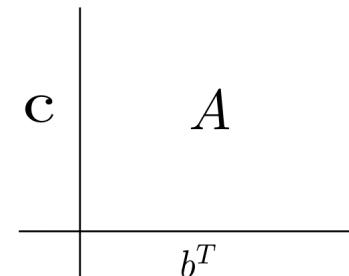


Figure 143: Schematic of a Butcher Tableau.

¹ John Charles Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2016

2. The sum of each row in the RK matrix is equal to the corresponding RK node.
3. the sum of the RK weights is equal to 1.

FROM THIS PERSPECTIVE the modified Euler's method presented in the last lecture is equivalent to a 2-stage RK method. The RK nodes, RK weights, and RK matrix are shown below:

$$c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad b^T = \begin{bmatrix} 1/2 & 1/2 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

and the Butcher Tableau is shown in Table 18.

0	0	0
1	1	0
	1/2	1/2

Table 18: Butcher Tableau for the modified Euler's method.

MATLAB Implementation of RK Methods

In this section I will present and explain three successive MATLAB implementations of an RK method. The goal for this implementation is clarity and generality, not performance. Students who are interested in achieving greater performance are encouraged to find optimizations sometime *after* they fully understand what is shown here.

2^{nd} -Order RK Method, Scalar 1^{st} -Order Equation

We will start with an implementation of the modified Euler's method cast as an RK method for a scalar differential equation of 1^{st} -order. The first portion of the function is shown in the listing below. Here we document the input and output variables; define variables to hold the RK matrix, RK weights, and RK nodes; and allocate a vector for the solution.

```

function y = odeRK2(f,a,b,N,yINI)
% function y = odeRK2(f,a,b,h,yINI)
% y = solution
% f = function handle for y'
% a,b = interval for solution
% N = number of steps between a and b (inclusive)
% yINI = initial value for the solution

A = [0 0;
      1 0]; % RK matrix
B = [0.5 0.5]; % weights
c = [0 1]';% sample points
stages = 2;

x = linspace(a,b,N);
y = nan(1,N);
y(1) = yINI;
h = x(2)-x(1);

```

Now we use the RK method to compute the solution:

```

for t = 1:(N-1)
    Xi = nan(1,stages); ①
    for s = 1:stages
        Xi(s) = y(t); ②
        for i = 1:(s-1)
            Xi(s) = Xi(s) + ...
                h*A(s,i)*f(x(t)+c(i)*h,Xi(i));
        end
    end

    y(t+1) = y(t);
    for i = 1:stages
        y(t+1) = y(t+1) + ...
            h*B(i)*f(x(t)+c(i)*h,Xi(i)); ③
    end
end
end

```

① A new array of slopes, one element for each stage, will be needed at each time step.

② This nested for loop calculates:

$$\xi_s = y_n + h \sum_{i=1}^{s-1} a_{s,i} f(t_n + c_i h, \xi_i)$$

for each value of s . The upper limit of the summation index, $s - 1$, is due to the fact that this is an explicit method and the RK matrix is strictly lower triangular.

③ This loop calculates:

$$y_{n+1} = y_n + h \sum_{i=1}^s b_i f(t_n + c_i h, \xi_i)$$

WE WILL USE this function to solve an example problem.

Example #1: Solve the following initial value problem:

$$y' = x^2/y, \quad y(0) = 2$$

where the exact solution is:

$$y(x) = \left(\frac{2}{3}x^3 + 4 \right)^{1/2}$$

In the listing below we set up the problem and invoke our 2nd-order RK solver:

```

clear
clc
close 'all'

%% Define the problem to be solved.
%%
% $$y'^{\prime} = x^2/y, \quad y(0) = 2$$
%
f = @(x,y) (x.^2)./y; % y' = f(x,y)
yINI = 2; % y(0) = 2
y_exact = @(x) sqrt((2/3)*x.^3 + 4);
xMin = 0; xMax = 2.0;
x_gold = linspace(xMin,xMax,1000);

%% Invoke the solver
N = 30;
x = linspace(xMin,xMax,N);
y_RK2 = odeRK2(f,xMin,xMax,N,yINI);

```

Note: It is worth emphasizing yet again the importance of testing a new method and/or a new implementation on a problem where you already know the solution.

Once the solution is complete we visualize the results.

```
%>> %% Plot the results
figure(1)
plot(x,y_RK2,'-b',...
x_gold,y_exact(x_gold),'-r',...
'linewidth',3);
title("Solution with RK2 Method",'fontsize',14,...
'fontweight','bold');
xlabel('X','fontsize',12,'fontweight','bold');
legend('RK2 Solution','Exact Solution','location','best');
grid on;
set(gca,'fontsize',10,'fontweight','bold');
```

A comparison between the numeric and exact solutions is shown in Figure 144.

WHILE IT SEEMS evident that the numeric solution is correct, plots of this sort are not adequate for verifying the correct performance of the numeric solver. The numeric method should exhibit 2nd-order convergence and we want to, somehow, verify this behavior. The next MATLAB listing re-computes the numeric solution with successively smaller step-sizes. For each solution, the relative error is computed in the 2-norm. If the relative error is, indeed, proportional to h^2 , then we can have more confidence that the algorithm is implemented correctly. Figure 145 compares the relative error trend with what we would expect for a 2nd-order convergent method. The MATLAB code to carry out this task is shown in the next listing.

```
%>> %% Convergence Test
N = 3:18;
t = length(N);
err_array = nan(1,t);
h_array = nan(1,t);
for s = 1:t
    Nx = 2^(N(s));
    x = linspace(xMin, xMax, Nx);
    h = x(2)-x(1);
    h_array(s) = h;
    y_ns = odeRK2(f,xMin,xMax,Nx,yINI);
    err_array(s) = norm(y_exact(x)-y_ns,2) ./ ... ❶
        norm(y_exact(x),2);
end

err_gage = h_array.^2; ❷

figure(2)
loglog(h_array,err_array,'-b',...
h_array,err_gage,'-r','linewidth',2);
title("Convergence Explicit RK2 Method",...
'fontsize',14,'fontweight','bold');
xlabel('h','fontsize',12,'fontweight','bold');
ylabel('Relative Error','fontsize',12,'fontweight','bold');
grid on;
legend('Relative Error','h^2','location','best');
```

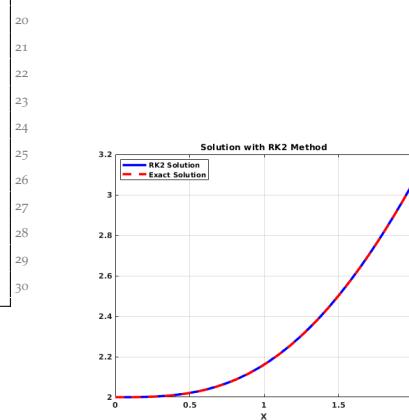


Figure 144: Comparison between numeric and exact solution for Example #1.

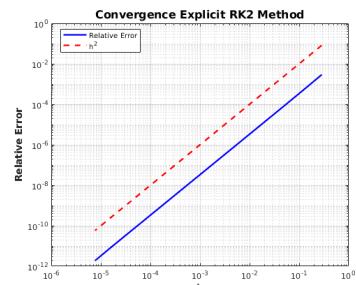


Figure 145: Convergence behavior of 2nd-order RK method for Example #1.

❶ Compute 2-norm of the error and normalize by the 2-norm of the exact solution to get the *relative* error in the 2-norm.

❷ Create a vector proportional to h^2 to compare with the relative error.

Generalized Explicit RK Method for 1st-Order Equation

Instead of hard-coding the RK weights, sample points, and RK matrix into a solver, we can instead create a generalized solver that can use any explicit RK method where the aforementioned parameters are passed in via the Butcher tableau. Consider the function shown in the MATLAB listing below:

```

function y = odeExplicitRK(f,a,b,N,yINI,BT)
% function y = odeExplicitRK(f,a,b,h,yINI,BT)
% y = solution
% f = function handle for y'
% a,b = interval for solution
% N = number of steps between a and b (inclusive)
% yINI = initial value for the solution
% BT = Butcher Tableau

% Un-pack Butcher tableau parameters ❸
s = length(BT)-1;
c = BT(1:s,1);
B = BT(s+1,2:end);
A = BT(1:s,2:end);
stages = s;

%% Carry out explicit RK method as specified
x = linspace(a,b,N);
y = nan(1,N);
y(1) = yINI;
h = x(2)-x(1);

for t = 1:(N-1)
    Xi = nan(1,stages);
    for s = 1:stages
        Xi(s) = y(t);
        for i = 1:(s-1)
            Xi(s) = Xi(s) + ...
                h*A(s,i)*f(x(t)+c(i)*h,Xi(i));
        end
    end

    y(t+1) = y(t);
    for i = 1:stages
        y(t+1) = y(t+1) + h*B(i)*f(x(t)+c(i)*h,Xi(i));
    end
end
end

```

❸ The Butcher tableau is stored in a $(s + 1) \times (s + 1)$ matrix, where s is the number of stages. The layout is as shown in Figure 143.

We can use this generalized solver to repeat the calculation for Example #1 as shown in the MATLAB listing below.

```

%% Use Generalized Explicit RK method based on Butcher Tableau
% Construct Butcher tableau for 2nd-order RK method
s = 2;
BT = zeros(s+1,s+1);
C = [0; 1; 0];
B = [0 1/2 1/2];
A = [0 0;
     1 0];
BT(:,1) = C;
BT(end,:)= B;

```

```

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

BT(1:s,2:end) = A;

%% Set parameters and invoke the solver
N = 30;
yINI = 2;
x = linspace(xMin,xMax,N);
y_RK2 = odeExplicitRK(f,xMin,xMax,N,yINI,BT);

%% Plot the results
figure(3)
plot(x,y_RK2,'-b',...
    x_gold,y_exact(x_gold),'-.r',...
    'linewidth',3);
title("Solution with RK2 Method",'fontsize',14,...
    'fontweight','bold');
xlabel('X','fontsize',12,'fontweight','bold');
legend('RK2 Solution','Exact Solution','location','best');
grid on;
set(gca,'fontsize',10,'fontweight','bold');

```

With this function, *any* explicit RK method can be carried out; all that the user needs to do is to provide the corresponding Butcher tableau.

Explicit RK Method for Systems of 1st-Order Equations

It should be clear to the reader that what we have done so far is not quite enough. The method is only written to deal with first-order equations while, in general, our solver should be able to handle *systems* of equations so higher-order IVPs can be solved.

Example #2: Solve the following initial value problem using an explicit Runge-Kutta method.

$$4y'' + 4y' + 17y = 0, \quad y(0) = -1, \quad y'(0) = 2$$

The exact solution is:

$$y(x) = e^{-x/2} \left[-\cos(2x) + \frac{3}{4} \sin(2x) \right]$$

To solve this equation we need to re-formulate the IVP as a first order system of equations:

$$\begin{aligned} y'' &= -y' - \frac{17}{4}y \\ w &= \begin{bmatrix} y \\ y' \end{bmatrix}, \quad dw = \begin{bmatrix} w(2) \\ -w(2) - \frac{17}{4}w(1) \end{bmatrix} \end{aligned}$$

To HANDLE THIS situation, we will re-write the generalized Runge-Kutta ODE solver so that it can handle a system of equations.

```

function y = odesExplicitRK(f,a,b,N,yINI,BT)
% function y = odeExplicitRK(f,a,b,h,yINI,BT)
% y = solution (vector)
% f = function handle for y'
% a,b = interval for solution
% N = number of steps between a and b (inclusive)
% yINI = initial value for the solution
% BT = Butcher Tableau

% Unpack Butcher tableau parameters
s = length(BT)-1;
c = BT(1:s,1);
B = BT(s+1,2:end);
A = BT(1:s,2:end);
stages = s;

%% Carry out explicit RK method on the system of equations
x = linspace(a,b,N);
sys_size = length(yINI);
y = nan(sys_size,N);
y(:,1) = yINI;
h = x(2)-x(1);
for t = 1:(N-1)
    Xi = nan(sys_size,stages);

    for s = 1:stages
        Xi(:,s) = y(:,t);
        for i = 1:(s-1)
            Xi(:,s) = Xi(:,s) + ...
                h*A(s,i)*f(x(t)+c(i)*h,Xi(:,i));
        end
    end

    y(:,t+1) = y(:,t);
    for i = 1:stages
        y(:,t+1) = y(:,t+1) + ...
            h*B(i)*f(x(t)+c(i)*h,Xi(:,i));
    end
end
end

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41

Note: Readers are strongly encouraged to compare this implementation with the explicit RK solver for 1st-order equations. The dependent variable, y , is now a two-dimensional array; one row for each equation in the system; the columns correspond to time-steps.

We can use a local function to implement the 1st-order system of ODEs:

```

function dw = ex2(~,w) ④
% generally expect 2 arguments for solvers (IV first)
dw = nan(2,1);
dw(1) = w(2);
dw(2) = (-w(2) + (17/4)*w(1));
end

```

1
2
3
4
5
6

④ The independent variable is not used in this equation. Still, for MATLAB built-in solvers, 2 arguments will be expected. If we include the 1st argument in the list but then do not use it in the function, MATLAB's Code Analyzer will issue a warning. We can avoid this warning while still having 2 arguments by using a tilde character in place of the first argument.

We use the generalized, explicit, RK solver along with the function corresponding to the differential equation to solve the initial value problem:

```

%% Generalize for System of ODEs
N = 30;
f = @(t,y) ex2(t,y);
yINI = [-1 2]; % initial values

```

1
2
3
4

```

x = linspace(xMin,xMax,N);
ys_RK2 = odesExplicitRK(f,xMin,xMax,N,yINI,BT);

y_exact = @(x) exp(-x./2).*(-cos(2*x)+0.75*sin(2*x));

%% Plot the result
figure(4)
plot(x,ys_RK2(1,:),'-b',...
    x_gold,y_exact(x_gold),'-r',...
    'linewidth',3);
title("Solution with RK2 Method",'fontsize',14,...
    'fontweight','bold');
xlabel('X','fontsize',12,'fontweight','bold');
legend('RK2 Solution','Exact Solution','location','best');
grid on;
set(gca,'fontsize',10,'fontweight','bold');

```

The solution of Example #2 is shown in Figure 146 and the convergence behavior of the 2nd-order RK solver for systems of equations is shown in Figure 147.

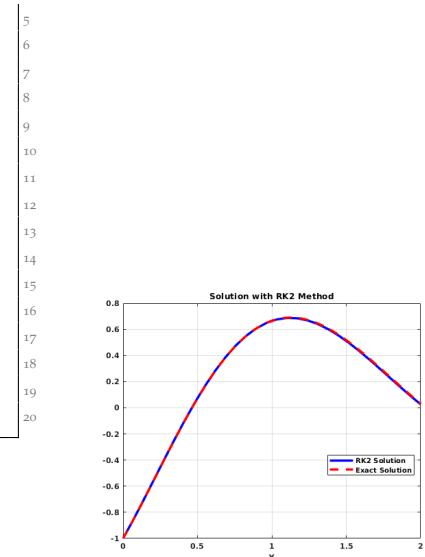


Figure 146: Solution of Example #2.

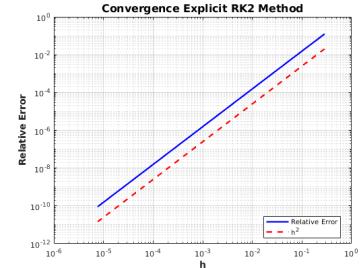


Figure 147: Convergence behavior of 2nd-order RK solver for systems of equations.

Assignment #8

1. Use MATLAB's built-in function `integral3` to evaluate the integral below.

$$\int_{-1}^1 \int_{-1}^1 \int_{-1}^1 (x^2 + y^2 + z^2) \, dx \, dy \, dz$$

Write a function that uses a generalization of the mean value algorithm for Monte Carlo integration, where $\int_V f(V) \, dV = \frac{\text{volume of integral bounds}}{N} \sum_{i=1}^N f(x_i, y_i, z_i)$; N is the number of sample points and, for this integral, the volume of the integral bounds is 8. Create a convergence plot to show that the relative error in your Monte Carlo solution—relative to the solution from `integral3`—is proportional to $1/\sqrt{N}$.

2. In imaging and treatment of breast cancers, an ellipsoidal shape as shown in Figure 148 may be used to represent certain tumors so that changes in their surface areas may be quantified and monitored during treatment. The surface area of an ellipsoid is given by:

$$S = 8ab \int_0^{\pi/2} \int_0^{\pi/2} \sin \theta \sqrt{1 - p \sin^2 \theta} \, d\theta \, d\phi$$

where $p = \delta \sin^2 \phi + \epsilon \cos^2 \phi$, $\delta = 1 - \frac{c^2}{a^2}$, $\epsilon = 1 - \frac{c^2}{b^2}$, and $2a$, $2b$, and $2c$ are the major dimensions of the

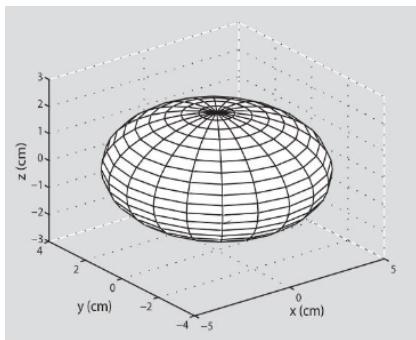


Figure 148: Elliptical shape for breast cancer.

ellipsoid along the x , y , and z axes, respectively. For $2a = 9.5$ cm, $2b = 8$ cm, and $2c = 4.2$ cm, calculate the surface area of this ellipsoid tumor using MATLAB built-in functions.

3. Write a user-defined MATLAB function to solve a first-order system of initial value ODEs. The signature should be $[t,y] = \text{odesEULER}(F,a,b,N,yINI)$ where F is a handle to a function that defines the system of ODEs, a and b are the starting and end points in t , and $yINI$ is a vector of initial conditions. Use this function to solve the following IVP:

$$\frac{d^2y}{dx^2} + 2\frac{dy}{dx} + 2y = 0, \quad 0 < x < 1.5, \quad y(0) = -1, \quad y'(0) = 0.2$$

The exact solution is: $y(x) = e^{-x}(-\cos(x) - 4\sin(x)/5)$. Solve the system once for $N = 100$ and again for $N = 1000$. Check that your solution exhibits convergence of order 1 using relative error in the 2-norm.

4. Consider the following first-order IVP:

$$\frac{dy}{dt} = y + t^3, \quad 0 < t < 1.5, \quad y(0) = 1$$

Solve by hand (pencil-paper-calculator) with the classical fourth-order Runge-Kutta method using $h = 0.5$. The analytical solution of the ODE is $y(t) = 7e^t - t^3 - 3t^2 - 6t - 6$. Calculate the error between the true solution and the analytic solution at $t=0.5$, 1.0 , and 1.5 seconds.

5. Consider the cylindrical water tank that is shown in Figure 149. The tank is being filled at the top, and water flows out of the tank through a pipe that is connected at the bottom. The rate of change of the height, h , of the water is given by the equation below:

$$\rho A_{\text{tank}} \frac{dh}{dt} = K_1 + K_2 \sin(5Ct) \cos(Ct) - \rho A_{\text{pipe}} \sqrt{2gh}$$

For the given tank, $A_{\text{tank}} = 3.13 \text{ m}^2$, $A_{\text{pipe}} = 0.06 \text{ m}^2$, $C = \frac{\pi}{12}$, $K_1 = 300 \text{ kg/s}$, and, $K_2 = 1000 \text{ kg/s}$.

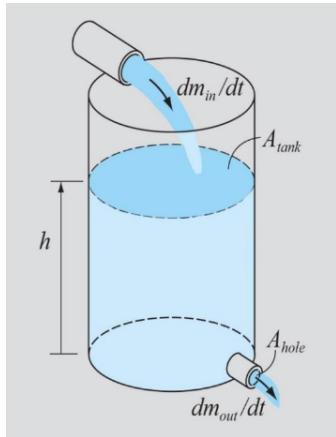


Figure 149: Water supply tank.

Also, $\rho = 1000 \text{ kg/m}^3$, and $g = 9.81 \text{ m/s}^2$. Determine and plot the height of the water as a function of time for $0 \leq t \leq 150$ seconds if, at $t = 0$, $h = 3\text{m}$.

- (a) Use the odesEuler function that you created for problem 3.
- (b) Create a user defined MATLAB function with signature $[t, y] = \text{odesCRK4}(F, a, b, N, yINI)$ that implements the classical fourth-order Runge-Kutta method. Use this function to solve the problem.

For both methods, use $N=1000$. In addition to the plot, output the value of $h(150)$ for each method.

Take some time to consider both the given initial value problem, your solution, and its implication on the design of this tank. What happens when you change the parameters A_{tank} , A_{pipe} , K_1 , K_2 , and C ? Do those changes make sense? Assuming there are limits on acceptable values of h , how does that impact your choice of design parameters such as A_{tank} and A_{pipe} relative to K_1 , K_2 , and C ? Write a short discussion (one or two paragraphs, with supporting plots) to present your findings.

6. A small rocket having an initial weight of 3000 lb (including 2400 lb of fuel), and initially at rest, is launched vertically upward. The rocket burns fuel at a constant rate of 80 lb/s, which provides a constant thrust, T , of 8000 lb. The instantaneous weight of the rocket is $w(t) = 3000 - 80t$ lb. The drag force, D , experienced by the rocket is given by: $D(t) = 0.005g \left(\frac{dy}{dt}\right)^2$ lb, where y is the distance in ft, and $g = 32.2 \text{ ft/s}^2$. Using Newton's law, the equation of motion for the rocket is given by:

$$\frac{w}{g} \frac{d^2y}{dt^2} = T - w - D$$

Determine and plot the position, velocity, and acceleration of the rocket (three sets of axes in one figure; use MATLAB's subplot function) as a function of time from $t = 0$, when the rocket starts moving upward from rest, until $t = 3$ seconds. Also, output the distance and velocity at $t = 3$ seconds.

Use the user-defined function you created for problem #5 to solve this equation using the classical fourth-order Runge-Kutta method with $N=1000$. You may need to adapt the function slightly if you did not initially implement the function to solve systems of equations.

Take some time to consider the given initial value problem and its solution:

- (a) Explain the governing equation and each of its terms.
- (b) How is the solution different if you do not consider the weight change due to burning of fuel?
- (c) What happens if the drag force is eliminated?
- (d) What if the leading term in the drag force is cut in half? How does this change the shape of the acceleration curve and does this make sense?

Write a short discussion (one or two paragraphs, with supporting plots) to present your findings.

Lecture 26 - High Order & Implicit Runge-Kutta Methods

Objectives

The objectives of this lecture are to:

- Introduce some high order RK methods useful for IVPs.
- Demonstrate how to use implicit RK methods.
- Demonstrate the convergence behavior of various RK methods for a variety of problems.

High Order RK Methods

So far in this class we have only discussed Runge-Kutta Methods that exhibit 1st- or 2nd-order convergence. Given how we employed integration methods with much higher order convergence properties, it stands to reason that we can do better.

In general, to obtain higher order convergence behavior, we need more stages. In allegiance to our commitment to avoid *deriving* RK schemes, in this section we will simply list the higher order methods and list their Butcher tableau.

Classical 3rd-Order, 3-stage RK

The Butcher tableau for this method is shown in Table 19.

Classical 4th-Order, 4-stage RK

The Butcher tableau for this method is shown in Table 20.

IMPLEMENTATION OF THESE RK methods within the framework described in Lecture 25 is straight-forward; simply encode the Butcher tableau as a matrix and pass to odesExplicitRK.

0	0	0	0
1/2	1/2	0	0
1	-1	2	0
	1/6	2/3	1/6

Table 19: Butcher tableau for classical 3rd-order, 3-stage explicit RK method.

0	0	0	0
1/2	1/2	0	0
1/2	0	1/2	0
1	0	0	1

Table 20: Butcher tableau for classical 4th-order, 4-stage explicit RK method.

Implicit Runge-Kutta Methods

If we allow the RK matrix to be full, Runge-Kutta methods with higher order convergence and better stability properties can be derived.¹ Butcher tableaux for a 2-stage, 3rd-order IRK is shown in Table 21 and a 2-stage, 4th-order IRK is shown in Table 22.

THE REAL ISSUE that must be dealt with for implicit RK methods is the need to solve a system of non-linear equations. For a first-order, scalar IVP, we might write the equations as shown below:

$$\begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} = y_n + h \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} f(t_n + c_1 h, \xi_1) \\ f(t_n + c_2 h, \xi_2) \end{bmatrix}$$

$$y_{n+1} = y_n + h \begin{bmatrix} b_1 & b_2 \end{bmatrix} \begin{bmatrix} f(t_n + c_1 h, \xi_1) \\ f(t_n + c_2 h, \xi_2) \end{bmatrix}$$

To find values for ξ_i , we need to solve the non-linear system of equations:

$$\begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = y_n + h \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} f(t_n + c_1 h, \xi_1) \\ f(t_n + c_2 h, \xi_2) \end{bmatrix} - \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} \quad (197)$$

WE STILL NEED the ability to solve higher-order IVPs. This means that our non-linear system of equations must accommodate separate values of ξ_i for multiple dependent values. To accomplish this, we will arrange our values of ξ as shown below:

$$\begin{bmatrix} \xi_{1,1} & \cdots & \xi_{1,s} \\ \vdots & \ddots & \vdots \\ \xi_{n,1} & \cdots & \xi_{n,s} \end{bmatrix}$$

where now we place all of the sample points of all stages of a given dependent variable along each row; with n rows, one for each dependent variable in a system of n equations.

Somehow, we need a function that does the equivalent of Equation 197 for the array of ξ values for each equation and for each stage. Such a function is shown in the MATLAB listing below:

```
function R = IRK_Ksol(F, t, y, h, K_g, c, A)
% F - the ODE to be solved
% t - the independent variable
```

¹ Convergence up to order 2s where s is the number of stages.

0	1/4	-1/4
2/3	1/4	5/12
0	1/4	3/4

Table 21: Butcher tableau for a 2-stage, 3rd-order implicit RK method.

1/2 - √3/6	1/4	1/4 - √3/6
1/2 + √3/6	1/4 + √3/6	1/4
0	1/2	1/2

Table 22: Butcher tableau for a 2-stage, 4th-order implicit RK method.

```
% y - the dependent variable for the current time step
% K_g - matrix of guessed values for xi.
% c - the RK nodes
% A - the RK matrix

[s,~] = size(A); % number of stages for the IRK scheme
d = length(y); % number of dependent variables

R = nan(d,s);

for stage = 1:s
    R(:,stage) = y;
    for i = 1:s
        R(:,stage) = R(:,stage) + ...
            h*(A(stage,i)*F(t+c(i)*h,K_g(:,i))); ①
    end
    R(:,stage) = R(:,stage) - K_g(:,stage);
end
end
```

4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

① These lines collectively compute the residual as shown in Equation 197 except that now the residual is a $n \times s$ array.

We will use the secant method to solve this nonlinear system of equations.

```
function Xs = SecantRootSys(Fun,t,Xa,Xb,imax,Err)
% function Xs = SecantRootSys(Fun,Xa,imax,Err)
% solves a system of non-linear equations using
% the Secant Method.
%
% Inputs:
% Fun - input function (must take 2 arguments: F(t,Xa)
% where t is a scalar % representing the current value
% of the independent variable; and Xa is a vector
% containing the initial value for all dependent
% variables.
%
% t - scalar - initial value of independent variable
% Xa - vector - initial values
% Xb - vector - a second set of values
% imax - maximum number of iterations
% Err - error tolerance (relative 1-norm)

for i = 1:imax
    FunXb = Fun(t,Xb);
    denom = Fun(t,Xa)-FunXb;

    denom(denom==0) = 1; ②

    Xi = Xb - FunXb.*(Xa - Xb)./(denom);

    % prevent further problems with zeros
    Xi(Xi==0) = rand; ③

    if (norm(Xi - Xb,inf)/norm(Xb,inf)) < Err
        Xs = Xi;
        break;
    end

    Xa = Xb;
    Xb = Xi;
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

Note: Recall that the secant method is like Newton's method except derivatives are replaced by the slope of the secant line. For a *system* of equations, in general, one needs to create a Jacobian and update each independent variable by solving a linear system of equations as described in Lecture 6. Even though this is really a system of equations, the method employed here is analogous to what one would do with an array of individual equations. The implementation is ugly but nonetheless works and achieves the expected convergence rate for all IRK methods tested.

② Here we need to address the problem: what do we do if any element of *denom* is equal to 0? Our answer: just change that element of the matrix to 1. This sounds crazy but one might hope that such perturbations would not always be necessary and would effectively be fixed on the next iteration. Convergence will be delayed but overall execution would become more reliable. At least that is the hope.

③ We take a slightly different approach here and set any offending values to a random number. Suffice it to say, these measures were arrived at on an ad hoc experimental basis.

```

end
if i == imax
    error('Error! Solution was not obtained at t=%g within %i
iterations.',t,imax);
end
end

```

39
40
41
42
43
44

Once we have found the sample points that satisfy all of the nonlinear equations, we can evaluate the ODE at the RK nodes and sample points and apply the RK weights to solve for the dependent variables for the next time step. The entire process is orchestrated by the MATLAB function shown below.

```

function y = odesImplicitRK(ODE,a,b,N,yINI,BT)
% y = solution (vector)
% ODE = function handle for y'
% a,b = begining and end of the interval for solution
% N = number of steps between a and b
% yINI = initial value for the solution
% BT = Butcher Tableau

% get Butcher Tableau Parameters
s = length(BT) - 1;
c = BT(1:s,1);
b_t = BT(s+1,2:end);
A = BT(1:s,2:end);
[sys_size,~] = size(yINI);
%h = (b-a)/N;
x = linspace(a,b,N);
h = x(2) - x(1);
y = zeros(sys_size,N);
y(:,1) = yINI;

% SecantRootSys arguments
imax = 10000;
Err = 1e-14;
Ka = ones(sys_size,s)*.01; %<-- maybe zero is a bad choice ...
Kb = ones(sys_size,s).*ODE(x(1),y(:,1));
for t = 1:(N-1)
    y(:,t+1) = y(:,t);
    % need to solve for the Ks
    F = @(iv,k) IRK_Ksol(ODE,iv,y(:,t),h,k,c,A);
    K = SecantRootSys(F,x(t),Ka,Kb,imax,Err);

    for i = 1:s
        y(:,t+1) = y(:,t+1) + h*b_t(i)*ODE(x(t)+c(s)*h,K(:,i));
    end

    % update these for the secant solver for next time step
    Ka = Kb;
    Kb = K;
end
end

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41

Lecture 27 - Embedded RK and MATLAB Built-in RK Methods

Objectives

The objectives of this lecture are to:

- Describe and demonstrate embedded Runge-Kutta methods as they relate to adaptive step-sizing
- Introduce MATLAB Built-in RK Methods
- Illustrate their use through example problems.

Embedded Runge-Kutta Methods

Besides developing methods to achieve higher order accuracy, there is also an incentive to minimize the required number of time steps while also achieving a specified absolute or relative error. In the preceding examples, the user has been called upon to input the desired time step size and interval over which the calculation is to be made and it was left for the user to decide if the time step size is appropriate. But if we consider the problem from a somewhat higher level, we would not expect the user to be focused on the time step size but rather whether or not the solution is accurate enough. The user should input the expected relative or absolute error tolerance instead of details about how many time steps to make or how big those steps should be.

What we need are:

1. A way to estimate the error in a solution; and
2. the means of making this estimate should not require a lot more work.

EMBEDDED RUNGE-KUTTA METHODS provide for those needs. Consider the Bogacki-Shampine method¹ that is used in MATLAB's

¹ Przemyslaw Bogacki and Lawrence F Shampine. A 3 (2) pair of runge-kutta formulas. *Applied Mathematics Letters*, 2 (4):321–325, 1989

built-in function `ode23`. The Butcher tableau is shown in Table 23.

In this Butcher tableau, there are two sets of weights: one for a 2nd-order convergent RK method and the other is for a 3rd-order convergent scheme. Note that the numbers to the left of the weights in the Butcher tableau are not standard and added to indicate the order of convergence for the corresponding set of weights.

A Runge-Kutta scheme of this type can form the basis for a method that includes step-size adaptation. Suppose we specify a relative error tolerance and compute y_{n+1} given y_n .

- Start with a default initial step size. This might be input by the user or may be set by the algorithm, for example, as a fixed fraction of the interval length.
- Compute the numeric solution for the $(i + 1)$ -th time-step using both lower-order and higher-order weights (w_{i+1} and z_{i+1} , respectively)
- Obtain a measure of the relative error from the two solutions.

$$e_{i+1} \approx \frac{\|w_{i+1} - z_{i+1}\|}{\|z_{i+1}\|}$$

- If the relative error measure fails to meet the specified error tolerance, we reduce the step size—for instance, $h_{new} = h/2$ —and re-compute.
- If the relative error tolerance is satisfied, we specify a larger step size for the next time step. One equation for doing this is shown in Equation 198 where p is the order of the solver, h is the time step size, e_{i+1} is the error measure, and SF is a chosen *safety factor* that, by setting to a value between 0 and 1, prevents changing the time step size too aggressively.²

$$h_{new} = (SF) \left(\frac{TOL}{e_{i+1}} \right)^{\frac{1}{p+1}} h_{old} \quad (198)$$

- Ensure the final time-step is sized so that the algorithm terminates at the end of the specified interval.

One possible implementation of this method is shown in the set of listings below.

```
function [t,y] = embeddedRK(F,tspan,yo,RTOL,EBT)
t_sz = 5000; % initial size for output arrays
tsMax = 100000;
sys_size = length(yo);
t = nan(1,t_sz); %❷
y = nan(sys_size,t_sz);
```

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{3}{4}$	0	$\frac{3}{4}$	0	0
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0
(2)	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$
(3)	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0

Table 23: Butcher tableau for the Bogacki-Shampine embedded Runge-Kutta method.

Note: Since the system is, in general, n^{th} -order, w and z are generally vectors of length n . Choose a suitable norm in which to obtain this relative error measure. In order to guard against small values of $\|z_{i+1}\|$, use a tool like $\min(z,\theta)$, where θ is a small non-zero value.

² Timothy Sauer. *Numerical analysis*. Addison-Wesley Publishing Company, 2011

❶ The last argument, EBT, is what we will call the *extended* Butcher tableau, so-called because it is extended to contain two sets of weights.

❷ For this method we do not know in advance how many time steps will be taken but we also want to pre-allocate arrays for our variables. Consequently we will make a conservative guess at the number of time steps necessary. At the end of the calculation we will trim away unused portions of the array.

Now we unpack the Butcher tableau and initialize our variables.

```
% set initial values
t(1) = tspan(1);
y(:,1) = yo;

[m,~] = size(EBT);
s = m-2;
C = EBT(1:s,1); ❸
A = EBT(1:(end-2),2:end);
BW = EBT(s+1,2:end);
BZ = EBT(s+2,2:end);
p = EBT((end-1),1);
```

7
8
9
10
11
12
13
14
15
16
17

❸ Here we unpack the Butcher tableau with the only difference being that we have two sets of weights to use in the low order w and high order z approximation.

The rest of the implementation follows; the local function getWZ() will be shown in the next listing.

```
SF = 0.9; % "safety factor"
theta = 1e-14; ❹
tStart = tspan(1);
tEnd = tspan(2);
h = (tEnd-tStart)/10; % initial step size
h_new = h;
stopFlag = 0; % flag to end on last time step

for ts = 1:tsMax
    cT = t(ts); % current time

    % find acceptable time step size ❺
    int_it_count = 0; % limit iterations in error control.
    while int_it_count < 10
        int_it_count = int_it_count + 1;
        h = h_new; % update with new step-size

        % if cT + h > tEnd, reduce h so we stop right on time
        if cT+h > tEnd
            h = tEnd-cT;
            stopFlag = 1; % stop after this time step
        end

        y(:,ts+1) = y(:,ts);
        K = getSlopeEst(F,t(ts),y(:,ts),h,C,A); ❻
        [w,z] = getWZ(y(:,ts),h,K,BW,BZ); ❼
        err_ts = abs(w-z); % error vector (length = # dofs)
        rel_err_ts = err_ts./max(abs(z),theta);
        max_rel_err_ts = norm(rel_err_ts,inf);
        if max_rel_err_ts < RTOL ❽
            h_new = SF*(RTOL/max_rel_err_ts)^(1/(p+1))*h;
            y(:,ts+1) = z;
            t(ts+1) = cT + h; % update current time
            break; % exit the while loop
        else
            h_new = h/2; ❾
            if stopFlag == 1
                stopFlag = 0;
            end
        end % if max_rel_err_ts ...

        if int_it_count > 10 ❿
            error('Error-control is broken!');
        end
    end
```

18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

❹ This is the small factor to ensure the normalization used in calculating relative error is not too close to zero.

❺ With each time step we need to find the right step-length. We will use an iterative process starting with the time step size from the previous time step.

❻ Move this process out to a different local function.

❼ Use the two sets of weights to get the low-order (w) and high-order (z) approximation.

❽ If the max relative error is within the relative error tolerance, accept the solution but also update the time step size for the next time around.

❾ If the relative error tolerance is not met, cut the time step size in half and try again.

❿ If it takes more than a couple of iterations, something has went wrong; stop the program and try to figure it out.

```

end % while

if stopFlag == 1 % end of time stepping. exit loop.
    break;
end
end %ts
% trim the output variables.
y = y(:,1:(ts+1)); ①
t = t(1:(ts+1));
end % function

```

❶ Recall, the variable arrays were oversized to accommodate an unknown number of time steps. Trim-off any unused time steps here.

The local functions used above are shown here for completeness.

```

function K = getSlopeEst(F,t,y,h,C,A)
sys_size=length(y);% gen # of dep vars
[s,~] = size(A); % get number of stages
K = zeros(sys_size,s);
K(:,1) = F(t,y);
for i = 2:s
    y_it = y(:);
    x_it = t + C(i)*h;
    for j = 1:(i-1)
        y_it = y_it + h*A(i,j)*K(:,j);
    end %j
    K(:,i) = F(x_it,y_it);
end %i
end

function [w,z] = getWZ(y,h,K,BW,BZ)
s = length(BW);
w = y; z = y;
for i = 1:s
    w = w + h*BW(i)*K(:,i);
    z = z + h*BZ(i)*K(:,i);
end
end

```

We use this embedded Runge-Kutta method to solve example #2 from Lecture 25:

$$4y'' + 4y' + 17y = 0, \quad y(0) = -1, \quad y'(0) = 2$$

with exact solution:

$$y(x) = e^{-x/2} \left[-\cos(2x) + \frac{3}{4} \sin(2x) \right]$$

A plot of the solution is shown in Figure 150. Note the shorter time steps in the vicinity of rapid solution variation or inflection points.

MATLAB Built-in Runge-Kutta Methods

MATLAB has a number of built-in methods for solving initial value problems. Most of the built-in methods are based on embedded Runge-Kutta methods. A partial list of recommended methods is given in Table 24.

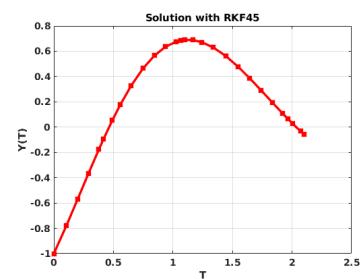


Figure 150: Solution of example problem with an embedded Runge-Kutta method.

Solver Name	Description
ode45	For non-stiff problems, best to apply as a first try for most problems. Single-step method based on fourth and fifth order explicit RK methods
ode78	For non-stiff problems where high accuracy is required. Based on 7th and 8th order explicit RK methods
ode89	For non-stiff problems where high accuracy is required over very long time intervals. Based on 8th and 9th order explicit RK methods
ode23	Consider this function if the problem is moderately stiff and/or if you have crude error tolerances. The algorithm is based on 2nd and 3rd order explicit RK methods
ode15s	Try this method if ode45 fails or if you think your equation is stiff. Based on numeric differentiation formulas of order 1 through 5.

Table 24: A partial list of built-in IVP solvers for MATLAB.

NOTE THAT IT is not necessarily better to simply use the method with the highest-possible order of convergence. If we define the “best” method as that method which reliably converges to an answer of acceptable accuracy the fastest, then higher-order methods are not always the best. A more detailed performance analysis would take into consideration the number of function evaluations and computation time needed to obtain a solution of the desired accuracy. Problems where the behavior of the solution varies significantly over a range of time scales³ can result in extra work adapting the time step-size and overall poorer performance.

To USE, for example, ode45, the syntax is as follows:

```
[tSol,ySol]=ode45(odeFun,tSpan,Yo)
```

where `odeFun`⁴ is a handle to the function implementing the governing equation, `tSpan` is either a vector with two elements indicating the range of the independent variable or a vector specifying time-steps at which the solution is desired. The argument `Yo` is a vector with initial conditions. The return variable `tSol` is a $1 \times N$ vector where N is the number of time steps ultimately taken by `ode45`; `ySol` is a $N \times s$ array where s is the number of dependent variables in the system of equations.

EACH OF THE methods listed are of variable time step-size; unless

³ This is the best intuitive description I have of what it means for a problem to be “stiff.”

⁴ Note that this function must take two inputs; the first is the independent variable, and the second is a vector of dependent variables. Even if the governing equation is not a function of the independent variable, the function you send to `ode45` and other built-in MATLAB initial value problem solvers, must accept it as its first argument.

the user specifies the time steps through the `tSpan` variable, the built-in methods will adaptively adjust the time step size to meet error tolerances. Optional parameters—like absolute and relative error tolerances—can be set by providing an options structure by using the `odeset` function as shown below:

```
optStruct = odeset('RelTol',1e-5,'AbsTol',1e-8,'Stats','on')
```

where some possible name/value pairs are provided for common options. Readers are encouraged to consult the MATLAB documentation on available options and default values for those optional parameters for each built-in function. The output variable `optStruct` is a structure containing the optional information. The output structure is provided to the built-in function as shown below:

```
tSol,ySol=ode45(odeFun, tSpan, Yo, optStruct)
```

An alternative way of getting the outputs from the built-in solver is to use a return structure:

```
sol = ode45(odeFun, tSpan, Yo, optStruct)
```

Now `sol` is a MATLAB structure that contains all of the output data for the dependent and independent variable in one object. Readers are encouraged to consult the MATLAB documentation for way in which the solution structure object should be used.

Part XII

Solving Boundary Value Problems

Lecture 28 - Solving Boundary value Problems Using the Shooting Method

Objectives

The objectives of this lecture are to:

- Review basic concepts for Boundary Value Problems with a single independent variable.
- Describe the Shooting Method.
- Do an example problem.

Boundary Value Problems

A boundary value problem (BVP) consists of a governing equation and boundary conditions. For a second-order boundary value problem with one spatial dimension¹, the general form of the governing equation is given in Equation 199:

$$\frac{d^2u}{dx^2} = f\left(x, u, \frac{du}{dx}\right), \quad a \leq x \leq b \quad (199)$$

where it is understood that $a < b$.

In order to obtain a unique solution, suitable boundary conditions must be provided. Linear boundary conditions for the second-order problem take the form shown below:

$$\begin{aligned} A_1u(a) + B_1u'(a) &= C_1 \\ A_2u(b) + B_2u'(b) &= C_2 \end{aligned}$$

where A_i , B_i , and C_i are constants and are not all zero for any value of i .

KNOWING WHAT CONSTITUTES a *suitable* set of boundary conditions is part of your job as an engineer. Depending on the given boundary

¹ Here we assume that the independent variable is a *spatial* variable. This is the conventional approach for applications of interest for this class.

conditions, the BVP may have one unique solution, no solution, or infinitely many solutions. Sometimes it is hard to tell in advance which of these will turn out to be the case. Physical insight and intuition can play an important role in predicting these outcomes so it is essential that one understands the physical interpretation of a proposed set of boundary conditions.

There are three basic types of boundary conditions:

- **Type 1** or *Dirichlet* boundary conditions. For this type of boundary condition, the value of the dependent variable is directly fixed on the boundary. For example:

$$u(a) = 100$$

For a BVP related to the heat equation, for instance, this is equivalent to specifying the temperature on a boundary.

- **Type 2** or *Neumann*² boundary conditions. For this type of boundary condition, the derivative of the dependent variable is directly fixed on the boundary. For example:

$$\frac{du}{dx} \Big|_{x=b} = 0$$

For a BVP related to the heat equation where the dependent variable is temperature, this is equivalent to specifying the heat flux on a boundary. For the example given, if we set the heat flux equal to zero, that is interpreted as an *insulated* boundary condition.

- **Type 3** or *Robin*³ or just *mixed* boundary conditions. As the reader may have deduced by now, boundary conditions of this type involve both the dependent variable and its derivative. For example:

$$\frac{du}{dx} \Big|_{x=a} = -h(u(a) - T_{\text{env}})$$

where T_{env} refers to the environmental temperature near the boundary and h is a non-negative constant. This example corresponds to heat transfer by convection. The constant h is the convective heat transfer coefficient and relates heat flux at the boundary to the difference between the temperature at the boundary of the domain, $u(a)$, and T_{env} . If h is high then a large heat flux can be passed through the boundary with only a small temperature difference; if h is low then the surface temperature must be much higher than the surrounding environment to transmit significant amounts of heat. In the limit of $h \rightarrow 0$, the boundary becomes insulated.

Note: This classification scheme is also discussed in Lecture 22 of the Analytical Methods portion of this text.

² Here we refer to Carl Gottfried Neumann who was a German mathematician in the late 19th and early 20th century. He taught at several universities and carried out research in pure and applied mathematics. There are several other mathematical terms named after him including the Neumann series, the Neumann boundary value problem and the Neumann-Poincaré operator.

³ Named for Victor Gustave Robin who was a French mathematician who lectured at the Sorbonne in Paris. To the best of this author's knowledge, he was not associated in any way with Batman.

The Shooting Method

We will introduce and illustrate our first method for solving BVPs, the shooting method, with an example.

Problem Statement:

A pin fin is a slender extension attached to increase the surface area and enable greater heat transfer. When convection and radiation are included in the analysis, the steady-state temperature distribution, $T(x)$, along a pin fin can be calculated from the solution of the equation below:

$$\frac{d^2T}{dx^2} - \frac{h_c P}{k A_c} (T - T_s) - \frac{\epsilon \sigma_{SB} P}{k A_c} (T^4 - T_s^4) = 0, \quad 0 \leq x \leq 0.1 \quad (200)$$

with boundary conditions $T(0) = T_A$ and $T(0.1) = T_B$. A schematic of the system is shown in Figure 151. There are a number of parameters given in the equation. These are specified in Table 25.

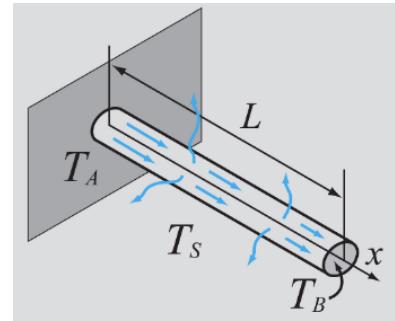


Figure 151: Pin Fin Boundary Value Problem Schematic.

Parameter	Value
Convective heat transfer coefficient (h_c)	40 W/m ² -K
Perimeter of the pin (P)	0.016 m
Radiative emissivity of the surface (ϵ)	0.4
Thermal conductivity of the pin material (k)	240 W/m-K
Cross-sectional area of the fin (A_c)	1.6×10^{-5} m ²
Temperature of surrounding air (T_s)	293 K
Stefan-Boltzmann constant (σ_{SB})	5.67×10^{-8} W/m ² -K ⁴
Temperature of fin at base (T_A)	473 K
Temperature of fin at end (T_B)	293 K

Table 25: Example problem parameters.

IT IS WORTH taking a moment to classify the given problem. This is a 2nd-order, non-homogeneous, non-linear, boundary value problem with non-homogeneous type-1 boundary conditions. Since it is non-linear and also not separable, none of the analytical methods we learned in the first portion of this book are applicable. We need to solve the problem numerically. Since we just finished a long section describing numerical methods for initial value problems, maybe there is a way we can use one those tools—like an explicit Runge-Kutta method—to solve this problem. With the Shooting method, we do exactly that.

Algorithm

1. **Formulate your BVP as an IVP.** Here we will introduce a new dependent variable, w :

$$w = \begin{bmatrix} T \\ T' \end{bmatrix}, \quad w(0) = \begin{bmatrix} T_A \\ T'_A \end{bmatrix}$$

$$dw = \begin{bmatrix} T' \\ T'' \end{bmatrix}$$

$$= \begin{bmatrix} w(2) \\ \frac{h_c P}{k A_c} (w(1) - T_s) + \frac{\epsilon \sigma_{SB} P}{k A_c} (w(1)^4 - T_s^4) \end{bmatrix}$$

Note: the “initial” condition T'_A is not part of the given boundary value problem. We will deal with that in the next step.

2. **Provide an estimate of the initial values.** We re-expressed our BVP as an IVP. The problem is, of course, that the BVP specifies the temperature at $x = 1$, but not the slope of the temperature at $x = 0$ that we use in the IVP statement. From the physics of the problem, we expect that $T'(0)$ will be *negative*. The whole point of the pin fin is to draw energy from whatever it is attached to (at $x = 0$) and transfer it by convection to the environment; this will result in a negative temperature gradient at the root of the fin. What we do *not* know is the actual value of $T'(0)$ that will result, upon solving the IVP, in the temperature at the tip of the fin being equal to the boundary condition specified in the BVP, $T(0.1) = T_B$. So we pick an informed estimate of $T'(0)$.⁴

3. **Solve the IVP using your method of choice.** We have introduced several: Euler’s method, Midpoint method, any kind of Runge-Kutta method whether a method built in to MATLAB or something you wrote yourself.⁵
4. **Evaluate $T(b)$ and compare with T_B .** If our aim was true, we would “hit” $T(b) = T_B$ with our “shot.” In MATLAB you might do this as follows:

- (a) Define a function that takes the estimate for $T'(0)$ as an argument and returns the resulting value of $T(b)$.
- (b) Create a second function that returns the difference between the function described above, and T_B . For example:

```
tgt_err_fun = @(dTa) fun(dTa) - Tb
```

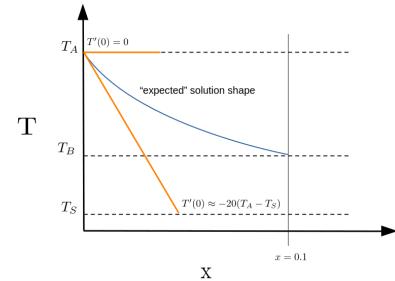


Figure 152: Notional pin temperature profile.

⁴ If we adopt the logical “firearms analogy” for the shooting method, think of this step as *taking aim*.

⁵ Continuing in the analogy, this is where you take your shot. If it makes the whole affair any less tedious, you might imagine a squad of storm troopers with their blasters chasing rebel scum through the corridors of the Death Star: “pew, pew, pew...”

where dTa is the estimated value for $T'(0)$, fun is the aforementioned function, and Tb is T_B .

5. **Iteratively repeat steps 2-4 until all boundary conditions are satisfied.** In this case, we iterate until $T(0.1) = T_B$. If we adopt the approach introduced in step 4, we can use one of the several algorithms for solving non-linear equations that we studied earlier in this class.⁶

```
%>>> %% Lecture 28 - Solving BVPs with the Shooting Method
clear
clc
close 'all'
%% Define the ODE and all parameters
a = 0; b = 0.1;
Ta = 473; % K, temperature at base of fin
Tb = 293; % K, temperature at the tip of the fin
N = 1000; %
ode = @(x,T) pin_fin(x,T); ❶
```

```
%>>> %% set RK4 solver tableau parameters
% note: any other ODE solver would also suffice.
solver = @odesExplicitRK; ❷
s = 4;
BT = zeros(s+1,s+1); % Butcher Tableau for RK method
C = [0; 1/2; 1/2; 1; 0]; % sample points
B = [0 1/6 2/6 2/6 1/6]; % weights
A = [0 0 0 0; % RK matrix
      1/2 0 0 0;
      0 1/2 0 0;
      0 0 1 0];
BT(:,1) = C;
BT(end,:)= B;
BT(1:s,2:end) = A;
```

```
%>>> %% Commence Shooting Method
odeWithSolver = @(dT_a_g) solver(ode,a,b,N,[Ta;dTa_g],BT); ❸
tgt_err_fun = @(dT_a_g) shot_function(odeWithSolver,dTa_g)
                  - Tb; ❹
```

Since some of the root-finding methods that we can choose from require two estimates of the slope and, for bisection method, the estimates must bracket the actual slope, we will propose a “high” and “low” estimate for the initial slope.

```
% need two guesses at the slope
dT_a_H = 0; % slope = 0 at x=a means no heat transfer out.
dT_a_L = 4*(Tb - Ta)/(b-a); % guess a strong linear function
```

For this demo, I like to choose from among several available algorithms for solving nonlinear equations. The functions BisectionRoot and SecantRoot are unchanged from when they were introduced earlier in the course.

Note that when this block of code is complete, we will have solved for the value of $T'(0)$ —denoted dT_a —that results in $T(0.1) = T_B$.

⁶ Noting that the bisection and secant methods each require two estimates for $T'(0)$ and further, for the bisection method, the two slope estimates must bracket the value of $T'(0)$ that results in $T(0.1) = T_B$.

❶ The governing equation is implemented as a local function and will be provided at the end of the script.

❷ For this example we will use our own implementation of the classical 4th-order Runge Kutta method. As an alternative we could use one of MATLAB’s built-in methods.

❸ Here we define a function that will take the ODE, a solve, and a guessed value for the initial conditions as needed for the shooting method.

❹ We also need to define a function that will solve the ODE with the guessed value and return 0 when the “correct” value is guessed. As written, this function can thus be passed to any root-finding algorithm.

```

method = 4;
switch method
    case 1
        tol = 1e-4; % tolerance on convergence to the BC
        dT_a = BisectionRoot(tgt_err_fun ,dT_a_H,dT_a_L,tol);
    case 2
        tol = 1e-4; % tolerance on convergence to the BC
        imax = 100;
        dT_a = SecantRoot(tgt_err_fun ,dT_a_H,dT_a_L,tol ,imax);
    case 3
        dT_a = fzero(tgt_err_fun ,dT_a_H);
    case 4
        dT_a = fsolve(tgt_err_fun ,dT_a_H);
    otherwise
        error('Invalid Solver Choice!');
end

```

Now we should visualize the result.

```

%% Solve one last time with converged dT_a
T = odeWithSolver(dT_a);

x = linspace(a,b,N);
figure(1)
plot(x,T(1,:),'-b','linewidth',3);
title('Pin Fin Temperature Profile',...
    'fontsize',14,'fontweight','bold');
xlabel('X [m]', 'fontsize',12,'fontweight','bold');
ylabel('T [^\circ C]', 'fontsize',12,'fontweight','bold');
grid on
set(gca,'fontsize',10,'fontweight','bold');

```

The resulting temperature profile through the pin fin is shown in Figure 153.

Local functions defined above are included here.

```

%% Local Functions
function Tb = shot_function(odeWithSolver,dT_a)
% given an estimate for T(a), solves the IVP and returns
% the value of T(b).
T_trial = odeWithSolver(dT_a);
Tb = T_trial(1,end);
end

function F = pin_fin(~,T)
% define the IVP
% two args so it can work with ode45
Ac = 1.6e-5; % m^2, fin cross sectional area
P = 0.016; % m, perimeter of pin cross section
h_c = 40; % W/m^2-K, convective heat transfer coefficient of air
k = 250; % W/m-K, thermal conductivity of pin material
emiss = 0.5; % emissivity of pin material
sigma_sb = 5.67e-8; % W/m^2-K^4, Stefan-Boltzmann constant
Ts = 293; % K, temperature of surrounding air

F = nan(2,1);
F(1) = T(2);

```

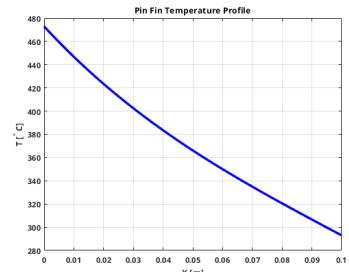


Figure 153: Shooting Method Example Solution.

Note: Take a few moments to think about this figure. Does the shape match your expectation? How do you think the shape of the curve would change if h_c were much larger or smaller? Readers are strongly encouraged to solve this problem and test these hypotheses.

It is not against the law also to critically consider the given boundary conditions. How realistic is it to specify the temperatures at both ends of the pin fin? What aspects of the solution should you look at to evaluate the performance of the pin fin as a heat management device?

```

F(2) = ((h_c*P)/(k*Ac))*(T(1) - Ts) + ...          85
      ((emiss*sigma_sb*P)/(k*Ac))*(T(1).^4 - Ts.^4); 86

end                                                 87

function y = odesExplicitRK(ODE,a,b,N,yINI,BT)    88
% function y = odeExplicitRK(ODE,a,b,h,yINI,BT)    89
% y = solution (vector)                            90
% ODE = function handle for y'                   91
% a,b = begining and end of the interval for solution 92
% N = number of steps between a and b            93
% yINI = initial value for the solution         94
% BT = Butcher Tableau                          95

% get Butcher Tableau Parameters                96
s = length(BT)-1;                                97
c = BT(1:s,1);                                  98
B = BT(s+1,2:end);                            99
A = BT(1:s,2:end);                            100
stages = s;                                     101

x = linspace(a,b,N);                           102
sys_size = length(yINI);                      103
y = nan(sys_size,N);                         104
y(:,1) = yINI;                               105
h = x(2)-x(1);                             106
for t = 1:(N-1)                            107
    Xi = nan(sys_size,stages);             108
    for s = 1:stages                     109
        Xi(:,s) = y(:,t);                 110
        for i = 1:(s-1)                  111
            Xi(:,s) = Xi(:,s) + h*A(s,i)*ODE(x(t)+c(i)*h,Xi(:,i)); 112
        end                                113
    end                                114
    y(:,t+1) = y(:,t);                  115
    for i = 1:stages                  116
        y(:,t+1) = y(:,t+1) + h*B(i)*ODE(x(t)+c(i)*h,Xi(:,i)); 117
    end                                118
end                                119
y(:,t+1) = y(:,t);                  120
for i = 1:stages                  121
    y(:,t+1) = y(:,t+1) + h*B(i)*ODE(x(t)+c(i)*h,Xi(:,i)); 122
end                                123

end                                124
end                                125

function x_bi = BisectionRoot(fun,a,b,TOL)    126
tol = TOL;                                 127
x_bi = (a+b)/2;                            128
% verify that fun(a) and fun(b) have different sign 129
if (fun(a)*fun(b)>0)                      130
    error('No root lies between %g and %g!\n',a,b);
end                                         131

maxIt = 1e2;                                132
for k = 1:maxIt                            133
    FxNS = fun(x_bi);                      134
    % check if FxNS is within the tolerance 135
    if (abs(FxNS) < tol)                  136
        return;                            137
    end                                138
end                                         139

```

```

% update brackets
if (fun(a)*FxNS < 0)
    b = x.bi;
else
    a = x.bi;
end

% update estimated x.bi based on new bracket
x.bi = (a+b)/2;

end

% if I ever get here, then I failed to meet the tolerance
% within the maximum number of iterations
fprintf('Warning: failed to find root within specified tolerance
.\n');
fprintf('Current root: %g, fun(x.bi) = %g. \n',x.bi,fun(x.bi));

end

function Xs = SecantRoot(Fun,Xa,Xb,Err ,imax)

for i = 1:imax
    FXb = Fun(Xb);
    Xi = Xb - FXb*(Xa - Xb)/(Fun(Xa) - FXb);

    if abs((Xi - Xb)/Xb) < Err
        Xs = Xi;
        break;
    end
    Xa = Xb;
    Xb = Xi;

end
end

```

Lecture 29 - Solving Boundary Value Problems, MATLAB Built-in Methods

Objectives

The objectives of this lecture are to:

- Introduce MATLAB built-in functions `bvp4c` and `bvp5c` and show how to use them.
- Do an example problem.

BVP4C and BVP5C

Both of these built-in solvers use variations of the finite difference method. For this lecture, we will *not* delve into any of the inner working details of the functions. Rather, I will seek only to show you how to *use* the functions. The interface for both functions are the same so, for all intents and purposes relevant for this class, you can use them interchangeably. This is not to say that the differences between the two functions are trivial. Interested readers are advised to consult the MATLAB documentation to learn more.

To GET A SOLUTION to a BVP using `bvp4c` or `bvp5c`, the user has to provide the typical information:

1. *The governing equation.* As with other built-in ODE solvers, this is provided as a function that returns the value for:

$$\frac{dw}{dx} = f(x, w)$$

where here we indicate the dependent variable with w . Since the BVPs of interest to us are 2nd-order and higher, the dependent variable w will be a vector and the governing equation for an n^{th} -order BVP will be a system with n equations. In the examples presented here, this function will ordinarily be implemented as

a local function. For particularly simple governing equations, an inline “anonymous” function may be used.

```
function dw = fun(x,w)
%
% governing equation implemented here %
%
end
```

2. *Boundary conditions.* Recall with IVP solvers like ode45 or ode78 initial conditions were simply given as vectors.¹ This is roughly equivalent to having only type 1 and type 2 boundary conditions. In order to gain the expressiveness that we need, a different approach is taken for BVP solvers. We provide a *residual function* that accepts as arguments the dependent variable and its derivatives at each boundary; the function returns the boundary condition in *residual form*.² Using the example from Lecture 28 where the boundary conditions were:

$$T(0) = T_A, \quad T(0.1) = T_B$$

we could encode these as follows:

```
bcbfun = @(wa,wb) [wa(1) - Ta; wb(1) - Tb];
```

where $wa(1)-Ta$ is equivalent to $T(0) - T_A$; and $wb(1)-Tb$ is equivalent to $T(0.1) - T_B$.³

3. *Initial mesh and solution guess.* Like other built-in MATLAB functions for solving ODEs, bvp5c will adapt the mesh as needed to provide a solution that satisfies the specified error tolerances.⁴ To specify the mesh and initial solution guess, we use the built-in function bvpinit as shown in the listing below:

```
a = 0; b = 1; % domain 0 < x < 1
Yguess = [1 0]; % guess constant solution
initial_mesh = [a b]; % as coarse as possible ❶
solinit = bvpinit(initial_mesh ,Yguess);
```

Alternatively, you may specify a more refined initial mesh using a tool like linspace.

```
a = 0; b = 1; N = 100;
Yguess = [1 0]; ❷
initial_mesh = linspace(a,b,N);
solinit = bvpinit(initial_mesh ,Yguess);
```

4. *Provide any options.* Default values are provided for all needed options. Any named parameter can be set using the built-in function bvpset(name1,value1,name2,value2,...). An example is shown in the listing below.

¹ This reflects the difference in what types of conditions are given for IVPs compared to BVPs. For IVPs, the only option presented for initial conditions was to make a direct assignment to the dependent variable and its derivative at one end of the domain.

² By *residual form* we just mean that the boundary condition is used as it is stated but with all of the terms moved to the left-hand-side of the equation.

³ If I need to refer to the derivative of the dependent variable at either boundary, I would refer to them as $wa(2)$ or $wb(2)$, respectively.

⁴ As with other solvers, absolute and relative error tolerances are provided by default and can be changed by the user as desired.

❶ bvp5c will adapt this mesh as needed. Note that the final mesh will not generally be uniform.

❷ I have yet to encounter a BVP where it was necessary to provide an initial solution guess more detailed than a constant value.

```
options = bvpset('RelTol',1e-3,'AbsTol',1e-6,'NMax',1000);  
1
```

THE SYNTAX FOR a typical call to bvp5c is shown in the listing below:

```
sol = bvp5c(fun,bcfun,solinit,options);  
1
```

The return value `sol` is a structure that contains, among other information, fields for the solution, `sol.y`, at discrete grid points, `sol.x`.⁵

Example #1

Consider the following boundary value problem:

Governing Equation: $\frac{d^2y}{dx^2} - y = \sin(x), \quad 0 < x < 2$
 BCs: $y(0) = 1, \quad y(2) = 0$

Using basic techniques you can show that the general solution to the governing equation is:

$$y(x) = c_1 \sinh(x) + c_2 \cosh(x) - \frac{1}{2} \sin(x)$$

Applying the boundary conditions we get the following values for the remaining constants:

$$c_1 = \frac{\frac{1}{2} \sin(2) - \cosh(2)}{\sinh(2)}$$

$$c_2 = 1$$

Let us use MATLAB and bvp5c to solve this BVP numerically.

```
clear  
clc  
close 'all'  
%% Construct Exact Solution  
C1 = (0.5*sin(2)-cosh(2))/sinh(2);  
C2 = 1;  
y_exact = @(x) C1*sinh(x) + C2*cosh(x) - 0.5*sin(x);  
  
%% Solve with BVP5C  
Ya = 1; Yb = 0;  
xMin = 0; xMax = 2;  
Yguess = [1 0];  
F = @(x,w) [w(2); w(1)+sin(x)]; ❶  
bcfun = @(ya,yb) [ya(1)- Ya; yb(1)-Yb];  
solinit = bvpinit([xMin xMax],Yguess);  
options = bvpset('RelTol',1e-10,'AbsTol',1e-8,'NMax',5000);  
sol5c = bvp5c(F,bcfun,solinit,options);  
  
%% Calculate the Relative Error  
rel_err = norm(y_exact(sol5c.x)-sol5c.y(:,1),2) / ...  
norm(y_exact(sol5c.x),2); ❷  
fprintf('Relative error = %g \n',rel_err);  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22
```

⁵ Once again, the reader is encouraged to review the MATLAB documentation to learn what other useful information is packed into the other fields of the output solution structure.

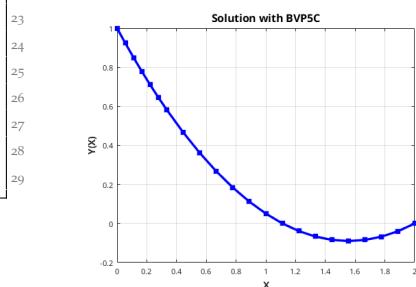
This is a 2nd-order, linear, constant-coefficient, non-homogeneous ordinary differential equation. See Lecture 5 in the analytical methods portion of this text for a review on how to solve problems like this.

Note: It is always a good idea, when learning to use new tools, to use the tool on a problem for which you already know the correct answer.

❶ This is one of the simple cases where the governing equation can easily be implemented with an in-line function.

❷ The output solution structure, `sol5c`, also has a field containing the maximum residual error in the field: `sol5c.stats.maxerr`.

```
%>> Plot the solution
plot(sol5c.x,sol5c.y(1,:),'-sb','linewidth',3);
title('Solution with BVP5C','fontsize',14,...
'fontweight','bold');
xlabel('X','fontsize',12,'fontweight','bold');
ylabel('Y(X)','fontsize',12,'fontweight','bold');
grid on
```



A plot of the solution is shown in Figure 154. The initial maximally coarse grid has been refined adaptively to arrive at a solution that satisfies the user-specified relative error tolerance.

Example #2

The axial temperature variation of a current-carrying bare wire, as illustrated in Figure 155, is described by the following boundary value problem:

$$\text{ODE: } \frac{d^2T}{dx^2} - \frac{4h}{kD} (T - T_\infty) - \frac{4\epsilon\sigma_{SB}(T^4 - T_\infty^4)}{kD} = -\frac{I^2\rho_e}{k(\frac{1}{4}\pi D^2)^2}$$

$$\text{BCs: } T(x=0) = 300K, \quad \left.\frac{dT}{dx}\right|_{x=L/2} = 0$$

where T is the temperature in Kelvin, x is the coordinate along the wire, $k = 72\text{W/m-K}$ is the thermal conductivity, $h = 2000\text{W/m}^2\text{-K}$ is the convective heat transfer coefficient, $\epsilon = 0.1$ is the radiative emissivity, $\sigma_{SB} = 5.67 \times 10^{-8}\text{W/m}^2\text{-K}^4$ is the Stefan-Boltzmann constant, $I = 2$ amps is the current, $\rho_e = 32 \times 10^{-8}$ Ohm-m is the electrical resistivity, $T_\infty = 300$ K is the ambient temperature, $D = 7.62 \times 10^{-5}\text{m}$ is the wire diameter, and $L = 4.0 \times 10^{-3}\text{m}$ is the length of the wire.

```
F = @(x,t) ex2(x,t); ❶
L = 4e-3; % m, length of wire
xMin = 0; xMax = L/2;
To = 300; % K, temperature at x=0

bcfun = @(ya,yb) [ya(1)- To; yb(2) - 0]; ❷
Tguess = [To 0]; % second entry is dT/dx
solinit = bvpinit([xMin xMax],Tguess);
options = bvpset('RelTol',1e-10,'AbsTol',1e-8,'NMax',5000);
sol2 = bvp5c(F,bcfun,solinit,options);

figure(1)
plot(sol2.x,sol2.y(1,:),'-sb','linewidth',3);
grid on
title('Example 2 Solution','fontsize',16,...
'fontweight','bold');
xlabel('X [m]','fontsize',14,'fontweight','bold');
ylabel('T(X) [K]','fontsize',14,'fontweight','bold');
set(gca,'fontsize',12,'fontweight','bold');
```

Figure 154: Solution of example problem with bvp5c.

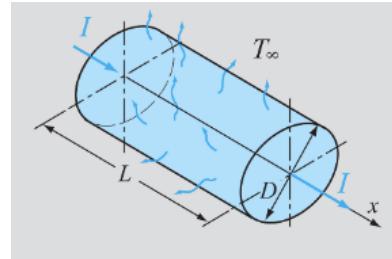


Figure 155: Current-carrying wire schematic.

Note: This is a 2nd-order, non-linear, non-homogeneous boundary value problem. This author is not aware of any analytic technique that would work to find an exact solution. Nonetheless, we will be able to solve this problem using bvp5c with not much more effort than what was required for the last example; the difference lies mainly in implementing the governing equation.

❶ For this problem it is somewhat less practical to use an in-line function to implement the governing equation. We include ex2(x,t) as a local function.

❷ Obviously we could have omitted the zero in the expression 'yb(2)-0', but for the sake of clarity it is not a horrible idea to include that term. The point of this example is to illustrate a problem with a type 2 boundary condition.

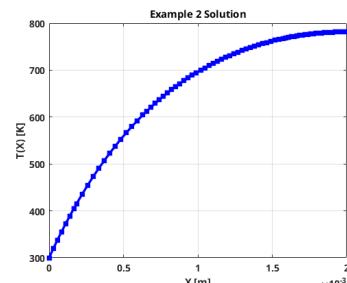


Figure 156: Solution for Example #2.

The solution is shown in Figure 156. The local function used to implement the governing equation is shown in the next listing.

```

%% Local Functions
function dTdx = ex2(~,T) %③
k = 72; % W/m-K
h = 2000; % W/m^2-K
emiss = 0.1;
sigma_sb = 5.67e-8; % W/(m^2-K^4)
I = 2; % amps
rho_e = 32e-8; % Ohm-m
Tinf = 300; % K
D = 7.62e-5; % m

% make some terms to simplify the equation
c1 = 4*h/(k*D);
c2 = 4*emiss*sigma_sb/(k*D);
c3 = -I^2*rho_e/(k*(0.25*pi*D^2)^2);
dTdx = [T(2);
    c1*(T(1) - Tinf) + c2*(T(1).^4 - Tinf^4) + c3];
end

```

③ Reminder that, since the governing equation is not a function of the independent variable—it is *autonomous*, in the language of ODEs—we can (and should) use a tilde to take the place of the first argument of the function.

Assignment #9

1. Write a user-defined MATLAB function that solves, with the shooting method in conjunction with the secant method, a second-order boundary value problem of the form:

$$\frac{d^2y}{dx^2} + f(x) \frac{dy}{dx} + g(x)y = h(x)$$

for $a \leq x \leq b$ with $y(a) = Y_a$ and $y(b) = Y_b$, where Y_a and Y_b are constants. For the function name and arguments use: $[x,y] = \text{BVPShootSecant}(F_x, G_x, H_x, a, b, n, Y_a, Y_b, W_a, W_b)$. The input arguments F_x , G_x , and H_x are names for the functions that calculate $f(x)$, $g(x)$, and $h(x)$ respectively. The arguments a and b define the domain if the solution, n is the number of subintervals, Y_a and Y_b are the boundary conditions, and W_a and W_b are the assumed slopes at $x = a$ that are used in the first two iterations of the Secant Method. Within the user-defined function BVPShootSecant , use the function odesCRK4 that you created in Assignment #8. The secant method should be carried out until the absolute value of the true error at $x = b$ is smaller than 0.001.

Use this function to solve the following boundary value problem:

$$\frac{d^2y}{dx^2} + 2x \frac{dy}{dx} + 5y - \cos(3x) = 0, \quad 0 \leq x \leq \pi$$

with boundary conditions: $y(0) = 1.5$, and $y(\pi) = 0$. Use $n = 100$, $W_a = -5$, and $W_b = -1.5$. Create a well-formatted plot of the solution.

2. A cylindrical pipe with inner radius 1 cm and the wall thickness 2.5 cm carries a fluid at a temperature of 600°C. The outer wall of the pipe is at 25°C. The governing equation for the temperature distribution in the pipe wall is:

$$r \frac{d^2T}{dr^2} + \frac{dT}{dr} = -500$$

subject to the boundary conditions $T(1) = 600^\circ\text{C}$ and $T(3.5) = 25^\circ\text{C}$. Solve for the temperature $T(r)$ and plot the temperature distribution.

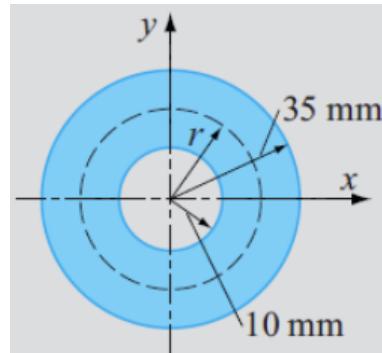


Figure 157: Schematic of cylindrical pipe.

- (a)
 - i. Solve using BVPShootSecant that you created for the first problem.
 - ii. Solve using bvp5c.
Plot the solution using both methods and output the value for $\frac{dT}{dr} \Big|_{r=3.5}$ using both methods.
- (b) What is the physical significance of the right-hand side (-500) in the governing equation? (i.e. what conservation law is being expressed with the governing equation?) What happens to the temperature profile when the right-hand side is made much bigger?
- (c) Replace the boundary condition at $r = 3.5\text{cm}$ to model convective heat transfer to a fluid medium maintained at 25°C. Assume a convective heat transfer coefficient of 1.5 W/cm²-K.

$$\frac{dT}{dr} \Big|_{r=3.5} = -h [T(3.5) - 25]$$

Solve the problem using bvp5c, plot the temperature profile and output $T(r = 3.5)$. Experiment with different values of the convective heat transfer coefficient to make sure the response is as you expect. What happens as the convective heat transfer coefficient gets very large? What happens when it gets small? Briefly describe your observation.

3. The radial distribution of temperature in a current-carrying bare wire is described by:

$$\frac{k}{r} \frac{d}{dr} \left(r \frac{dT}{dr} \right) = -\frac{I^2 \rho_e}{\left(\frac{1}{4} \pi D^2 \right)^2}$$

where T is the temperature in K, r is the radial coordinate in meters, $k = 72 \text{ W/m/K}$ is the thermal conductivity, $I = 0.5\text{A}$ is the current, $\rho_e = 32 \times 10^{-8} \Omega\text{-m}$ is the electrical resistivity, and $D = 1 \times 10^{-4}\text{m}$ is the wire diameter. Use MATLAB's built-in function `bvp5c` to solve the equation for $T(r)$. Solve twice for the following boundary conditions.

- (a) at $r = 10^{-6}\text{m}$, $\frac{dT}{dr} = 0$ and $r = D/2$, $T = 300\text{K}$.
- (b) at $r = 10^{-6}\text{m}$, $\frac{dT}{dr} = 0$ and at $r = D/2$, $\frac{dT}{dr} = -\frac{h}{k}(T(D/2) - T_\infty)$, where $h = 100 \text{ W/m}^2\text{-K}$ is the convective heat transfer coefficient and $T_\infty = 300\text{K}$ is the ambient temperature.

Important note: $r = 0$ is a singular point and therefore must be replaced with a small, non-zero value, for example 10^{-6} . As initial guesses, use $T = 500\text{K}$ and $\frac{dT}{dr} = 0$, use 50 subintervals.

For both parts a) and b), plot the temperature profiles and output the difference in temperature between the center-line of the wire and the outer surface of the wire. Give a brief physical explanation for the differences in the temperature profiles for part a) and b).

Lecture 30 - More Boundary Value Problem Examples

Objectives

The objectives of this lecture are to:

- Illustrate the use of Robin boundary conditions with another example.
- Show how to solve BVPs with an unknown parameter with bvp5c.

Example Problem #1

Fuel rods of a nuclear reactor are often constructed as cylindrical structures with the fuel retained inside cladding material. The fuel causes heat to be generated by nuclear reactions within the cylinder as well as in the cladding.¹ The outer surface of the cladding is cooled by flowing water at $T_\infty = 473\text{K}$ with heat transfer coefficient of $h = 10^4 \text{ W/m}^2\text{-K}$. The thermal conductivity of the cladding material is $k = 16.75 \text{ W/m}\cdot\text{K}$. The dimensions of the fuel rod are $R = 1.5 \times 10^{-2} \text{ m}$, and $w = 3.0 \times 10^{-3} \text{ m}$. The temperature distribution in the cladding is determined by the solution of the following boundary value problem. We will use MATLAB's built-in function

$$\begin{aligned} \text{ODE: } & \frac{1}{r} \frac{d}{dr} \left(rk \frac{dT}{dr} \right) = -10^8 \frac{e^{-r/R}}{r}, \quad R < r < R + w \\ \text{BCs: } & \left. \frac{dT}{dr} \right|_{r=R} = -\frac{6.32 \times 10^5}{k}, \quad \left. \frac{dT}{dr} \right|_{r=R+w} = -\frac{h}{k} (T(r + w) - T_\infty) \end{aligned}$$

bvp5c to solve the boundary value problem and plot the temperature distribution in the cladding as a function of radial position, r .

First, we must transform the governing equation:

$$\begin{aligned} \frac{1}{r} \frac{d}{dr} \left(rk \frac{dT}{dr} \right) &= -10^8 \frac{e^{-r/R}}{r}, \quad \text{Multiply both sides by } r. \\ \frac{d}{dr} \left(rk \frac{dT}{dr} \right) &= -10^8 e^{-r/R}, \quad \text{Apply the product rule to the left-hand side.} \\ k \frac{dT}{dr} + rk \frac{d^2T}{dr^2} &= -10^8 e^{-r/R}, \quad \text{Divide both sides by } rk \text{ and re-arrange terms.} \\ \frac{d^2T}{dr^2} + \frac{1}{r} \frac{dT}{dr} &= -\frac{10^8 e^{-r/R}}{rk} \end{aligned}$$

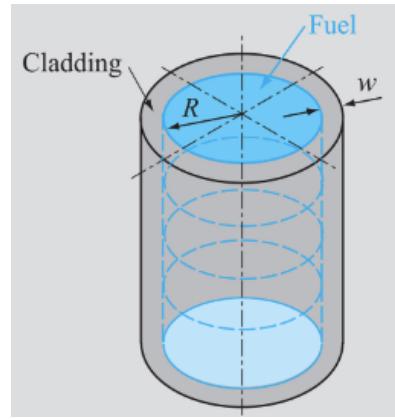


Figure 158: A typical nuclear reactor fuel pin.

¹ The former is energy produced by fission, the latter is energy deposited by fission-induced photons within the cladding material.

NOW WE ARE ready to solve this problem with `bvp5c` in the way described in the last lecture. The resulting temperature profile is shown in Figure 159.

```

1 clear
2 clc
3 close 'all'

4 %% Parameters
5 R = 1.5e-2; % m, radius of fuel
6 w = 3.0e-3; % m, thickness of cladding
7 k = 16.75; % W/(m-K), thermal conductivity of clad
8 Q = 1e8; % W/m^2, Source term from heat dep in cladding.

9 Q2 = 6.32e5; % W/m^2, Heat flux due to heat produced in fuel.
10 T_inf = 423; % K, temperature of water flowing on cladding
11 h = 1e4; % W/(m^2-K), convective heat transfer coefficient

12 %% Encode the governing equation and boundary conditions
13
14 F = @(r,T) [T(2); -(1./r)*T(2) - Q./(k*r)*exp(-r/R)];
15 bcfun = @(Ta,Tb) [Ta(2)+ Q2/k; ...
16 Tb(2) + (h/k)*(Tb(1)-T_inf)];
17
18 %% Establish initial mesh and solution estimate
19 rMin = R; rMax = R+w; nR = 200;
20 Tguess = [T_inf 0];
21 solinit = bvpinit(linspace(rMin,rMax,nR),Tguess);
22
23 %% Solve and plot the result
24 sol = bvp5c(F,bcfun,solinit);
25
26 figure(1)
27 plot(sol.x,sol.y(1,:),'-b','LineWidth',2);
28 title('Fuel Clad Temperature','FontSize',16,...
29 'FontWeight','bold');
30 xlabel('R [m]','FontSize',14,'FontWeight','bold');
31 ylabel('T(R) [K]','FontSize',14,'FontWeight','bold');
32 grid on
33 set(gca,'FontSize',12,'FontWeight','bold');
34
35
36

```

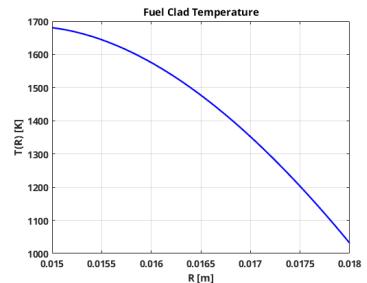


Figure 159: Cladding temperature profile.

Note: Take a moment to think about this temperature profile and consider the following questions:

1. **Question:** What direction is heat flowing? **Answer:** Heat is flowing from the cladding inner surface (inward heat flux is imposed as the boundary condition at the fuel/clad interface) through the cladding and out through the outer surface to the coolant.
2. **Question:** Does the downward curvature of the temperature profile make sense? **Answer:** Yes! In heat transfer class you may have been taught that curvature of the temperature profile indicates heat generation in the region. In this case, it is not *heat generation* so much as it is *heat deposition* but it acts the same mathematically.
3. **Question:** What controls the temperature at the inner and outer surface of the cladding? (there were no type 1 boundary conditions) **Answer:** The outer surface temperature is related to: a) the amount of heat that must be passed through the boundary—all the heat passed into the cladding from the fuel plus all of the heat deposited in the cladding due to the radiation; and b) the coolant temperature, T_∞ , and convective heat transfer coefficient.

READERS ARE ENCOURAGED to read and consider the questions and answers posed in the margin. Do not forget that *the purpose of computing is insight*. We are not solving these equations like they are a puzzle or to make impressive pictures but rather to improve our understanding of the physical processes represented by the boundary value problem. Take some time to run the MATLAB example and then experiment by changing parameters such as the thermal conductivity of the cladding, k , the convective heat transfer coefficient, h , or the source term for heat deposition in the cladding, Q . Do the changes you see to the results make sense? How can you, as an engineer, actually change those parameters in a real physical system?

Example Problem #2

From the reactor fuel pin in Example #1, consider the inhomogeneous source term representing heat deposition in the cladding material:

$$S(r) = -10^8 \frac{e^{-r/R}}{r}$$

Suppose that the leading constant of 10^8 was an *unknown parameter*, Q . We might pose a question like: How high can Q be while maintaining temperature at the cladding inner surface, $T(r = R)$, less than 1500K? Clearly, all other parameters being the same, Q would have to be *smaller* than 10^8 since, from our calculation in Example #1, for that value the cladding inner surface temperature is nearly 1700K.

We could approach this problem by systematically changing Q by hand and running the script. MATLAB allows users to build *parameters* into the boundary value problems that are being solved and determine these parameter values as part of the solution process.

To USE THIS feature, we must take the following steps:

1. We include this parameter in our definition of the ODE function:

```
% add Q to my arguments for F
F = @(r,T,Q) [T(2); -(1./r)*T(2) - Q./(k*r)*exp(-r/R)];
```

2. The constraint on the parameter is added to the residual functions defined for the boundary conditions.

```
TgtTemp = 1500; % K
bcfun = @(Ta,Tb,Q) [Ta(2)+ Q2/k; ...
    Tb(2) + (h/k)*(Tb(1)-T_inf); ...
    Ta(1) - TgtTemp];
```

Note: Even though the function defined by bcfun does not make use of the third argument, Q , corresponding to the unknown parameter you must provide it as the third argument to our boundary condition function.

3. We also must supply an initial guess for the parameter when we initialize our solution.

```
rMin = R; rMax = R+w; nR = 200;
Tguess = [T_inf 0];
Qguess = 1e8;
solinit = bvpinit(linspace(rMin,rMax,nR),Tguess,Qguess);
```

4. The value for the parameter that meets our constraint is provided in the solution structure in the field named: *parameters*.

```
sol2 = bvp5c(F,bcfun,solinit);
fprintf('Q = %g \n',sol2.parameters);
```

The full script with these modifications is provided in the listing below. The resulting temperature profile is shown in Figure 160 and the Q satisfying this constraint is: $Q = 8.361 \times 10^7$.

```

R = 1.5e-2; % m, radius of fuel
w = 3.0e-3; % m, thickness of cladding
k = 16.75; % W/(m-K), thermal conductivity of clad
%Q = 1e8; % W/m^2, Source term from heat dep in cladding.
Q2 = 6.32e5; % W/m^2, Heat flux due to heat produced in fuel.
T_inf = 423; % K, temperature of water flowing on cladding
h = 1e4; % W/(m^2-K), convective heat transfer coefficient

TgtTemp = 1500; % K

% add Q to my arguments for F
F = @(r,T,Q) [T(2); -(1./r)*T(2) - Q./(k*r)*exp(-r/R)];

% ... and for my BC functions
bcfun = @(Ta,Tb,Q) [Ta(2)+ Q2/k; ...
    Tb(2) + (h/k)*(Tb(1)-T_inf); ...
    Ta(1) - TgtTemp];
rMin = R; rMax = R+w; nR = 200;
Tguess = [T_inf 0];

% ... and add this to solinit
Qguess = 1e8;
solinit = bvpinit(linspace(rMin,rMax,nR),Tguess,Qguess);

sol2 = bvp5c(F,bcfun,solinit);

figure(2)
plot(sol2.x,sol2.y(1,:),'-b','linewidth',2);
title('Fuel Clad Temperature','fontsize',14,...
    'fontweight','bold');
xlabel('R [m]','fontsize',12,'fontweight','bold');
ylabel('T(R) [K]','fontsize',12,'fontweight','bold');
grid on
set(gca,'fontsize',10,'fontweight','bold');

fprintf('Q = %g \n',sol2.parameters);

```

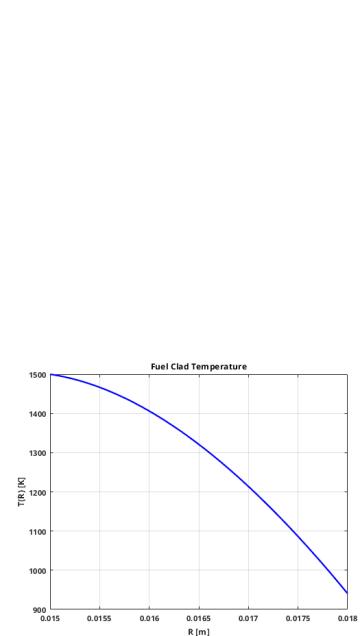


Figure 160: Temperature profile with constrained Q.

Lecture 31 - Solving BVPs with the Finite Difference Method

Objectives

The objectives of this lecture are to:

- Describe the Finite Difference Method (FDM) for BVPs.
- Demonstrate the method with an example problem.

Finite Difference Method for Boundary Value Problems

In the Finite Difference Method, the derivatives in the differential equation are replaced with *finite difference approximations*. The domain is divided into finite intervals defined by grid points as shown in Figure 161. The solution is approximated by the value of the dependent

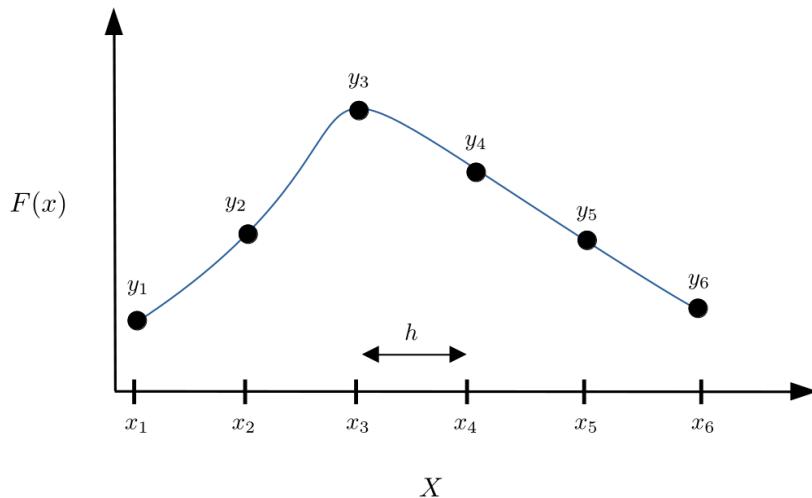


Figure 161: Function discretized into a finite number of intervals for the FDM.

variable at each grid point.

Consider the linear boundary value problem:

$$\text{Governing Equation: } y'' + f(x)y' + g(x)y = P(x), \quad a < x < b$$

$$\text{BCs: } y(a) = Y_a, \quad y(b) = Y_b$$

We will apply finite difference formulas with $\mathcal{O}(h^2)$ convergence for each grid point. As a reminder, finite difference formulas with this property are shown in Table 26 and Table 27.

CONSIDER EACH TERM of the governing equation. We will apply the appropriate finite difference formulas to create a linear system of equations.

1. First term:

$$y'' \approx \frac{1}{h^2} \begin{bmatrix} 2 & -5 & 4 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 4 & -5 & 2 \end{bmatrix} \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}}_{y}$$

2. Second term:

$$f(x)y' \approx \frac{1}{2h} \begin{bmatrix} f(x_1) & 0 & 0 & 0 & 0 & 0 \\ 0 & f(x_2) & 0 & 0 & 0 & 0 \\ 0 & 0 & f(x_3) & 0 & 0 & 0 \\ 0 & 0 & 0 & f(x_4) & 0 & 0 \\ 0 & 0 & 0 & 0 & f(x_5) & 0 \\ 0 & 0 & 0 & 0 & 0 & f(x_6) \end{bmatrix} \underbrace{\begin{bmatrix} -3 & 4 & -1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & -4 & 3 \end{bmatrix}}_{D_x} \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}}_y$$

Note that this second term is an approximation both because of the finite difference approximation for y' but also because we are representing the function $f(x)$ by its value at the grid points only. This will similarly be the case for the two remaining terms.

Grid Point	y'
left boundary	$\frac{-3y_i + 4y_{i+1} - y_{i+2}}{2h}$
interior point	$\frac{y_{i+1} - y_{i-1}}{2h}$
right boundary	$\frac{y_{i-2} - 4y_{i-1} + 3y_i}{2h}$

Table 26: Finite difference formulas for y' with $\mathcal{O}(h^2)$ convergence.

Grid Point	y''
left boundary	$\frac{2y_i - 5y_{i+1} + 4y_{i+2} - y_{i+3}}{h^2}$
interior point	$\frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$
right boundary	$\frac{-y_{i-3} + 4y_{i-2} - 5y_{i-1} + 2y_i}{h^2}$

Table 27: Finite difference formulas for y'' with $\mathcal{O}(h^2)$ convergence.

3. Third term:

$$g(x)y \approx \underbrace{\begin{bmatrix} g(x_1) & 0 & 0 & 0 & 0 & 0 \\ 0 & g(x_2) & 0 & 0 & 0 & 0 \\ 0 & 0 & g(x_3) & 0 & 0 & 0 \\ 0 & 0 & 0 & g(x_4) & 0 & 0 \\ 0 & 0 & 0 & 0 & g(x_5) & 0 \\ 0 & 0 & 0 & 0 & 0 & g(x_6) \end{bmatrix}}_G \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

4. Right-hand Side:

$$p(x) \approx \underbrace{\begin{bmatrix} p(x_1) \\ p(x_2) \\ p(x_3) \\ p(x_4) \\ p(x_5) \\ p(x_6) \end{bmatrix}}_P$$

Using the notation shown above, we can combine all of the terms into a linear system of equations:

$$\begin{aligned} y'' + f(x)y' + g(x)y &= p(x) \\ D_{xx}y + FD_xy + Gy &= P \\ [D_{xx} + FD_x + G]y &= P \\ Ly &= P \end{aligned}$$

THIS LINEAR SYSTEM of equations is a discrete representation of the differential equation. In order to find a unique solution to the boundary value problem, we must apply the boundary conditions.

Dirichlet Boundary Conditions

For this course, we will only consider Dirichlet (type-1) boundary conditions for problems that we solve with the finite difference

method. For this type of a boundary condition, our goal is to force the finite difference solution to be equal to the prescribed value at the boundaries. One simple way to do this is, as follows:

1. Replace the first row of L with all zeros except the first entry, which you will set to $L(1, 1) = 1$.
2. Replace the first entry on the right-hand side to Y_a .
3. Replace the last row of L with all zeros except the last entry, which you will set to $L(n, n) = 1$, where n is the number of equations in your system.¹
4. Replace the last entry on the right-hand side to Y_b .

The resulting system of equations is illustrated schematically in Figure 162.

Solving the System of Equations

How shall we solve this linear system of equations? Readers who have been following the book from the beginning should immediately know that the answer is either:

1. A direct method like Gauss elimination, LU decomposition, or whatever method `mldivide` chooses if you are working in a MATLAB environment.
2. An iterative method like Jacobi, Gauss-Seidel, successive-over-relaxation or MATLAB built-in methods like `pcg`, or `gmres`.

Two key points that may influence your decision that I want to highlight now are as follows:

1. *Most of the entries of L are zero.* For this small system it may not be very obvious. If we instead discretized the domain into, for example, ten thousand grid points, the structure of the matrices would be the same. Namely that, while each row would now have a total of 10^4 entries, only 3 or 4 of the entries in each row would be non-zero. If high accuracy is demanded users may be motivated to use 10^5 or 10^6 grid points. In such cases it is essential that sparse matrix data structures are used and algorithms that avoid fill-in are chosen for the solution process.²
2. *The matrix L is nearly symmetric.* Indeed, alternative methods for applying Dirichlet boundary conditions have been devised specifically to preserve symmetry of the linear system of equations. In this case, direct algorithms like `chol` or iterative methods like `pcg` that are designed for symmetric matrices may be used and generally result in obtaining a solution with less computational work.

¹ Of course, the number of equations, n , is equal to the number of grid points in your discrete representation of $y(x)$.

$$\left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ & \boxed{L(2 : 5, :)} & & & & \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right] \begin{matrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{matrix} = \begin{matrix} Y_a \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ Y_b \end{matrix}$$

Figure 162: Linear system after applying Dirichlet boundary conditions.

² Of course, for this example we have chosen finite difference formulas with $\mathcal{O}(h^2)$ convergence. If we had chosen formulas with higher-order convergence behavior, the sparsity of L would be reduced but, except in cases where high-order spectral methods are used (not discussed in this class), the matrix L remains sparse.

Constructing Sparse Differentiation Matrices

Before we move along to example problems, we will present functions that we will use to construct the 1st- and 2nd-order differentiation matrices. We will take advantage of MATLAB's tools for constructing sparse matrices.

A MATLAB function for constructing the 1st-order differentiation matrix is shown in the listing below:

```

function dx_sp = Dx(a,b,N)
% function dx_sp = Dx(a,b,N) returns a sparse matrix
% for a first order differentiation matrix using 2nd-order
% centered-difference for interior nodes and 2nd-order
% forward/backward-difference nodes for the respective
% endpoints of the domain.
%
% Inputs:
% a - scalar, left endpoint of domain
% b - scalar, right endpoint of domain
% N - number of points in the domain inclusive of the endpoints

% compute the number of entries in the sparse matrix:
% 3-each for the 2 endpoints + 2 each for the N-2 interior
% points
NumEntries = 3*2 + 2*(N-2);

% Initialize the sparse matrix data vectors
dx_row = nan(NumEntries,1); ❶
dx_col = nan(NumEntries,1);
dx_val = nan(NumEntries,1);

h = (b-a)/(N-1);

% first three entries for the left end-point
dx_row(1) = 1; dx_col(1) = 1; dx_val(1) = -3/(2*h);
dx_row(2) = 1; dx_col(2) = 2; dx_val(2) = 4/(2*h);
dx_row(3) = 1; dx_col(3) = 3; dx_val(3) = -1/(2*h);

ind = 4;

for i = 2:(N-1)
    dx_row(ind) = i; dx_col(ind) = i-1; dx_val(ind) = -1/(2*h);
    ind = ind+1;
    dx_row(ind) = i; dx_col(ind) = i+1; dx_val(ind) = 1/(2*h);
    ind = ind+1;
end

% last three entries for the right end-point
dx_row(ind) = N; dx_col(ind) = N; dx_val(ind) = 3/(2*h);
ind = ind+1;
dx_row(ind) = N; dx_col(ind) = N-1; dx_val(ind) = -4/(2*h);
ind = ind+1;
dx_row(ind) = N; dx_col(ind) = N-2; dx_val(ind) = 1/(2*h);

dx_sp = sparse(dx_row,dx_col,dx_val,N,N); ❷

end

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

Note: The discrete representations of the differential operators will be represented as sparse matrices. The idea is that, based on the chosen finite difference equations, we can determine in advance all of the non-zeros of the matrix. We will store the row and column number along with the value of all the non-zeros into vectors. Those vectors will then be supplied to a MATLAB built-in function to construct the sparse matrix.

❶ Per MATLAB style guides we pre-allocate space to store non-zero matrix element data. If N is large, this is much more efficient than having MATLAB "grow" the data vectors as we go.

❷ We provide the row and column number and the value for all non-zeros along with the overall dimensions of the array. See the MATLAB documentation for the sparse matrix constructor, `sparse`, for more details.

THE FUNCTION TO construct a 2nd-order differentiation matrix is similar and shown in the listing below:

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53

function dxx_sp = Dxx(a,b,N)
% function dxx_sp = Dxx(a,b,N) returns a sparse matrix
% for a second order differentiation matrix using 2nd-order
% centered-difference for interior nodes and 2nd-order
% forward/backward-difference nodes for the respective
% endpoints of the domain.
%
% Inputs:
% a - scalar, left endpoint of domain
% b - scalar, right endpoint of domain
% N - number of points in the domain inclusive of the endpoints

% compute the number of entries in the sparse matrix:
% 4-each for the 2 endpoints + 3 each for the N-2 interior
% points
NumEntries = 4*2 + 3*(N-2);

% Initialize the sparse matrix data vectors
dx_row = nan(NumEntries,1);
dx_col = nan(NumEntries,1);
dx_val = nan(NumEntries,1);

h = (b-a)/(N-1);

% first three entries for the left end-point
dx_row(1) = 1; dx_col(1) = 1; dx_val(1) = 2/(h^2);
dx_row(2) = 1; dx_col(2) = 2; dx_val(2) = -5/(h^2);
dx_row(3) = 1; dx_col(3) = 3; dx_val(3) = 4/(h^2);
dx_row(4) = 1; dx_col(4) = 4; dx_val(4) = -1/(h^2);

ind = 5;

for i = 2:(N-1)
    dx_row(ind) = i; dx_col(ind) = i-1; dx_val(ind) = 1/(h^2);
    ind = ind+1;
    dx_row(ind) = i; dx_col(ind) = i; dx_val(ind) = -2/(h^2);
    ind = ind+1;
    dx_row(ind) = i; dx_col(ind) = i+1; dx_val(ind) = 1/(h^2);
    ind = ind+1;
end

% last four entries for the right end-point
dx_row(ind) = N; dx_col(ind) = N; dx_val(ind) = 2/(h^2);
ind = ind+1;
dx_row(ind) = N; dx_col(ind) = N-1; dx_val(ind) = -5/(h^2);
ind = ind+1;
dx_row(ind) = N; dx_col(ind) = N-2; dx_val(ind) = 4/(h^2);
ind = ind+1;
dx_row(ind) = N; dx_col(ind) = N-3; dx_val(ind) = -1/(h^2);

dxx_sp = sparse(dx_row,dx_col,dx_val,N,N);
end

```

Example #1

Consider the following boundary value problem:

$$\text{Governing Equation: } y'' - 4y = 0, \quad 0 < x < 1$$

$$\text{BCs: } y(0) = 0, \quad y(1) = 5$$

The analytic solution to this BVP is: $y(x) = 5 \sinh(2x) / \sinh(2)$. In the listing below we present the MATLAB code to find an approximate solution using the finite difference method.

```

clear
clc
close 'all'

a = 0; b = 1;
N = 200;
x = linspace(a,b,N); x = x';

% known exact solution
y_exact = @(x) 5*sinh(2*x)./sinh(2);

Dxx_op = Dxx(a,b,N); % get 2nd order differentiation matrix
L = Dxx_op - 4*speye(N,N);% form the differential operator ①
rhs = zeros(N,1); % initialize the right hand side

% apply boundary conditions
L(1,:) = 0; L(1,1) = 1; rhs(1) = 0;
L(N,:) = 0; L(N,N) = 1; rhs(N) = 5;

y = L\rhs; % solve the system of equations

figure(1)
subplot(2,1,1)
xs = x(1:10:end);
plot(xs,y_exact(xs), 'sr', ...
      'x', 'c', 'linewidth', 3);
ylabel('Numeric Solution', 'fontweight', 'bold');
title('Solution to:  $y'''' - 4y = 0$ ;  $y(0)=0$ ;  $y(1)=5$ ', ...
      'fontsize', 14);
set(gca, 'fontweight', 'bold');
legend('Exact', 'Numeric', 'location', 'northwest');
grid on

subplot(2,1,2)
plot(x,abs(y - y_exact(x)), '-r', 'linewidth', 3);
ylabel('Numerical Error', 'fontweight', 'bold')
set(gca, 'fontweight', 'bold');
grid on

```

The solution and error are shown in Figure 163. Note that the error is (as expected?) zero at the boundaries since the boundary conditions are applied exactly at those locations. If we double the number of grid points, by what factor do we expect the solution to improve? Figure 164 shows that the maximum error is reduced by a factor of 4 which is what one should expect for a 2nd-order convergent approximation.

- ❶ The constructor `speye(N,N)` creates a sparse $N \times N$ identity matrix.

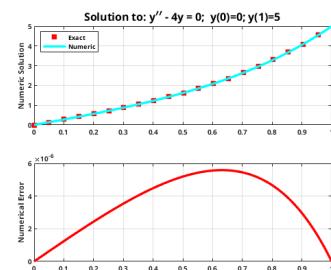


Figure 163: Finite difference method solution to Example #1 and point-wise error. $N = 200$

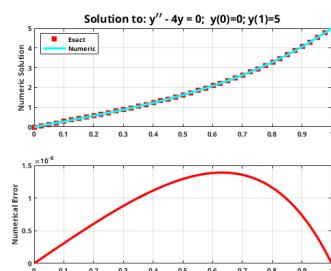


Figure 164: Finite difference method solution to Example #1 and point-wise error. $N = 400$

Example #2

As a somewhat more difficult example, consider the boundary value problem below:

$$\text{Governing Equation: } x^2y'' - 3xy' + 3y = 0, \quad 1 < x < 2$$

$$\text{BCs: } y(1) = 0, \quad y(2) = 0$$

The exact solution is: $y(x) = 12x - 15x^3 + 3x^5$.

The MATLAB code to solve this equation is presented in the listing below:

```

1 a = 1; b = 2;
2 N = 100;
3
4 Dxx_op = Dxx(a,b,N);
5 Dx_op = Dx(a,b,N);
6
7 x = linspace(a,b,N); x = x';
8
9 % known exact solution
10 y_exact = @(x) 12*x - 15*(x.^3) + 3*(x.^5);
11
12 L = (sparse(diag(x.^2)))*Dxx_op - ...
13     3*sparse(diag(x))*Dx_op + ...      ②
14     3*speye(N,N);
15 rhs = 24*(x.^5);
16
17 % apply BCs
18 L(1,:) = 0; L(1,1) = 1; rhs(1) = 0;
19 L(N,:) = 0; L(N,N) = 1; rhs(N) = 0;
20
21 % solve the system
22 y = L\rhs;
23
24 figure(1)
25 subplot(2,1,1)
26 xs = x(1:10:end);
27 plot(xs,y_exact(xs),'sr',...
28      'x,y,-c','linewidth',3);
29 title('Solution to: x^2y^{\\prime\\prime} - 3xy^{\\prime}+3y=24x^5;')
30 y(1)=y(2)=0',...
31 'fontweight','bold','fontsize',14);
32 ylabel('Numeric Solution','fontweight','bold');
33 set(gca,'fontweight','bold');
34 legend('Exact','Numeric');
35 grid on
36
37 subplot(2,1,2)
38 plot(x,abs(y - y_exact(x)),'-r','linewidth',3);
39 ylabel('Numerical Error')
40 set(gca,'fontweight','bold');

```

Note: This result was arrived at using the (very handy) method of “manufactured solutions.” This is where you devise of a solution first, $y_s(x)$, then plug it into the differential operator, $L(y_s)$. The result, $L(y_s) = p(x)$ becomes the source term and you evaluate y_s at the boundaries to get the Dirichlet boundary conditions. This is a well-accepted technique for validating your solver.

② Compare these lines of code to the governing equation. Does the syntax make sense? As an example, the nested functions `sparse(diag(x.^2))`, where x is a vector, results in a diagonal matrix with the values of x^2 along the diagonal stored in a sparse matrix representation.

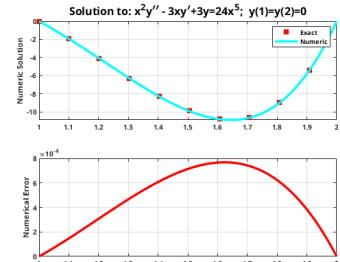


Figure 165: Finite difference method solution to Example #2 and point-wise error.

The solution and error are shown in Figure 165. Readers are encouraged to run this example and confirm 2nd-order convergence to the solution.

Lecture 32 - Solving Non-linear BVPs with Finite Difference Methods

Objectives

The objectives of this lecture are to:

- Describe a way to solve non-linear BVPs with Finite Difference Methods.
- Illustrate the method through an example.

A Non-Linear Boundary Value Problem

We will use the Finite Difference Method to solve the same problem that we tackled in Lecture 28. At risk of offending readers through gratuitous repetition, we will recall the problem statement here.

Problem Statement:

A pin fin is a slender extension attached to increase the surface area and enable greater heat transfer. When convection and radiation are included in the analysis, the steady-state temperature distribution, $T(x)$, along a pin fin can be calculated from the solution of the equation below:

$$\frac{d^2T}{dx^2} - \frac{h_c P}{kA_c} (T - T_s) - \frac{\epsilon\sigma_{SB} P}{kA_c} (T^4 - T_s^4) = 0, \quad 0 \leq x \leq 0.1 \quad (201)$$

with boundary conditions $T(0) = T_A$ and $T(0.1) = T_B$. A schematic of the system is shown in Figure 166. There are a number of parameters given in the equation. These are specified in Table 28.

WE WANT TO use the FDM which, as we saw last lecture, usually involves replacing the differential operators with their discrete finite difference equivalent. We then form a linear system of equations and solve it. The problem here is that the system of equations that we obtain is *non-linear*. The reason for this non-linearity is, as readers should know, the term involving T^4 .

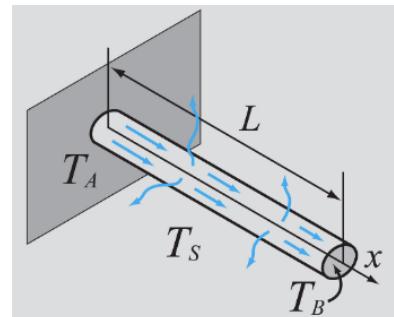


Figure 166: Pin Fin Boundary Value Problem Schematic.

Reminder: A differential equation is *non-linear* when the *dependent variable* or any of its derivatives appear in a non-linear term in the governing equation.

Parameter	Value
Convective heat transfer coefficient (h_c)	40 W/m ² -K
Perimeter of the pin (P)	0.016 m
Radiative emissivity of the surface (ϵ)	0.4
Thermal conductivity of the pin material (k)	240 W/m-K
Cross-sectional area of the fin (A_c)	1.6×10^{-5} m ²
Temperature of surrounding air (T_s)	293 K
Stefan-Boltzmann constant (σ_{SB})	5.67×10^{-8} W/m ² -K ⁴
Temperature of fin at base (T_A)	473 K
Temperature of fin at end (T_B)	293 K

Table 28: Example problem parameters.

We will address this problem with the following method:

1. **Step #1:** Re-arrange the equation to put non-homogeneous and non-linear terms on the right-hand side.

Carrying out this task for the governing equation for our example problem gives us:

$$\frac{d^2T}{dx^2} - \underbrace{\frac{h_c P}{k A_c}}_{\alpha_1} T = \underbrace{\frac{h_c P}{k A_c} T_s}_{\alpha_1} + \underbrace{\frac{\epsilon \sigma_{SB}}{k A_c} (T^4 - T_s^4)}_{\alpha_2}$$

2. **Step #2:** Apply finite difference approximations to linear terms and apply boundary conditions.

$$\begin{aligned} D_{xx} T - \alpha_1 [I] T &= \alpha_1 [I] T_s + \alpha_2 [I] (T^4 - T_s^4) \\ (\underbrace{D_{xx} - \alpha_1 [I]}_L) T &= \underbrace{(\alpha_1 [I] T_s - \alpha_2 [I] T_s^4)}_b + \alpha_2 [I] T^4 \\ LT &= b - \underbrace{\alpha_2 [I] T^4}_{\phi(T)} \end{aligned}$$

3. **Step #3:** Apply current (or initial) value of dependent variable to the non-linear term, $\phi(T)$.

$$LT = b - \phi(T_i)$$

4. **Step #4:** Solve the linear system of equations and evaluate the relative change in the dependent variable.

$$\begin{aligned} T_{i+1} &= L^{-1} [b - \phi(T_i)] \\ \epsilon &= \frac{||T_{i+1} - T_i||}{||T_i||} \end{aligned}$$

Note that the term b on the right-hand side is constant. The term $\phi(T)$ is the non-linear term in the dependent variable. L is the linear operator with appropriate modifications to incorporate the Dirichlet boundary conditions.

Note: Elements of T corresponding to boundaries are maintained constant and equal to the specified boundary conditions.

5. Repeat Steps #3 and #4 until $\epsilon < \text{TOL}$ for some specified tolerance.

This is a form of *fixed point iteration*. Looking again at the equation in Step #4 above:

$$T_{i+1} = L^{-1} [b - \phi(T_i)] = g(T_i)$$

The solution we are looking for is when $g(T_i) \rightarrow T_i$; this is called a *fixed point*. The conditions under which $g(T)$ will have a fixed point are beyond the scope of this class and will not be discussed further in this lecture.¹

MATLAB Implementation

A listing of MATLAB code that implements the method described above for our example problem is provided below. We begin by clearing the workspace and defining constants.

```

clear
clc
close 'all'

%% Define constants
a = 0; b = 0.1;
N = 2000;
Ac = 1.6e-5; % m^2, fin cross sectional area
P = 0.016; % m, perimeter of pin cross section
h_c = 40; % W/m^2-K, convective heat transfer coefficient
k = 250; % W/m-K, thermal conductivity of pin material
emiss = 0.5; % emissivity of pin material
sigma_sb = 5.67e-8; % W/m^2-K^4, Stefan-Boltzmann constant
Ts = 293; % K, temperature of surrounding air

% Boundary Conditions
Ta = 473; Tb = 293;

```

¹ Interested readers can learn more about the Banach fixed point theorem which says, roughly, that if $g(T)$ is a *contraction mapping* (at least in some region "close" to a fixed point) then $g(T)$ has a fixed point and the fixed point iteration will succeed. This is, it goes without saying, a loose statement.

Next we construct the differential operators and apply boundary conditions to them.

```

% Discretize the space and get Operators
x = linspace(a,b,N); x = x';
Dx_op = Dx(a,b,N);
Dxx_op = Dxx(a,b,N);
alpha_1 = h_c*P/(k*Ac);
alpha_2 = emiss*sigma_sb*P/(k*Ac);
L = Dxx_op - alpha_1*speye(N,N);

% apply boundary condition to L
L(1,:) = 0; L(1,1) = 1;
L(N,:) = 0; L(N,N) = 1;

```

In order to form the right-hand side of our system of equations, we need to estimate the temperature distribution. It's probably not fair since we have solved this problem before but, given the boundary conditions, a linear initial guess between the fin root and tip makes a lot of sense.

```
% Estimate initial temperature distribution
To = linspace(Ta,Tb,N); To = To';

% Form RHS
b = -ones(N,1)*Ts*alpha_1 - ones(N,1)*(Ts^4)*alpha_2;
phi = (To.^4)*alpha_2;
RHS = b - phi;

% apply BC to RHS
RHS(1) = Ta; RHS(N) = Tb;
```

31
32
33
34
35
36
37
38
39
40

Now we are ready to apply the fixed-point iteration scheme.

```
tol = 1e-7; imax = 100; T = To;

for i = 1:imax

    %solve the system of equations for new estimate of
    %temperature
    Tnew = L\RHS;
    % obtain convergence criterion "error estimate"
    Err = norm(T - Tnew, Inf)/norm(T, Inf);
    % exit loop if error is within tolerance
    if Err < tol
        fprintf('Success!! Iteration converged after %d
iterations.\n',i);
        fprintf('Error estimate: %g \n',Err);
        break;
    end
    % error tolerance not met, prepare for next iteration
    phi = (Tnew.^4)*alpha_2;
    RHS = b - phi;
    RHS(1) = Ta; RHS(N) = Tb; %re-apply BC
    T = Tnew;

end

if i == imax
    fprintf('Error! Solution not converged after %i iterations.\n
',imax);
    fprintf('Last residual = %g \n',Err);
end

fprintf('\n\n Plotting Last Solution: \n\n');
figure(1)
plot(x,T,'-r','linewidth',3);
title('Solution of Non-linear BVP with Finite Difference Method'
...
'fontsize',16,'fontweight','bold');
xlabel('X (m)', 'fontsize',14,'fontweight','bold');
ylabel('T (K)', 'fontsize',14,'fontweight','bold');
set(gca,'fontsize',12,'fontweight','bold');
grid on
```

42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77

The temperature distribution obtained using this method is shown in Figure 167. The result required 5 iterations with a relative error of 5.8×10^{-9} .

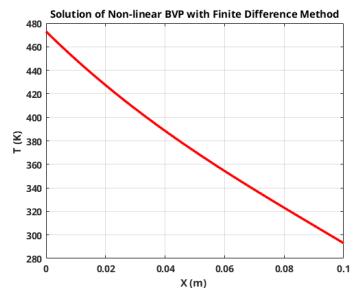


Figure 167: Solution of example problem with Finite Difference Methods

Lecture 33 - The Finite Element Method, Galerkin Method and Weak Form

Objectives

The objectives of this lecture are to:

- Describe the Method of Weighted Residuals to approximately solve a simple ODE.
- Motivate and derive a Weak Formulation of the example problem.

Example BVP

Consider the following boundary value problem:

$$\frac{d^2u}{dx^2} - u = -x, \quad 0 < x < 1$$

with homogeneous Dirichlet boundary conditions: $u(0) = u(1) = 0$.

THIS IS A second-order, linear, non-homogeneous, equation with constant coefficients. The complementary solution is: $u_c(x) = c_1 \cosh(x) + c_2 \sinh(x)$. Using the method of undetermined coefficients, one can show that a particular solution is: $u_p = x$. Combining the two gives us the general solution: $u(x) = u_c + u_p = x + c_1 \cosh(x) + c_2 \sinh(x)$. Applying boundary conditions gives us the solution:

$$u(x) = x - \frac{\sinh(x)}{\sinh(1)} \quad (202)$$

which is plotted in Figure 168.

Method of Weighted Residuals

In the past few lectures, we have explored a couple of different ways to approximately solve this problem. All of them, in one way or another, have been based on a discrete approximation of the differential

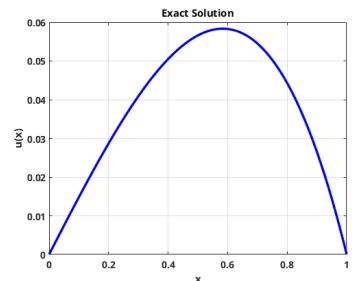


Figure 168: Exact solution of the example BVP.

operator. Today we will explore a method that takes a different approach.

Step #1: Select a *trial function* to represent the approximate solution; we will denote this as \tilde{u} . For this problem, we select the trial function given in Equation 203.

$$\tilde{u} = ax(1 - x) \quad (203)$$

where a is a yet-to-be-determined parameter. The exact form of the trial function is up to you as the modeler but there are some attributes that an effective trial function should have, specifically:

1. The trial function used in an n -th order equation should have n non-zero derivatives; and
2. the trial function should satisfy any Dirichlet boundary conditions.

Not coincidentally, the trial function given in Equation 203 satisfies those requirements for the given boundary value problem.

Step #2: Compute the residual. By *residual*, we mean that we arrange the governing equation so all of the terms are placed on one side and we insert our trial function as the tentative solution as shown below:

$$\begin{aligned} R &= \frac{d^2\tilde{u}}{dx^2} - \tilde{u} + x \\ &= \underbrace{-2a}_{d^2\tilde{u}/dx^2} - \underbrace{ax(1-x)}_{\tilde{u}} + x \end{aligned}$$

If we happened to pick the exact solution as the trial function, the residual would be zero. Since the exact solution involves the hyperbolic sine function, we cannot expect our 2nd-order polynomial trial function to yield a zero residual.¹

As ONE MIGHT EXPECT, we will want to minimize the residual and this begs the question of how, exactly, we are to go about measuring the size of the residual. This brings us to the next step in the process.

Step #3: Select a *test function*, w , to serve as a weight for the residual. The basic idea is that we will measure the size of the residual by taking the inner product of the residual and the weight function:

$$(w, R) = \int_a^b wR \, dx$$

Among the several choices for the weight function that, over time, have become standard, we will describe three:

Look again at the exact solution and mentally sketch a parabola to approximate \tilde{u} . The goal will be to pick a , which relates to the maximum height of the parabola, such that \tilde{u} is, in some sense, as close as possible to $u(x)$.

¹ Recall that $\sinh(x) = \frac{e^x - e^{-x}}{2}$ and that exponentials are, essentially, infinite-order polynomials. No finite-order polynomial can represent the exact solution.

1. Set the weight function equal to the derivative of the residual with respect to the unknown parameter a . Specifically: $w = \frac{dR}{da} = -2 - x(1 - x^2)$. This is equivalent to what we did back in Lecture 13 of this text when we used Least Squares curve fitting except, in this case, we only have one equation.
2. Use the Dirac delta function, $w = \delta(x - x_0)$. This is known as the *Collocation method*. From the properties of the Dirac delta function, $\int_a^b \delta(x - x_1) R(x) dx = R(x_1)$, where, $x_1 \in [a, b]$, is known as a collocation point.
3. Set the weight function equal to the derivative of the trial function with respect to the unknown parameter, a , so that: $w = \frac{d\tilde{u}}{da} = x(1 - x)$. This is called the Galerkin method.²

There are pros and cons to each of these approaches, a detailed discussion of which is beyond the scope of this text. For reasons that will be discussed later, we will choose to use the Galerkin method for this example.

Step #4: Minimize the weighted residual over the interval. In particular, we will solve for a such that:

$$(w, R) = \int_0^1 x(1-x) [-2a - ax(1-x) + x] dx = 0$$

In so doing, we will then, in a sense, have the value of a such that $\tilde{u}(x)$ best represents the exact solution. MATLAB code to carry out this task is shown in the listing below.

```

1 clear
2 clc
3 close 'all'

4 u_exact = @(x) x - sinh(x)/sinh(1);
5 u_trial = @(x,a) a.*x - a.*x.^2;
6 Resid = @(x,a) -2*a - a.*x + a.*x.^2 + x;
7 Weight_Galerkin = @(x) x.*x;
8
9 F = @(a) integral(@(x) Weight_Galerkin(x).*Resid(x,a),0,1);
10
11 a = fzero(F,1); ❶
12 fprintf('a = %12.11f\n',a);
13

```

For this example problem, $a = 0.227272727$ is the resulting parameter value. The approximate solution can be compared with the exact solution in Figure 169.

IN THE LIKELY event that one would like to obtain a more accurate approximation to the solution, the following approaches are available:

Note: As any pedant worth his pounds in salt will quickly point out, the “Dirac delta function” is not a function in the usual sense but technically is a *generalized function*. You should think of it in the same way you consider symbols such as ∞ to represent the somewhat abstract concept of infinity. Whereas one might consider ∞ to represent a number that can increase without bound so that it may be arbitrarily large, $\delta(x)$ is a “function” where $\delta(0) = \infty$ and $\delta(a) = 0$ if $a \neq 0$. The relevant property of $\delta(x)$ is when it is used in an integral where, if $a \leq x_1 \leq b$, then: $\int_a^b f(x)\delta(x - x_1) dx = f(x_1)$ if $x_1 \in [a, b]$. Effectively you *sample* the function at x_1 . If $x_1 \notin [a, b]$, then $\int_a^b f(x)\delta(x - x_1) dx = 0$.

² Named after Boris Galerkin who was a Russian engineer and mathematician. Interestingly, he wrote his first published paper while an inmate in the “Kresty” prison; a fact that might shame undergraduate students and junior tenure-track faculty members.

❶ Alternatively, you can use any other appropriate tool for solving a non-linear equation.

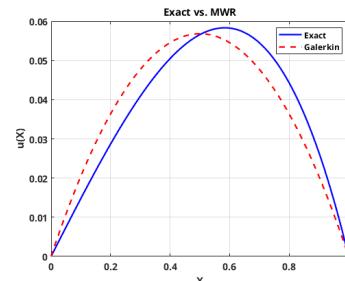


Figure 169: Approximate solution using the Galerkin method.

1. Choose a more complex trial and test function. For example, one might choose higher-order polynomials or even trigonometric functions. This is the approach taken in Chebyshev and Fourier spectral methods as described in, for example, the amazing textbook by Boyd.³ This is often referred to as p -refinement, where p is meant to represent the order of polynomial used for the trial and test functions.
2. Subdivide the domain with trial and test functions defined on each of the sub-domains. This approach is referred to as h -refinement. Finite Element Methods (FEM), that will be introduced in the next lecture, take this approach.
3. Do a combination of #1 and #2 above. This is sometimes called hp -refinement.

All of these approaches will be illustrated in the next lecture on FEM.

³ John P. Boyd. *Chebyshev and Fourier Spectral Methods*. Dover, Mineola, NY, Second edition, 2001

Weak Formulation

Before we finish this lecture, there is one more detail that I would like to address. There is at least one drawback to the method of weighted residuals as we have described it so far. That is that the trial function really needs to have two non-zero derivatives to be useful for the second-order differential operator. We could not, therefore, choose piece-wise linear functions for our trial or test functions if we pursued an h -refinement approach. The problem is that we *really want* to choose piece-wise linear functions. They're so simple to use and, with enough pieces, they can describe pretty much any function with arbitrary precision.

ON THE SURFACE it would seem that there is nothing to be done about our little problem. It is, after all, a second-order differential operator and we can't change that, right? Well, actually we can. That is what the weak form is all about. Let us re-visit the method of weighted residuals in, what we will now call, its "strong form":

$$\begin{aligned} \int_0^1 w \left(\frac{d^2 \tilde{u}}{dx^2} - \tilde{u} + x \right) dx &= 0 \\ \underbrace{\int_0^1 w \frac{d^2 \tilde{u}}{dx^2} dx}_{\text{integrate by parts}} - \int_0^1 w \tilde{u} dx + \int_0^1 w x dx &= 0 \\ w \frac{d \tilde{u}}{dx} \Big|_0^1 - \int_0^1 \frac{dw}{dx} \frac{d \tilde{u}}{dx} dx - \int_0^1 w \tilde{u} dx + \int_0^1 w x dx &= 0 \end{aligned}$$

Note: Notice that the first term on the left-hand-side of the last line involves derivative of the trial function at the boundary.

The last line is what we call the “weak form” of the minimization statement. Notice that now both the trial and test function need have only one non-zero derivative. We can solve this to find a . This process is shown in the MATLAB listing below.

```
%// Weak Form
W = @(x) x - x.^2;
Ut = @(x,a) a.*(x - x.^2);
dW_dx = @(x) 1 - 2*x;
dUt_dx = @(x,a) a.*(1-2*x);

Weak_Form = @(a) W(1)*dUt_dx(1,a) - W(0)*dUt_dx(0,a) - ...
    integral(@(x) dW_dx(x).*dUt_dx(x,a),0,1) - ...
    integral(@(x) W(x).*Ut(x,a),0,1) + ...
    integral(@(x) W(x).*x,0,1);

a_weak = fzero(Weak_Form,1);
```

1
2
3
4
5
6
7
8
9
10
11
12

The result is: $a_{\text{weak}} = 0.22727272727$, just as with the strong form.

THE FACT THAT the trial and test functions can now be piece-wise linear is not the only advantage of the weak form. Another advantage lies with the inclusion of the derivative of the trial function at the boundary in our formulation. Recall that we chose the trial function in part for the fact that it satisfied the Dirichlet boundary conditions. Satisfying the type-1 boundary condition is, in this sense, “essential” and, for some expositions on FEM, is why such boundary conditions are often referred to as *essential boundary conditions*. In the weak form, we also expose type-2 boundary conditions within the residual that we are minimizing. It is here where we now have the opportunity to specify a type-2 boundary condition directly within the formulation. In the literature, type-2 boundary conditions are often referred to as *natural boundary conditions* and it is in this sense that they appear “naturally” within the formulation.

Lecture 34 - Finite Element Method, Galerkin Method FEM in One Dimension

Objectives

The objectives of this lecture are to:

- Describe a Galerkin FEM for solving a simple BVP.
- Illustrate the method with a MATLAB implementation.
- Demonstrate hp -refinement and introduce the tools required to accomplish this task.

Review of the Problem

Recall the boundary value problem that we addressed in Lecture 33:

ODE: $\frac{d^2u}{dx^2} - u = -x, \quad 0 < x < 1$

BCs: $u(0) = u(1) = 0$

Analytic solution: $u(x) = x - \frac{\sinh(x)}{\sinh(1)}$

The strong form of the method of weighted residuals resulted in the following problem:

$$\begin{aligned}\int_0^1 wR \, dx &= \int_0^1 w \frac{d^2u}{dx^2} \, dx - \int_0^1 wu \, dx + \int_0^1 wx \, dx = 0 \\ &= \int_0^1 x(1-x) [-2a - ax(1-x) + x] \, dx = 0\end{aligned}$$

that we would solve for the unknown parameter a .

In order to address some technical deficiencies in the method as described in its strong form, we derived the weak form:

$$w \frac{d\tilde{u}}{dx} \Big|_0^1 - \int_0^1 \frac{dw}{dx} \frac{d\tilde{u}}{dx} \, dx - \int_0^1 w\tilde{u} \, dx + \int_0^1 wx \, dx = 0$$

which we would solve in a similar fashion but, in this case, with a wider range of suitable trial and test functions available—in particular piece-wise linear functions would work nicely—and with the helpful presence of the boundary term in the statement which is needed for handling type-2 boundary conditions.

In this lecture we seek to describe the Finite Element Method (FEM) as a procedure by which this weak form is solved to obtain a solution that is as accurate as we would like.

Galerkin Finite Element Method

In the finite element method, in addition to converting the strong form of the governing equation into the weak form of the minimum weighted residual statement, we will discretize the domain into a finite number of elements. For example, let us consider the one-dimensional domain divided into 3 elements, using piece-wise continuous trial functions and test functions as shown in Figure 170.

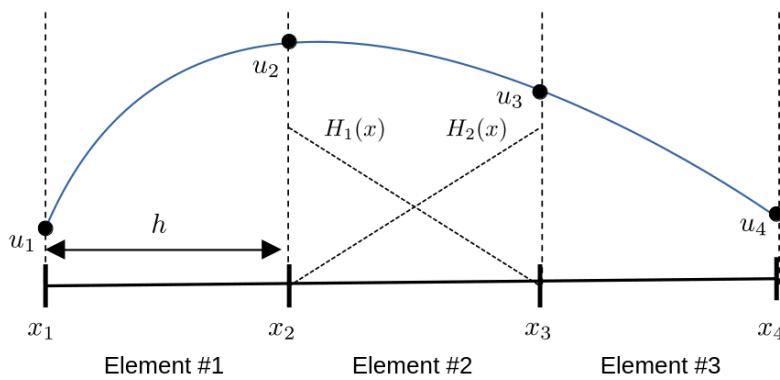


Figure 170: Discretization of problem domain with piece-wise linear elements.

Note: The shape functions, $H_1(x)$ and $H_2(x)$, are defined on each element—i.e. $H_1(x)$ and $H_2(x)$ on element #1 are distinct from $H_1(x)$ and $H_2(x)$ on the other elements. Nonetheless, the definitions are similar for each element, namely:

$$H_1(x) = \frac{x_{i+1}-x}{h_i}, \quad H_2(x) = \frac{x-x_i}{h_i} \text{ where } h_i = x_{i+1} - x_i.$$

$$\frac{dH_1}{dx} = -\frac{1}{h_i}, \quad \frac{dH_2}{dx} = \frac{1}{h_i}$$

Notice $H_1(x_i) = 1$ and $H_1(x_{i+1}) = 0$ while $H_2(x_i) = 0$ and $H_2(x_{i+1}) = 1$. This property is referred to as *cardinality* of the shape functions. Note also that $H_1(x) + H_2(x) = 1$ if $x \in [x_i, x_{i+1}]$. Lastly note that, in principle, the element size, h_i , can be different for each element. For this example, we will take the element size to be uniform.

Both the trial and test functions are constructed from $H_1(x)$ and $H_2(x)$ which are commonly referred to as *shape functions* that are defined on each element.

trial functions: $u(x) = H_1(x)u_i + H_2(x)u_{i+1}$

test functions: $w(x) = H_1(x) + H_2(x)$

So to recap and relate to the previous discussion about the method of minimum weighted residuals: we have a piece-wise linear trial and test functions; the unknown parameters are $\{u_1, u_2, u_3, u_4\}$; the test functions are “the same” as the trial functions insofar as that they are the same as the trial functions but without the unknown parameters. Our goal is to solve for the unknown parameters, $\{u_1, u_2, u_3, u_4\}$, so

that the weighted residual is minimized.

Now we will insert our newly defined trial functions and test functions into the weak form of the residual:

$$-\int_0^1 \frac{dw}{dx} \frac{du}{dx} dx - \int_0^1 wu dx + \int_0^1 wx dx = 0$$

Breaking this down one component at a time, on a per-element basis:

$$\begin{aligned} \frac{dw}{dx} \frac{du}{dx} &= \underbrace{\begin{bmatrix} \frac{dH_1}{dx} \\ \frac{dH_2}{dx} \end{bmatrix}}_{dw/dx} \underbrace{\begin{bmatrix} \frac{dH_1}{dx} & \frac{dH_2}{dx} \end{bmatrix}}_{du/dx} \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} = \begin{bmatrix} \left(\frac{dH_1}{dx} \frac{dH_1}{dx} \right) & \left(\frac{dH_1}{dx} \frac{dH_2}{dx} \right) \\ \left(\frac{dH_2}{dx} \frac{dH_1}{dx} \right) & \left(\frac{dH_2}{dx} \frac{dH_2}{dx} \right) \end{bmatrix} \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} = [K_1] \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} \\ wu &= \underbrace{\begin{bmatrix} H_1 \\ H_2 \end{bmatrix}}_w \underbrace{\begin{bmatrix} H_1 & H_2 \end{bmatrix}}_u \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} = \begin{bmatrix} (H_1 H_1) & (H_1 H_2) \\ (H_2 H_1) & (H_2 H_2) \end{bmatrix} \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} = [K_2] \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} \\ xw &= \begin{bmatrix} x_i & 0 \\ 0 & x_{i+1} \end{bmatrix} \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} = r \end{aligned}$$

where, readers should bear in mind, the matrices $[K_1]$, $[K_2]$, and the vector r are all functions of x . Putting this all together for all 3 elements gives us:

$$\sum_{i=1}^3 \left[- \int_{x_i}^{x_{i+1}} [K_1] \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} dx - \int_{x_i}^{x_{i+1}} [K_2] \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} dx + \int_{x_i}^{x_{i+1}} r dx \right] = 0$$

We also need to deal with the integrals. For some problems and element types, this integration can be done analytically, but the overwhelmingly common process is to carry out the integration numerically using Gauss quadrature. For bi-linear shape functions as is used in this example, two quadrature points will be used on each element. As always, when performing Gauss quadrature, the interval of integration needs to be mapped to $[-1, 1]$ so the Gauss points are mapped accordingly and, for each element, a Jacobian is calculated and incorporated into the integration.

Note: We have left off the boundary term. When we eventually apply boundary conditions, we will force both the trial and test function to be equal to zero on the boundary. Thus:

$$w \frac{du}{dx} \Big|_0^1 = \cancel{w(1) \frac{du}{dx}} \Big|_{x=1}^0 - \cancel{w(0) \frac{du}{dx}} \Big|_{x=0}^0 = 0.$$

Note: the index i corresponds to each of the 3 elements.

$$\sum_{i=1}^3 (\text{Jac})_i \left[\sum_{q=1}^{qp} (\text{wgt})_q \left\{ - [K_1(x_q) + K_2(x_q)] \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} + r(x_q) \right\} \right] = 0$$

Assembly into a Linear System

Once the quadrature has been completed, the matrices K_1 , K_2 and the vector r calculated on each element needs to be assembled into the system matrix. We say “assembly” but this is merely the natural result of carrying out the summation of the integrals across all elements. The output of the process is depicted in Figure 171. The

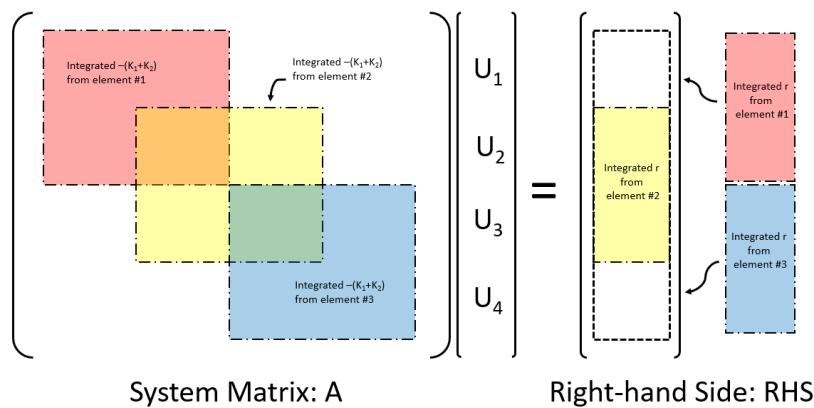


Figure 171: Schematic of system assembly.

matrices K_1 and K_2 of element #1 contribute to the first two rows and columns of the system matrix; the matrices from element #2 contribute to the second and third rows and columns; the matrices from element #3 to the third and fourth rows and columns. For entries where there is overlap, the results are added. The same process is carried out for the vector r that is placed on the right-hand-side.

ONCE THE SYSTEM matrix is assembled, all that remains is to apply the Dirichlet boundary conditions and solve by some convenient method. The resulting coefficient values: $\{U_1, U_2, U_3, U_4\}$ are the parameters for our approximate solution.

MATLAB Implementation

Code for an example MATLAB implementation of the Finite Element Method with linear shape functions is given below. We start by cleaning out the workspace and initializing a representation of the exact solution.

```
clear
clc
close 'all'

%% Exact Solution
u_exact = @(x) x - sinh(x). / sinh(1); ❶
```

❶ Always, always, always solve a known problem when learning and testing a new method.

```
%>% Finite Element Parameters and Data Structures
a = 0; b = 1;
Ua = 0; Ub = 0; % boundary conditions

nelem = 3; % specify # of elements
gcoord = linspace(a,b,nelem+1); % x-coordinates of all nodes
nnodes = length(gcoord); %number of nodes
nodes = nan(nelem,2);    ②
nodes(:,1) = 1:nelem;% x_i node number
nodes(:,2) = 2:(nelem+1);%x_i+1 node number

```

7
8
9
10
11
12
13
14
15
16

% sample points for Gauss Quadrature
q = [-0.57735027; 0.57735027];
w = [1; 1]; % weights ③
nqp = length(q); % number of quadrature points

% initialize global arrays
K1 = zeros(nnodes,nnodes);
K2 = zeros(nnodes,nnodes); ④
R = zeros(nnodes,1);

for ele = 1:nelem

% local arrays to be populated
k1 = zeros(2,2);
k2 = zeros(2,2); ⑤
r = zeros(2,1);

% local mapping for GQ
aL = gcoord(nodes(ele,1));
bL = gcoord(nodes(ele,2));
xT = @(t) ((bL - aL)*t + aL + bL)/2; ⑥
Jac = (bL - aL)/2;

% shape functions for this element
H = cell(2,1); ⑦
hi = bL - aL;
H{1} = @(x) (bL-x)/hi;
H{2} = @(x) (x-aL)/hi;

Hp = cell(2,1);
% for generality, use functions
Hp{1} = @(x) -1/hi;
Hp{2} = @(x) 1/hi;

for qp = 1:nqp
% sum weighted contribution at Gauss Points
k1(1,1) = k1(1,1) + ...
 Hp{1}(xT(q(qp)))*Hp{1}(xT(q(qp)))*w(qp);
k1(1,2) = k1(1,2) + ...
 Hp{1}(xT(q(qp)))*Hp{2}(xT(q(qp)))*w(qp);
k1(2,1) = k1(2,1) + ...
 Hp{2}(xT(q(qp)))*Hp{1}(xT(q(qp)))*w(qp);
k1(2,2) = k1(2,2) + ...
 Hp{2}(xT(q(qp)))*Hp{2}(xT(q(qp)))*w(qp); ⑧

k2(1,1) = k2(1,1) + ...
 H{1}(xT(q(qp)))*H{1}(xT(q(qp)))*w(qp);
k2(1,2) = k2(1,2) + ...
 H{1}(xT(q(qp)))*H{2}(xT(q(qp)))*w(qp);

67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165

② The nodes array serves as a map between global and local node numbers. Each row contains the global node numbers of the local nodes of each element. The column numbers correspond to the local node numbers.

③ It should seem odd that we would hard-code in the quadrature points and weights for the integration scheme. We will soon generalize this so that the order of integration can be selected at runtime.

④ We will initialize these arrays now so they are available for the assembly process.

⑤ These are the elemental sub-arrays. Alternatively, you can write your code so that these elemental values are stored directly in the system array.

⑥ The nodes array is used to get the coordinates of the local nodes for the current element. This allows us to map the quadrature points to the correct physical location with each element.

⑦ Cell arrays are used to store the shape functions, $H(x)$, and the derivatives of shape functions, dH/dx , since cell arrays allow the array members to be function handles.

⑧ Here we accumulate the weighted contribution from each quadrature point for the K_1 and K_2 matrices and the r vector.

```

k2(2,1) = k2(2,1) + ...
H{2}(xT(q(qp)))*H{1}(xT(q(qp)))*w(qp);
k2(2,2) = k2(2,2) + ...
H{2}(xT(q(qp)))*H{2}(xT(q(qp)))*w(qp);

r(1) = r(1) + xT(q(qp))*H{1}(xT(q(qp)))*w(qp);
r(2) = r(2) + xT(q(qp))*H{2}(xT(q(qp)))*w(qp);

end % qp
% apply Jacobian
k1 = k1*Jac; k2 = k2*Jac; r = r*Jac; ⑨

```

⑨ Finally the Jacobian is applied to complete the numerical quadrature.

```

% add local arrays to global arrays ("assembly")
for i = 1:2
    for j = 1:2
        row = nodes(ele,i); col = nodes(ele,j); ⑩
        K1(row,col) = K1(row,col) + k1(i,j);
        K2(row,col) = K2(row,col) + k2(i,j);
    end % j
    dof = nodes(ele,i);
    R(dof) = R(dof) + r(i);
end % i
end % numel

```

⑩ Here again the nodes array is used as a map to facilitate the global matrix assembly process.

Now that the integration and assembly processes are complete to construct the system matrix, we are ready to apply the Dirichlet boundary conditions, solve for the unknown parameters, and plot the solution. For three elements with linear shape functions, the approximate solution is shown in Figure 172.

```

% Gather into system matrix and vector
A = -K1 - K2;
RHS = -R;

% apply boundary conditions
A(1,:) = 0; A(1,1) = 1; RHS(1) = Ua;
A(nnodes,:) = 0; A(nnodes,nnodes) = 1; RHS(nnodes) = Ub;

% solve the system of equations
u = A\RHS;

%% Plot the solution and check error
x = gcoord;
x_gold = linspace(a,b,1000);
figure(1)
plot(x,u, '-b', ...
    x_gold, u_exact(x_gold), '--r', ...
    'LineWidth', 2);
title('Finite Element Solution',...
    'FontSize', 14, 'FontWeight', 'bold');
xlabel('X', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('U(X)', 'FontSize', 12, 'FontWeight', 'bold');
grid on
legend('U - FEM', 'U - Exact', 'Location', 'best');

```

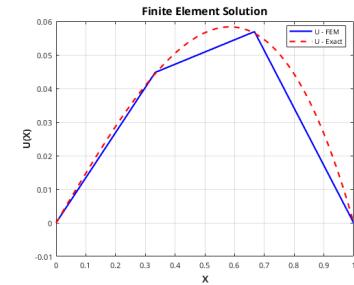


Figure 172: Approximate solution with three linear elements.

Of course we would not normally just use three elements; if we increase the number of finite elements to 30, the solution is much improved. This solution along with a plot of absolute error is shown

in Figure 173. If greater precision is needed, we can simply increase the number of elements.

FEM with P-refinement

So far we have not taken advantage of the generality of the FEM approach to finding approximate solutions to boundary value problems. We have presented the mathematical analysis and an accompanying MATLAB script that solves one problem with FEM using linear shape functions on an arbitrary number of elements. This allowed us to carry out h-refinement to improve our solution. In the section we will slightly generalize the structure of the MATLAB code and incorporate the ability to increase the polynomial order of the shape functions used within each element. A few notes before we begin to present the code:

1. The basic structure of the MATLAB implementation is the same. The main difference is that the shape functions and the derivative of the shape functions will be created at “run time” based on input from the script.
2. Higher order shape functions will mean that the calculations for populating the local arrays K_1 , K_2 and r will be more lengthy but, it turns out, that we will accomplish this task with *less code* by generalizing the local element matrix calculation.

Now for the code:

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
clear
clc
close 'all'

%% Exact Solution
u_exact = @(x) x - sinh(x). / sinh(1);

%% Finite Element Parameters and Data Structures
a = 0; b = 1;
Ua = 0; Ub = 0; % boundary conditions
nelem = 3; % select # of elements ❶
order = 1;

[gcoord, nodes] = genMesh1D(a, b, nelem, order); ❷

% nldofs = number of local dofs for each element
nldofs = order+1;

% global x-coordinate of each node
nnodes = length(gcoord);

% sample points for Gauss Quadrature
nqp = order+1; % number of sample points requested
[q,w] = getGPandWeights(nqp); ❸

```

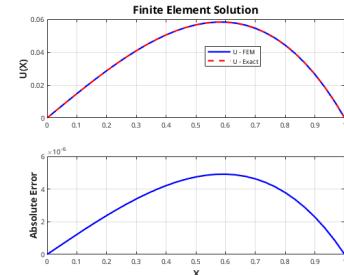


Figure 173: FEM solution with 30 elements.

❶ Notice that we are, again, going to solve a problem that we've done before. Here we will confirm that we get the same result that we did with the less generalized implementation.

❷ This local function will be presented later but you should note right away that the *outputs* from `genMesh1D`, the `gcoord` and `nodes` arrays fulfill the same purpose as they did in the previous code; provide the x-coordinate for all nodes and provide the local-to-global node number mapping for each element. It is a good idea to establish standardized data structures when implementing FEM codes and this is intended to be an example of this type of standardization.

❸ As the function name suggests, we are going to get the sample points and weights for the requested Gauss quadrature scheme. This is an alternative to hard-coding the sample points and weights like was done for the previous example.

```
% initialize global arrays
K1 = zeros(nnodes,nnodes);
K2 = zeros(nnodes,nnodes);
R = zeros(nnodes,1);
```

27
28
29
30

The next task will be to carry out the numeric integration and element assembly process. Since we allow the user to specify the order of interpolation, we will need to generate appropriate shape functions and their derivatives. Leveraging the work we did earlier in the course we will use Lagrange polynomials (Lecture 15) for the shape functions and the derivatives of those Lagrange polynomials (Lecture 18).

```
% carry out assembly process
for ele = 1:nelem
    % local arrays to be populated
    k1 = zeros(nldofs,nldofs); ④
    k2 = zeros(nldofs,nldofs);
    r = zeros(nldofs,1);

    % local mapping for CQ
    aL = gcoord(nodes(ele,1));
    bL = gcoord(nodes(ele,nldofs));
    xT = @(t) ((bL - aL)*t + aL + bL)/2;
    Jac = (bL - aL)/2;

    % Get sample points for shape functions
    xgl = gcoord(nodes(ele,:));

    % get Lagrange Interpolant of requested order
    H = getLagrangeInterp(xgl); ⑤
    Hp = getLagrangeInterpDeriv(xgl);

    for qp = 1:nqp
        % sum weighted contribution at Gauss Points
        for i = 1:nldofs
            for j = 1:nldofs
                k1(i,j) = k1(i,j) + ...
                           Hp{i}(xT(q(qp)))*Hp{j}(xT(q(qp)))*w(qp); ⑥
                k2(i,j) = k2(i,j) + ...
                           H{i}(xT(q(qp)))*H{j}(xT(q(qp)))*w(qp);
            end % j
            r(i) = r(i) + xT(q(qp))*H{i}(xT(q(qp)))*w(qp);
        end % i
    end % qp
    % apply Jacobian to map to physical coordinates
    k1 = k1*Jac; k2 = k2*Jac; r = r*Jac;

    % add local arrays to global arrays "assembly"
    for i = 1:nldofs
        for j = 1:nldofs
            row = nodes(ele,i); col = nodes(ele,j);
            K1(row,col) = K1(row,col) + k1(i,j);
            K2(row,col) = K2(row,col) + k2(i,j);
        end % j
        dof = nodes(ele,i);
        R(dof) = R(dof) + r(i);
    end % i
end % numel
```

31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76

④ These arrays are the same as in the last example except that they must be sized according to the number of local degrees of freedom, nldofs. In the technical parlance, each unknown is often referred to as a “degree of freedom.”

⑤ These local functions will be presented separately.

⑥ This is essentially the same as in the last example except that now loops are used so that the arbitrary number of local degrees of freedom can be handled seamlessly.

```

% apply boundary conditions
A = -K1 - K2;
RHS = -R;
A(1,:) = 0; A(1,1) = 1; RHS(1) = Ua;
A(nnodes,:) = 0; A(nnodes,nnodes) = 1; RHS(nnodes) = Ub;

% solve the system of equations
u = A\RHS;

```

Some solutions are shown in Figure 174 and Figure 175. This shows the benefit of p-refinement. Similar precision can be obtained with lower-order shape functions if the number of elements is increased, in hp-refinement.

Local Functions

The several local functions that were used for the generalized MATLAB implementation are shown in this section.

genMesh1D

```

function [gcoord, nodes] = genMesh1D(a,b,nelem,order)
%generates the global coordinates of all mesh points and
%provides a mapping between global node number and local
%element dof for all elements.

ldofs = order + 1; % need order+1 local degrees of freedom
ndofs = nelem+1 + (ldofs - 2)*nelem;
gcoord = nan(1,ndofs);
nodes = nan(nelem,ldofs);
ele_boundaries = linspace(a,b,nelem+1);
dof_it = 1;
for e = 1:nelem
    aL = ele_boundaries(e);
    bL = ele_boundaries(e+1);
    xT = @(t) ((bL - aL)*t + aL + bL)/2;
    [xgl,~]=legendre_gauss_lobatto(ldofs);
    xgl = xT(xgl); % translate to current element

    for ld = 1:ldofs
        nodes(e,ld) = dof_it;
        gcoord(dof_it) = xgl(ld);
        dof_it = dof_it + 1;
    end % ld
    % set back for shared node at element boundary
    dof_it = dof_it - 1;
end % e

end

```

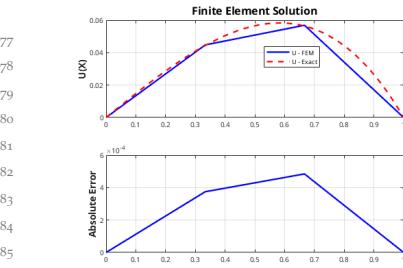


Figure 174: FEM Solution for 3 linear elements, revisited.

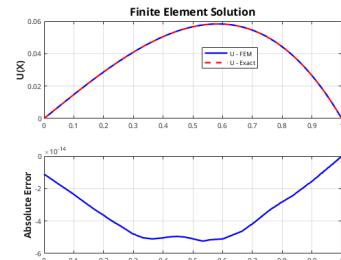


Figure 175: FEM Solution for 3 elements with 9th-order shape functions.

getGPandWeights This function, as the name indicates, gets the Gauss points and weights for the specified number of points. This function implements the theory described in Lecture 20 for Gauss quadrature.

```

function [xgl, wgl] = getGPandWeights(P)
%GetGP(P) Get Gauss Points and Weights for P-point Gauss Quad
% input: P - # of Gauss Points to use in integration
% output: xgl - vector of gauss points
%           wgl - vector of weights
%
%
%% Generate Legendre Polynomials of Order o through P
% Store handles to these functions in a cell array.
isQuadSet = false; % flag for special exit of quadrature scheme.
Pn = cell(P+1,1);
Pn{1} = @(x) 1;
Pn{2} = @(x) x;
if P == 0
    error('P must be greater than 0');
elseif P == 1
    xgl = 0; % for P == 1, GQ reduces to midpoint rule
    wgl = 2;
    isQuadSet = true;
else
    for n = 2:P
        % use recurrence relation to generate higher order
        % Legendre Polynomials ("Pn functions")
        Pn{n+1} = @(x) ...
            (2*(n-1)+1)*x.*Pn{n}(x)./((n-1)+1) ...
            - (n-1)*Pn{n-1}(x)./((n-1)+1);
    end
end

if ~isQuadSet
    %% Compute Roots to the Pth order Legendre Polynomial
    % get an approximate zeros from the Chebychev points
    Tch = @(n) cos((2*(1:n)) - 1*pi/(2*n));
    xEst = Tch(P);

    % use fzero and approximate root to find root of the Pn
    % polynomial.
    xgl = NaN(1,P);
    for r = 1:P
        xgl(r) = fzero(Pn{P+1},xEst(r));
    end

    %% Sample Pn functions at roots of Pth order function
    % These values form the matrix that will be used to
    % find the weights for the quadrature method.
    if P == 1
        A = xgl(1);
    else
        A = NaN(P,P);
        A(1,:) = Pn{1}(xgl);
        A(2,:) = Pn{2}(xgl);
        for n = 2:(P-1)
            A((n+1),:) = Pn{n+1}(xgl);
        end
    end

    %% Form LHS vector
    % These are equal to the integral of the lower order Pn
    % functions over the domain. For P=0, the integral

```

```
% equals 2; for all other orders, the integral is zero.
k = zeros(P,1); k(1) = 2;

%% Solve for the weights
wgl = A\k;
end
end
```

174
175
176
177
178
179
180

genLagrangeInterp

```
function H = getLagrangeInterp(Xi)
%function dH = getLagrangeInterp(Xi)
% input Xi - vector of sample points
% output H - cell array containing Lagrange Interp Functions

n = length(Xi);
H = cell(n,1);

for i = 1:n
    L = @(x) 1; %<-- initialize the Lagrange Function
    for j = 1:n
        if j ~= i
            L = @(x) L(x) .*((x - Xi(j))./(Xi(i) - Xi(j)));
        end
    end
    H{i} = L;
end
end
```

181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198

genLagrangeInterpDeriv

```
function dH = getLagrangeInterpDeriv(Xi)
%function dH = getLagrangeInterpDerive(Xi)
% input: Xi - vector of sample points
% output dH - cell array containing the derivative
% of Lagrange Interpolant functions.
n = length(Xi);
dH = cell(n,1);

for i = 1:n
    dLi = @(x) 0;
    for k = 1:n
        if k ~= i
            dLp = @(x) 1;
            for j = 1:n
                if ((j ~= i) && (j ~= k))
                    dLp = @(x) dLp(x) .*...
                        ((x - Xi(j))./(Xi(i) - Xi(j)));
                end
            end
            dLi = @(x) dLi(x) + ...
                ((1./((Xi(i) - Xi(k))).*dLp(x));
        end
    end
    dH{i} = dLi;
end
end
```

199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225

legendre_poly

```

function [Lo,Lo_1,Lo_2] = legendre_poly(p,x)
%
%This code computes the Legendre Polynomials and
%its 1st and 2nd derivatives
%Written by F.X. Giraldo on 4/2008
%          Department of Applied Mathematics
%          Naval Postgraduate School
%          Monterey, CA 93943-5216
%
L1=0;L1_1=0;L1_2=0;
Lo=1;Lo_1=0;Lo_2=0;

for i=1:p
    L2=L1;L2_1=L1_1;L2_2=L1_2;
    L1=Lo;L1_1=Lo_1;L1_2=Lo_2;
    a=(2*i-1)/i;
    b=(i-1)/i;
    Lo=a*x*L1 - b*L2;
    Lo_1=a*(L1 + x*L1_1) - b*L2_1;
    Lo_2=a*(2*L1_1 + x*L1_2) - b*L2_2;
end
end

```

226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247

legendre_gauss_lobatto

```

function [xgl,wgl] = legendre_gauss_lobatto(P)
%
%This code computes the Legendre-Gauss-Lobatto points
%and weights which are the roots of the Lobatto Polynomials.
%Written by F.X. Giraldo on 4/2008
%          Department of Applied Mathematics
%          Naval Postgraduate School
%          Monterey, CA 93943-5216
%
p=P-1; %Order of the Polynomials
ph=floor( (p+1)/2 );
xgl = nan(1,P);
wgl = nan(1,P);
for i=1:ph
    % estimate the roots
    x=cos( (2*i-1)*pi/(2*p+1) );
    for k=1:20
        % evaluate L, dL, ddL
        [Lo,Lo_1,Lo_2]=legendre_poly(p,x);
        %Compute Nth order Derivatives of Legendre Polys
        %
        %Get new Newton Iteration
        dx=-(1-x^2)*Lo_1/(-2*x*Lo_1 + (1-x^2)*Lo_2);
        x=x+dx;
        if (abs(dx) < 1.0e-20)
            break
        end
    end
    xgl(p+2-i)=x;
    wgl(p+2-i)=2/(p*(p+1)*Lo^2);
end

```

248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280

```

%Check for Zero Root
if (p+1 ~=~ 2*ph)
    x=0;
    [Lo,~,~]=legendre_poly(p,x);
    xgl(ph+1)=x;
    wgl(ph+1)=2/(p*(p+1)*Lo^2);
end
281
282
283
284
285
286
287
288
289
290
291
292
293
294

%Find remainder of roots via symmetry
for i=1:ph
    xgl(i)=-xgl(p+2-i);
    wgl(i)=+wgl(p+2-i);
end
end

```


Part XIII

Back Matter

Bibliography

Witch of Agnesi. URL https://en.wikipedia.org/wiki/Witch_of_Agnesi.

Approximation Theory and Approximation Practice. Society of Industrial and Applied Mathematics, Philadelphia, PA, 2013.

OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy*, 82:90–97, 2015.

Przemyslaw Bogacki and Lawrence F Shampine. A 3 (2) pair of runge-kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989.

Frank Bowman. *Introduction to Bessel functions*. Courier Corporation, 2012.

John P. Boyd. *Chebyshev and Fourier Spectral Methods*. Dover, Mineola, NY, Second edition, 2001.

John Charles Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2016.

James W Demmel. *Applied numerical linear algebra*. SIAM, 1997.

William L Dunn and J Kenneth Shultis. *Exploring Monte Carlo Methods*. Elsevier, 2022.

John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*, 2020. URL <https://www.gnu.org/software/octave/doc/v5.2.0/>.

Alexander D. Lindsay et al. 2.0 - MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX*, 20:101202, 2022. ISSN 2352-7110.

Balay et al. PETSc/TAO Users Manual. Technical Report ANL-21/39 - Revision 3.19, Argonne National Laboratory, 2023.

Stanley J Farlow. *Partial Differential Equations for Scientists and Engineers*. Courier Corporation, 1993.

George Elmer Forsythe, Cleve B Moler, and Michael A Malcolm. *Computer methods for mathematical computations*. Prentice-Hall, 1977.

Walter Gander. On Halley's iteration method. *The American Mathematical Monthly*, 92(2):131–134, 1985.

Gilat, Amos and Subramaniam, Vish. *Numerical Methods for Engineers and Scientists: an Introduction with Applications Using MATLAB*. Wiley, Hoboken,NJ, third edition, 2014.

John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM journal on matrix analysis and applications*, 13(1):333–356, 1992.

A. Henderson. ParaView Guide, A Parallel Visualization Application. Technical report, Kitware Inc., 2007.

Nick Higham. What is the Matrix Inverse? URL <https://nhigham.com/2022/03/28/what-is-the-matrix-inverse/>. Accessed on July 19, 2023.

Benjamin S Kirk, John W Peterson, Roy H Stogner, and Graham F Carey. libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22: 237–254, 2006.

Kwon, Young W and Bang, Hyochoong. *The Finite Element Method using MATLAB*. CRC press, 2018.

Cleve Moler. Floating Point Numbers, 2014. URL <https://blogs.mathworks.com/cleve/2014/07/07/floating-point-numbers/>. Accessed on July 12, 2023.

National Institute of Standards and Technology. Matrix Market. URL <https://math.nist.gov/MatrixMarket/>. Accessed on July 18, 2023.

William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

Timothy Sauer. *Numerical analysis*. Addison-Wesley Publishing Company, 2011.

Lawrence F Shampine. Vectorized adaptive quadrature in MATLAB. *Journal of Computational and Applied Mathematics*, 211(2):131–140, 2008.

Jonathan Richard Shewchuk. Conjugate gradient method without the agonizing pain. *School of Computer Science, Carnegie Mellon University Pittsburgh, 1994.*

GW Stewart. *Afternotes on Numerical Analysis*. University of Maryland at College Park, 1993.

Inc. The Math Works. Matlab, v2022a, 2022. URL <https://www.mathworks.com/>.

Top500. The Top 500 List. URL <https://top500.org/>. Accessed on July 19, 2023.

Lloyd N Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 2022.

University of Tennessee, Knoxville. HPL Benchmark. URL <https://icl.utk.edu/hpl/index.html>. Accessed on July 19, 2023.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Dennis G Zill. *Advanced Engineering Mathematics*. Jones & Bartlett Learning, 2020.

Appendices

Matlab Style Rules

1. **Rule:** All scripts will start with the commands: `clear`, `clc`, and `close 'all'`

Rationale: No script should depend upon any data visible in the MATLAB workspace when the script starts. By omitting these commands, residual data within the workspace may hide errors.

2. **Rule:** Your code must be documented with enough details such that a reader unfamiliar with your work will know what you are doing.

Rationale: Code documentation is a habit. For more significant projects readers may need help in deciding what the author of the code intended. For your own code, the most likely reader is you—a few months into the future.

3. **Rule:** Function and variable names must be meaningful and reasonable in length.

Rationale: Failing to do either make code harder to read and maintain.

4. **Rule:** All outputs from the code must be meaningful; numbers should be formatted, part of a sentence, and include units. Graphs should be readable and axis labels should make sense and include units.

Rationale: Code output is a form of communication. It is important that this communication be clear and unambiguous.

5. **Rule:** Do not leave warnings from the Code Analyzer undressed.

Rationale: Sometimes Code Analyzer warnings can be safely ignored. Most of the time the warning points to a stylistic error that would be unacceptable in software that you use. Occasionally these warnings are indicative of a hidden error.

6. **Rule:** Use the “smart indentation tool” to format the indentation of your code.

Rationale: This tool improves code readability. It will also occasionally point out errors that you did not see before.

7. **Rule:** Pre-allocate arrays; if possible initialize with **NaN** values.

Rationale: Pre-allocation improves performance and helps readability. Initialization with **NaN** helps avoid a range of potential logical errors.

8. **Rule:** Avoid “magic numbers” — i.e. hard-coded constants.

Rationale: Constants included in your code tend to hide your program logic. Also, “magic numbers” make code maintenance more difficult and error prone.

9. **Rule:** Only write one statement per line.

Rationale: Multi-statement-lines hurt code readability in almost all cases.

10. **Rule:** Do not write excessively long lines of code; use the line continuation “...” and indentation to spread long expressions over several lines.

Rationale: Following this rule improves code readability.

Index

Bessel Function, modified, 114
Bessel's Equation, 103
Bessel's Equation, modified, 114
Bessel's Equation, parametric, 114
boundary value problem, 42

Cauchy-Euler equation, 63
continuity, 42
convergence, 72
convergence, interval of, 72
convergence, radius of, 72

divergence, 72

error, modeling, 288
error, round-off, 288
error, truncation, 288
Euler formula, 49
even function, 130

Fourier Series, 127

gamma function, 106
Gibbs phenomena, 135

indicial equation, 99

infinite series, 72
initial value problem, 41

Jacobian, 322

Legendre polynomials, 93
Legendre's equation, 91
limit, 72
linear constant coefficient equations, 47
linear dependence, 43
linear ordinary differential equation, 30

method of Frobenius, 97
method of solution, first order linear, 36

neutron diffusion equation, 258

odd function, 130
ordinary differential equation, 30

partial differential equation, 30
permutation matrix, 349

point, ordinary, 77
point, singular, 77
power series, 72
power series, three-term recurrence, 86

separable differential equation, 33
sequence, 71
series, 72
spectral radius, 365
standard form, first order linear, 35
Standard form, second order linear, 77

Sturm-Liouville Eigenvalue Problem, 141

superposition, 43

Taylor series expansion, 321

Undetermined Coefficients, 54

van der Waals equation, 306
variation of parameters, 36

Wronskian, 44