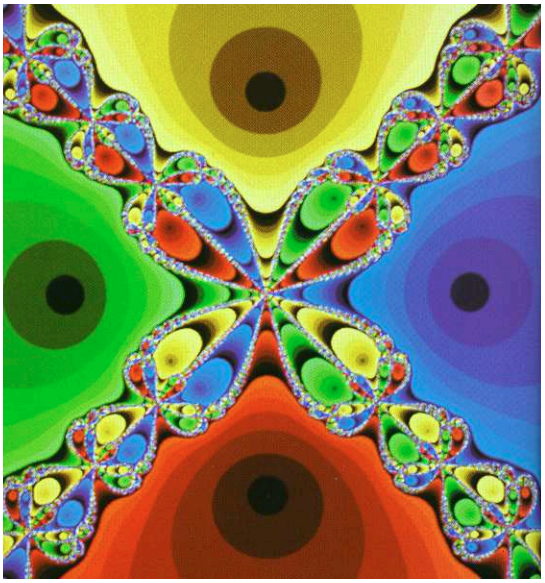


An Introduction to Computer Programming for Engineers and Scientists

Tad W. Patzek and Ruben Juanes

Department of Civil and Environmental Engineering

April 20, 2006

Third Edition

The authors and the publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising from the use of the information or programs contained herein.

Camera-ready manuscript typeset with LaTeX(2e).

To My Wife Joanna
And My Children:
Lucas, Sophie and Julie

Contents

1 Notation **1**

2 Introduction to E77 **3**

2.1 What is E77? 3

2.2 Subjects covered in E77 5

2.3 The Nitty-Gritty of E77 6

2.3.1 Who Are You? 7

2.4 Exercises 7

2.5 MATLAB Software 10

2.6 MATLAB code to generate Figure 2.1 10

2.7 MATLAB code to generate Figure 2.2 11

3 Getting Started **17**

3.1 What Are You Going to Learn? 17

3.2 Why Is It Important? 17

3.3 Start MATLAB 18

3.3.1 Change MATLAB window appearance 18

3.3.2 Set path to your work directory 18

3.4 Start MATLAB Editor 19

3.5 Using MATLAB Help 20

3.6 Restart MATLAB 21

3.7 MATLAB functions 21

3.8 Summary 22

3.9 Exercises 22

4 Basics of MATLAB **25**

4.1 What Are You Going to Learn? 25

4.2 Why Is It Important? 25

4.3 What Is MATLAB? 26

4.3.1 MATLAB as an Interpreter 26

4.3.2 MATLAB as a Calculator 26

4.4 Creating and Addressing Arrays 30

4.4.1 Creating Arrays 30

4.4.2 Addressing Arrays and the Colon Notation 31

4.4.3 Element-by-Element Operations 32

4.5 Files and Functions 34

4.5.1 Files 34

4.5.2 Mathematical Functions 34

4.5.3 User-Defined Functions 35

4.6 Basic Programming Tools 36

4.6.1 Relational Operators 36

4.6.2 Logical Operators 36

4.6.3 Decision-Making Structures 37

4.6.4 Loops 39

4.7 Output of Results 43

4.7.1 Formatted Text Output 43

4.7.2 Basic Plotting 44

5 Introduction to Computers 47

5.1 What Are You Going to Learn? 47

5.2 Why is it Important? 47

5.3 What is Computer? 47

5.4 Numbers, Powers and Logarithms 48

5.4.1 Numbers 49

5.4.2 Powers 51

5.4.3 Logarithms 52

5.5 Talking to a Computer 52

5.6 The File Clerk Model 54

5.7 Exercises 57

5.8 Summary 58

6 Arrays and Vectors 59

6.1 What Are You Going to Learn? 59

6.2 Why Is It Important? 59

6.3 Arrays and Vectors 59

6.4 Scalar product 62

6.5 Rotation 63

6.6 MATLAB Arrays 65

6.7 Arrays 65

6.7.1 Some useful array functions 65

6.7.2 Special matrices 66

6.7.3 Matrix multiplication 67

6.7.4 Multidimensional arrays 68

6.8 Exercises 69

6.9 MATLAB Code for Vector Algebra 70

6.10 MATLAB Code to Draw Vectors 71

7 How to Teach a Computer: Binary Addition for Idiots 75
7.1 What Are You Going to Learn? 75
7.2 Why Is It Important? 76
7.3 Binary addition 76
7.3.1 Logical Instructions 79
7.3.2 For-Loops 85
7.4 Computer Representation of Numbers 86
7.4.1 Putting it All Together 88
7.4.2 Special Values 88
7.5 Exercises 89

8 Data Structures 93
8.1 What Are You Going To Learn? 93
8.2 Why Is It Important? 94
8.3 Useful Definitions 94
8.4 Data Types in MATLAB 94
8.4.1 Complex Double-Precision Matrices 96
8.4.2 Sparse Matrices 98
8.4.3 MATLAB Strings 98
8.4.4 Cell Arrays 98
8.4.5 Structures 101
8.4.6 Objects 102
8.4.7 Multidimensional Arrays 103
8.4.8 Empty Arrays 103
8.4.9 Data Storage 103
8.5 Exercises 103

9 Fundamentals of Plotting 105
9.1 What Are You Going To Learn? 105
9.2 Why Is It Important? 106
9.3 What Is MATLAB Figure? 106
9.4 Figure Labels 109
9.5 Figure Legends 110
9.6 Multiple Plots in One Figure 111
9.7 Surface Plots 113
9.8 Parametric Plots 114

10 Mathematical Induction 117
10.1 What Are You Going To Learn? 117
10.2 Why Is It Important? 117
10.3 Important Definitions 117
10.4 Essentials of Mathematical Induction 118
10.4.1 Structure of Proof by Induction 119
10.4.2 Example: A sum of Fibonacci Numbers 121
10.5 Fibonacci and His Numbers 123

10.6 Exercises 123

11 Recursion, Part I **125**

11.1 What Are You Going To Learn? 125

11.2 Why Is It Important? 126

11.3 Useful Background 126

11.4 Introduction 128

11.5 More Complicated Recursion: Greatest Common Divisor (GCD) 132

11.5.1 GCD Algorithm 1: Brute Force 132

11.5.2 GCD Algorithm 2: Euclid's Algorithm 133

11.5.3 GCD Algorithm 3: Dijkstra's Algorithm 134

11.6 Summary and More Solved Examples 135

11.7 Exercises 137

12 Trees **139**

12.1 What Are You Going To Learn? 139

12.2 Why Is It Important? 139

12.3 Useful Definitions 139

12.3.1 Binary trees 140

12.3.2 Traversing binary trees 140

12.4 Traversing Forests of Trees 142

12.5 Recursion Trees 143

13 Finding Roots of Nonlinear Functions **147**

13.1 What Are You Going To Learn? 147

13.2 Why Is It Important? 148

13.3 Introduction 148

13.3.1 A Sample Nonlinear Function 148

13.3.2 Function Handles 149

13.4 Bisection method 149

13.5 Newton's method 151

13.6 MATLAB Code for the Bisection Method 157

13.7 MATLAB Code for Newton's Method 159

13.8 MATLAB Code for Newton with Bisection 160

13.9 MATLAB Code for Newton Method's Domains of Influence . . . 162

14 Iteration: Fractal Geometry of Mandelbrot's Set **165**

14.1 What Are You Going To Learn? 165

14.2 Why Is It Important? 165

14.3 Iteration and Mandlebrot's Set 166

14.4 The Mandelbrot Set 169

14.5 Why $z > 2$ Runs Away to Infinity? 173

14.6 MATLAB Code for the MandelBrot Iteration 175

14.7 MATLAB Code for the MandelBrot Set 178

15 Recursion, Part II* **181**
15.1 What Are You Going To Learn? 181
15.2 Why is it important? 182
15.3 One-Step Feedback Machine: First Order Recursion 182
15.4 One-Step Feedback Machine with Two Variables: Second Order Linear Recursion 184
15.4.1 Feedback Machines with Memory 184
15.4.2 One-Step Machines with Two Variables 185
15.5 Exercises 191

16 Quadratic Iterator: Don't Trust Your Computer* **193**
16.1 What Are You Going To Learn? 193
16.2 Why Is It Important? 193
16.3 Quadratic Iterators 194
16.3.1 Error Propagation Matters! 196
16.3.2 More on the Quadratic Iteration 197
16.3.3 Equivalence of Logistic Equation and Quadratic Iterator . 198
16.3.4 Is Loss of Precision Alone Responsible For the Sometimes Wild Behavior of the Iterator? 199
16.3.5 Graphical representation of the feedback process 200
16.4 Exercises 204

17 Polynomials and Taylor Series **205**
17.1 What Are You Going To Learn? 205
17.2 Why Is It Important? 205
17.3 Useful Background 205
17.4 Polynomials 207
17.4.1 Representation 207
17.4.2 Evaluation of polynomials 207
17.4.3 Addition of polynomials 208
17.4.4 Multiplication (convolution) of polynomials 209
17.4.5 Division (deconvolution) of polynomials 210
17.4.6 Rational functions 210
17.5 Taylor series expansion 211
17.5.1 Example 1. Exponential function 212
17.5.2 Example 2. Natural logarithm 213
17.6 Taylor Series for e^x 214
17.7 Taylor Series for $\ln x$ 214

18 Solution of Systems of Linear Equations **217**
18.1 What Are You Going To Learn? 217
18.2 Why Is It Important? 217
18.3 Linear systems of equations 218
18.3.1 Introduction 218
18.3.2 Determined systems 219
18.3.3 Underdetermined systems 220

18.3.4 Overdetermined systems 221
18.3.5 Solving systems of equations with MATLAB 223

19 Least Squares Fit to a Straight Line 227
19.1 What Are You Going To Learn? 227
19.2 Why Is It Important? 228
19.3 Useful Definitions 228
19.4 Fitting Data to a Straight Line 231
19.5 Normal Equations 233
19.6 Analysis of Variance 234
19.6.1 How Good is the Correlation? 234
19.6.2 What Are the Uncertainties in a and b? 235
19.7 Better Least Squares Equations 236
19.8 Example of Least Square Fit 236
19.9 Exercises 237
19.10Least Square Fit Demo 239
19.11Least Square Fit Algorithm Based on
Eqs. (19.22-19.24) 240

20 Least Squares Fit to a Polynomial 243
20.1 What Are You Going To Learn? 243
20.2 Why Is It Important? 243
20.3 General Linear Least Squares 244
20.3.1 Solution With the Normal Equations 245
20.3.2 Examples 247
20.4 Exercises 247
20.5 General Linear Least Square Fit Demo 248
20.6 General Linear Least Square Fit Algorithm Based on Eqs. (20.9
- 20.10) 250

21 Interpolation and Approximation 253
21.1 What Are You Going To Learn? 253
21.2 Why Is It Important? 253
21.3 MATLAB Shortcut 254
21.4 Preliminaries 254
21.5 Approximation 254
21.6 Interpolation 256
21.7 Lagrange Interpolations of Different Orders 259
21.8 Lagrange Interpolation Demo 263
21.9 Interpolation of Sensor Positions 266

22 Numerical Differentiation 271
22.1 What Are You Going To Learn? 271
22.2 Why Is It Important? 271
22.3 MATLAB Shortcut 271
22.4 Finite Differences 272

22.4.1 First Derivative 272
22.4.2 Second Derivative 274
22.5 Differentiation Demo 280

23 Numerical Integration 285
23.1 What Are You Going To Learn? 285
23.2 MATLAB Shortcut 285
23.3 Why Is It Important? 285
23.4 Introduction 286
23.5 Closed Newton-Cotes Formulæ 287
23.5.1 The Trapezoidal Rule 287
23.5.2 Simpson's Rule 288
23.5.3 Boole's Rule 288
23.6 Extended Formulae (Closed) 288
23.6.1 Extended Trapezoidal Rule 289
23.6.2 Extended Simpson's Rule 289
23.7 Romberg Integration 289

24 Ordinary Differential Equations 291
24.1 What Are You Going To Learn? 291
24.2 Why Is It Important? 292
24.3 MATLAB Shortcut 292
24.4 Conversion of ODEs into the standard MATLAB form 293
24.5 Simple Example: Throwing a Ball 294
24.5.1 Euler's Forward Method 294
24.5.2 Runge-Kutta Method 295
24.6 Predator-Prey Processes 298
24.7 MATLAB Code for the Baseball Problem 300
24.8 MATLAB Code for the L-V Equation 302

25 Sort & Search Algorithms, Big O Analysis 305
25.1 What are you going to learn? 305
25.2 Why is it important? 306
25.3 Sorting 306
25.3.1 Bubble sort 307
25.3.2 Straight insertion (Picksort) 308
25.3.3 Other methods 309
25.4 Searching 312
25.4.1 Binary search 312
25.5 Analysis of algorithms 313
25.5.1 ***O***-notation 313
25.5.2 Analysis of bubble sort 315
25.5.3 Analysis of MATLAB sort 316
25.5.4 Analysis of binary search 316

26 Data Structures, Part II. Linked Lists, Queues and Stacks 321
26.1 What Are You Going To Learn? 321
26.2 Why Is It Important? 321
26.3 MATLAB Shortcuts 322
26.3.1 Summary 322
26.3.2 Toolbox contents 322
26.4 Useful Definitions 324
26.5 Singly Linked List, Example 327
26.5.1 What Does the Demo Do? 328
26.6 Stack Example 329
26.6.1 What Does the Demo Do? 330
26.7 Queue Example 331
26.7.1 What Does the Demo Do? 332

27 Graphs, Adjacency Matrices and Lists, & Depth-First Search 333
27.1 What Are You Going To Learn? 333
27.2 Why Is It Important? 334
27.3 Background 335
27.4 Graph Definitions 337
27.5 Adjacency Matrix Representation 339
27.5.1 Examples 340
27.6 Adjacency List Representation 341
27.7 Depth-First-Search 343
27.8 Exercises 343

28 Object Oriented Programming in MATLAB 345
28.1 What Are You Going To Learn? 345
28.2 Why Is It Important? 346
28.3 Classes and Objects 347
28.4 Example: A Polynomial Class 347
28.4.1 Polynom Data Structure 347
28.4.2 Polynom Methods 348
28.5 Object Precedence 355
28.5.1 Specifying Precedence of User-Defined Classes 356
28.6 Followup 356

29 The Greatest Minds Behind the Modern Computer 357
29.1 What Are You Going To Learn? 357
29.2 Why Is It Important? 357

List of Tables

2.1 E77 Course Outline 15

4.1 Scalar arithmetic operations 27

7.1 An English oral numbering system using base 2 [19]. 77
7.2 Decimal integers and their binary equivalents 77
7.3 The basic logical operators 80
7.4 The Truth Table for the logical AND operator 81
7.5 The Truth Table for the logical OR operator 81
7.6 The Truth Table for the logical XOR operator 81
7.7 Conversion of decimal 813 to a binary array 91
7.8 Conversion of decimal 0.1 to a binary array 91

11.1 Steps in implementing a mathematical definition. 128

19.1 Assorted poisons in the popular cigarettes 230
19.2 Covariances and correlation coefficients 230

24.1 MATLAB ODE integration functions 292

25.1 Example of bubble sort 308
25.2 Example of straight insertion 309
25.3 Examples of binary search 313

List of Figures

2.1 Student distribution by years at Cal 7
2.2 Student distribution by major 8

3.1 MATLAB command window 19

4.1 Flowchart of a program written in a high-level language, such as Fortran or C 27
4.2 Flowchart of a program written in an interpreted language, such as MATLAB 27
4.3 Matlab plot of the function $f(x, y)$ 38
4.4 Matlab plot of the function $y = \sin(x)$ 45

5.1 1950: ENIAC 49
5.2 1960: IBM 360 50
5.3 1990: Cray Y-MP/432 50
5.4 Records on a salesman data card. 54

6.1 A 3D array 60
6.2 Coordinates of two unique vectors in 2D 62
6.3 Vector rotation in 2D 63
6.4 Output of function vectoralgebra.m. 64

7.1 Ancient abacus 78
7.2 Binary addition of pebbles in compartments of an ice tray 78
7.3 Binary addition function 82
7.4 Binary representation of a number with single precision (32 bits) 86
7.5 Binary representation of a number with double precision (64 bits) 87

8.1 Result of cellplot(B). 101

9.1 Hierarchy of MATLAB graphics objects. 109
9.2 Displaying multiple legends. 110
9.3 A two-by-two plot created with the subplot command. 112
9.4 MATLAB plot of the function $f(x, y)$ defined in pwfun.m. 113
9.5 A three-dimensional spiral generated as a parametric curve. . . . 115

10.1 Plant stems follow the FIBONACCI sequence 120
10.2 FIBONACCI'S statue. 121
10.3 Unrestricted growth of rabbit population follows the FIBONACCI sequence 123

11.1 The first book of *Algebra* 127
11.2 The recursive calculation of n factorial 130
11.3 Flowchart of 15 recursive calls made by `fib(5)` function. 131
11.4 Flowchart of calculations in a recursive function 136

12.1 Tree examples 141
12.2 Different binary trees 142
12.3 Binary tree used in Example 12 142
12.4 Binary tree representation of calculations on an HP and TI calculator 143
12.5 Forest traversal 144
12.6 Flowchart of the four recursive calls made by `nf(3)` function. . . 144
12.7 Flowchart of 15 recursive calls made by `fib(5)` function. 145

13.1 The roots of $f = z^3 - 1$, and their rough basins of attraction for NEWTON's method 150
13.2 Function $f(x) = x^3 - 1$ 151
13.3 Sequence of intervals 151
13.4 Evolution of error 151
13.5 Geometrical interpretation of NEWTON's method 152
13.6 NEWTON's method with $x_0 = 2$ 153
13.7 NEWTON's method with $x_0 = -2$ 154
13.8 Domains of convergence of NEWTON's method have fractal boundaries 156

14.1 The MANDELBROT set 166
14.2 n-cycle regions in the MANDELBROT set 169
14.3 A "time" series plot and amplitude histogram for an orbit 171
14.4 A "time" series plot and amplitude histogram for an orbit 172
14.5 A "time" series plot and amplitude histogram for an orbit 173
14.6 A "time" series plot and amplitude histogram for an orbit 174
14.7 A "time" series plot and amplitude histogram for an orbit 175

15.1 One-step feedback machine 182
15.2 Feedback machine with memory 184
15.3 One-step feedback machine with two variables. 185
15.4 The ancestral tree of the FIBONACCI rabbit family 186
15.5 he FIBONACCI numbers grow exponentially 187
15.6 Second order linear recurrence 188
15.7 Second order linear recurrence 189
15.8 Second order linear recurrence 189

16.1 The quadratic iterator with a control parameter c. 194

16.2 The "time series" of a quadratic iterator response. 196

16.3 P. F. Verhulst. 198

16.4 The "time series" of a chaotic response of the quadratic iterator. 199

16.5 The "time series" of a chaotic response of the quadratic iterator . 200

16.6 The "time series" of a fixed-point response of the quadratic iterator. 201

16.7 The "time series" of a fixed-point response of the quadratic iterator. 202

16.8 The "time series" of a cyclic response of the quadratic iterator. 203

16.9 The "time series" of a chaotic response of the quadratic iterator. 204

17.1 Plot of a polynomial 208

17.2 Plot of three polynomials 209

17.3 B. Taylor 211

17.4 Taylor series approximation of $f(x) = \exp(x)$. 212

17.5 Taylor series approximation of $f(x) = \ln(x)$. 213

18.1 Graphical representation of a 2 × 2 determined system. 220

18.2 A 2 × 2 underdetermined system with no solution. 221

18.3 A 2 × 2 underdetermined system with infinite solutions. 222

18.4 Graphical representation of a 3 × 2 overdetermined system. 222

18.5 Electric circuit. 224

19.1 Least square fits of the cigarette data in Table 19.1 231

19.2 Geometry of identity (19.15). 234

19.3 Unweighted least square fit 237

19.4 Weighted least square fit 237

20.1 Unweighted cubic fit 247

20.2 Weighted cubic fit 248

21.1 Karl Theodor Wilhelm Weierstrass 255

21.2 Joseph-Louis Lagrange 257

21.3 Different orders of function interpolation 258

21.4 Linear interpolation 258

21.5 Quadratic interpolation 259

21.6 Cubic interpolation 260

21.7 Sensor amplitude interpolations 260

21.8 Piecewise linear Lagrange interpolation 261

21.9 24th degree Lagrange polynomial interpolation 261

21.10Step function in sensor amplitudes. 262

21.11Piecewise linear Lagrange interpolation of a step function 262

21.12 9th-degree polynomial Lagrange interpolation of a step function 263

22.1 Equally spaced points on the x-axis. 272

22.2 Forward, backward and central finite difference 274

22.3 The measured sensor amplitudes 276

22.4 Interpolation and extrapolation of sensor positions using cubic splines . . . 277
22.5 The exact first derivative of sensor amplitude and its numerical approximation . . . 277
22.6 The exact second derivative of sensor amplitude and its numerical approximation . . . 278
22.7 The first derivative of e^x using the cubic spline approximation and central difference. . . . 278
22.8 The second derivative of e^x using the he cubic spline approximation and central difference. . . . 279
22.9 The first derivative of e^x by taking the logarithm and scaling back279
22.10 The second derivative of e^x by taking the logarithm and scaling back . . . 280

23.1 Geometrical interpretation of the integral $\int_a^b f(x)\,dx$ . . . 286
23.2 Geometrical interpretation of the trapezoidal rule. . . . 289
23.3 Geometrical interpretation of SIMPSON's rule. . . . 290

24.1 The exact and approximate baseball trajectory with the forward EULER . . . 296
24.2 The baseball trajectory with texttttode45.m and `rk4.m` . . . 298
24.3 Trajectories of prey and predator populations . . . 299
24.4 History of prey and predator populations . . . 300

25.1 Big O analysis of truncation error . . . 315
25.2 Scaling of bubble sort with n^2. . . . 317
25.3 Scaling of pick sort with n^2. . . . 318
25.4 Scaling of MATLAB sort with $n \log n$. . . . 319

26.1 A singly linked list . . . 325
26.2 Stack . . . 326
26.3 Queue . . . 326

27.1 A walk around the Königsberg bridges . . . 335
27.2 LEONHARD EULER . . . 336
27.3 An *undirected* graph . . . 337
27.4 An equivalent undirected graph . . . 338
27.5 A *directed* graph . . . 339
27.6 A simple undirected graph . . . 339
27.7 A more complicated directed graph . . . 340
27.8 The directed graph in **Example 26**. . . . 341
27.9 The *square* of the graph in Figure 27.8. . . . 342

29.1 EUCLID . . . 358
29.2 RENÉ DESCARTES . . . 359
29.3 GOTTFRIED WILHELM VON LEIBNIZ . . . 360
29.4 GEORGE BOOLE . . . 361

29.5 Gottlob Frege 362
29.6 Georg Cantor 363
29.7 David Hilbert 364
29.8 Kurt Gödel 365
29.9 Alan Mathison Turing 366
29.10 John (János) von Neumann 367
29.11 Donald Knuth 368

Chapter 1

Notation

Generally, we use the notation introduced by HOUSEHOLDER [17]:

Capital letters	A, B, C, Δ, Λ	for arrays
Capital bold letters	$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{\Delta}, \boldsymbol{\Lambda}$	for matrices
Subscripted lower case letters	$a_{ij}, b_{mn}, c_{kl}, \delta_{ij}, \gamma_{ij}$	for matrix elements
Lower case letters	x, y, z, v, h	for 1D "vectors"
Lower case bold letters	$\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$	for physical vectors
Subscripted lower case letters	$x_i, b_j, c_k, \delta_j, \gamma_j$	for vector elements
Lower case Greek letters	$\alpha, \beta, \delta, \varepsilon, \varphi$	for scalars

In reporting particular implementations of the algorithms discussed in this class (we call these implementations *computer codes*), we use the following approach:

- All algorithms will be represented in MATLAB language as actual code or, sometimes, as pseudocode.
- Comments begin with the % symbol.
- The values taken by an integer variable are described using the colon notation: "`i = 1:n`" means the same as "$i = 1, 2, 3, \ldots, n-1, n$".
- Arrays (matrices) and and subarrays (submatrices) are also represented in the colon notation. For example `A(p:q,r:s)` denotes the subarray (submatrix) of A formed by the intersection of rows p and q and columns r and s.

Chapter 2

Introduction to E77

In his brilliant little book, Roszak [29] states:

"The burden of my argument is to insist that there is a vital distinction between what machines do when they process information and what minds do when they think. At a time when computers are being intruded massively upon the schools, that distinction needs to be kept plainly in view by teachers and students alike. But thanks to the cult-like mystique that has come to surround the computer, the line that divides mind from machine is being blurred. Accordingly, the powers of reason and imagination, which the schools exist to celebrate and strengthen, are in danger of being diluted with lowgrade mechanical counterfeits.

If we wish to reclaim the true art of thinking from this crippling confusion, we must begin by cutting our way through undergrowth of advertising hype, media fictions, and commercial propaganda. But having done that much to clear the ground, we come upon the hard philosophical core of the cult of information, which is as much the creation of the academies and laboratories as of the marketplace. Gifted minds in the field of computer science have joined the cult for reasons of power and profit. Because the hucksters have enlisted so many scientists in their cause, there are tough intellectual questions as well as political interests that need to be examined if we are to understand the full influence of the computer in our society. In a very real sense, the powers and purposes of the human mind are at issue. If the educators are also finally swept into the cult, we may see the rising generation of students seriously hampered in its capacity to think through the social and ethical questions that confront us as we pass through the latest stage of the ongoing industrial revolution."

2.1 What is E77?

E77 will introduce you to the principles of programming in the context of engineering and scientific applications. MATLAB® under MS Windows 2000® Professional will be the high-level computing environment of choice.

You simply cannot be a good engineer or scientist if you do not know what a digital computer can or cannot do. To survive the onslaught of computer propaganda and the software sales pitches, you must have good working knowledge of basic computational algorithms, programming techniques, and inherent limitations of finite computing machines.

It may come to you as surprise that computer science is not strictly a science[1], such as physics, astronomy or chemistry; it does not study natural phenomena [12]. While it uses logic and mathematical concepts extensively, computer science in *not* pure mathematics, as you might be tempted to think. Rather, computer science is akin to engineering: its purpose is to get some device, or a system of devices, to do something, rather than dealing with pure abstractions. I am far from saying that computer science is all practical, like paving a road or constructing a building; it is actually very far from it.

On the other hand, as a famous mathematician and one of the fathers of the digital computer, John von Neumann, once said [15]: "The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretation, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work - that is, correctly describe phenomena from a reasonable wide area." Von Neumann's definition provides a nice bridge among natural sciences, computer science and the various branches of engineering.

After an initial period of exhilaration and unwarranted optimism[2], we are finding out that computer science and computer hardware have changed almost every detail of our lives for better or worse. In short computers have become a mature technology: they create as many problems as they solve. Says Roszak [29]: "The use of computers in the classroom raises several issues. By far the strongest argument for going high tech in the schools has been the promise of future employment; the computer literacy supposedly represents the royal road to a good job. But, by a cruel irony, in the global economy that surrounds our schools, the main effect of automation and computerization is to deskill, disempower, and disemploy people from the assembly line up through middle management. Computers have played a crucial role in the downsizing of companies and in the creation of a part-time, temporary and low-paid workforce. Our schools do not exist in vacuum. They are surrounded and permeated by social and economic forces that pose more problems in the lives of our students than any machine will ever solve."

My goal in E77 is to provide you with a set of skills that are strong and fundamental enough to guard you against becoming obsolete and downsized for many years. I will not achieve this goal without your *very* active help. In the future you will have to apply your new skills to everyday engineering practice, or else they will atrophy and you will downsize yourself.

In E77, we shall immediately go down to the "salt mines," while trying to

[1] From Latin *scientia*: having knowledge or to know. The scientific knowledge is concerned with the physical world and its phenomena.

[2] Please read some of the more outrageous quotations at the end of this chapter.

generalize whenever possible. It does no harm to concentrate on details first. The accompanying computer laboratories will be all salt mines, while in the lectures generalizations will occur quite often. The homework assignments will be a go-between the labs and lectures.

Remark 1 Lectures, computer laboratories and homework assignments serve different purposes and will not always overlap.

2.2 Subjects covered in E77

Many different corners of computer science and several computational techniques will be visited[3] in E77:

1. The rudiments of computer architecture and basic grammar of the new MATLAB language you will have to learn to talk to the idiot computer. Because you are much smarter than the computer, you will have to explain very carefully what you want the dense machine to do. This will introduce you to procedural thinking and simplest algorithms.
2. More details of the MATLAB language, problem solving and plotting techniques.
3. The most versatile computer science tools to build long-lasting computer programs: induction, recursion and iteration, and the many links between recursion and iteration.
4. The representation of real numbers in binary form, the cost of arithmetic operations, and the common sources of computation errors.
5. The big O notation to estimate the computational costs of different algorithms.
6. The most versatile holders of MATLAB data, called structures and cell arrays.
7. Solving systems of linear equations in the context of the least-square data fits.
8. Interpolation, extrapolation and smoothing of data.
9. The numerical differentiation and integration, and how to integrate some simple ordinary differential equations.
10. Trees and their relationships to computer algorithms.

[3] Grouped by theme, not by chapter. We will try to show you the more important subjects in several different ways.

11. Data sorting algorithms and their efficiencies.

12. Other data structures: linked lists, queues and stacks

13. Most general data structures: graphs, adjacency matrices and neighbor lists

14. Introduction to the Object-Oriented-Programming, or the most fashionable OOP ideology.

2.3 The Nitty-Gritty of E77

Here are some useful tidbits about the course:

1. There will be 29 lecture topics in this course. The material covered in both sections will be more less identical.

2. There will about 12 required computer laboratories.

3. Homework assignments will be due every two weeks and will include more substantial problems.

4. **Before** each new lecture, you will be asked to read the Class Reader, as well as the corresponding chapter(s) in Palm's book [25]. The required reading is listed in **Table 2.1**.

5. The chapters that are not required are denoted with an asterisk; read them anyway.

6. You are encouraged to participate actively in the lectures by asking questions and requesting further explanations. In fact, the more active you are the better off we all are.

7. You are highly encouraged to solve all exercises at the end of each chapter. Some of them will end up as homework assignments. If you turn in electronically a working solution, you will earn "brownie points." More importantly, you can learn computer science only by solving practical problems of increasing complexity.

8. Inevitably, as in every large class, there will be a vocal and active minority in the first two-three rows, and a silent majority behind. Some of you will continue to wonder why are you here? I do hope that by end of the semester you will find out. Others will try to avoid coming to lectures altogether because (a) the lectures are a waste of time, or (b) you already know everything that will be covered in this stupid course the college requires you to take.

9. It may come as a surprise to you "experts" that you will have to unlearn some of your poor or downright harmful programming attitudes to achieve success in this course. The non-experts will just have to learn the proper techniques.

10. Both midterms and the final will be the same for the entire class.

2.3.1 Who Are You?

Your are mostly sophomores and older, see **Figure 2.1**, and mostly Mechanical, Chemical, Undeclared, Civil, and Bio engineers, see **Figure 2.2**.

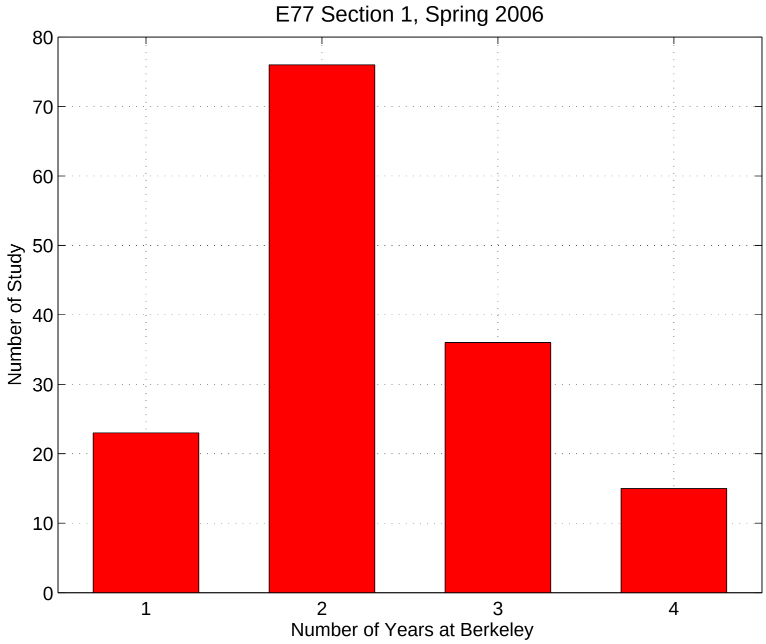


Figure 2.1: The frequency distribution of the number of years, "Levels," you have been at Berkeley (Level 1 = Freshman,...), generated with MATLAB code listed as `function PlotLevels()`.

2.4 Exercises

1. Read the chapter quotes at the end. Please email me the funniest or most outrageous quotation you can find and include its source. The best quotations will make it to the future lectures with a reference to your names.

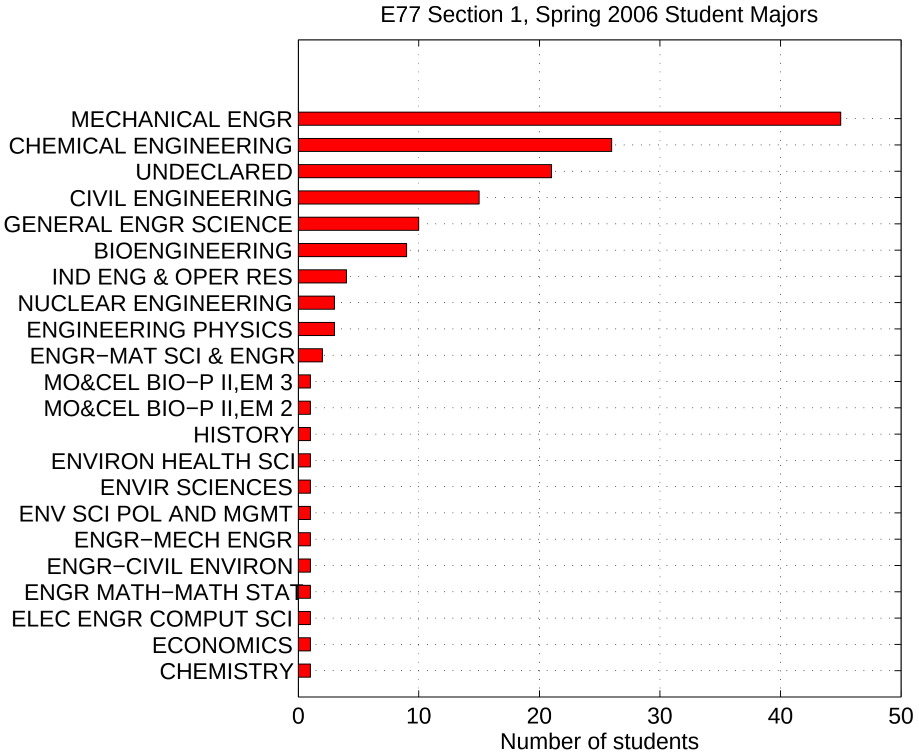


Figure 2.2: The frequency distribution of the majors you have declared. This plot has been generated with the MATLAB code listed as `function PlotMajors()`. Do not fear the apparent complexity of this code; soon you'll see how easy it actually is. I am not expecting you to write such code until much later in the semester; still, check out the listing.

2. Read, as much as you can, the two MATLAB code examples and try to figure out what is going on there. You can download the source code from fttp:\\petroleum.berkeley.edu, and play with it as much as you care.

2.5 MATLAB Software

This book comes with many MATLAB programs which you might use to learn the required material faster and develop your own programs better. To get you into the mood, please go to the Introduction folder on fttp:\\petroleum.berkeley.edu, where there are two files that contain two MATLAB *functions*:

PlotLevels.m to generate a fancy bar graph with the distribution of you by the number of years at Cal: freshmen, sophomores, juniors, seniors.

PlotMajors.m to generate a bar graph with the deparements/programs you are from.

NOTE: We *do not* expect you to know or even understand the code in these two functions. Please just look it up, to see what you will be able to do two months from today.

2.6 MATLAB code to generate Figure 2.1

```
function PlotLevels()
%--------------------------------------------------------
% Plot the frequency of student levels in Spring 2006
% E77, Section 1.  Level 1 = Freshman, 2 = Sophomore,...
% Written by T.W. Patzek, January 16, 2006
% E77 Class Material, University of California, Berkeley
%--------------------------------------------------------

clc % = CLear Console = clear screen
%
% Column vector Levels (Levels are years of enrollment)
% The three dots allow me to continue in next line...
Levels = [...
1
2
3
4];

filename = 'Section1ClassInfo.xls';
[Data] = xlsread(filename,'ClassInfo','c1:c150');

%Bin the data using the MATLAB function hist
[Frequency,n] = hist(Data,Levels);

% Dump them on the screen
Levels
Frequency

% Transpose column vector Levels into a row vector
Levels = Levels'
```

```
%
close all % Close all open figures

% Start a new plot
figure(99)

% Plot a bar graph
% Levels and Frequency must have the same sizes
% The third argument, 0.6, is the relative width of the bars:
% 1 is for the bars that touch
bar(Levels, Frequency, 0.6)

set(gca,'Fontsize',12)

title('E77 Section 1, Spring 2006','fontsize', 14)
xlabel('Number of Years at Berkeley')
ylabel('Number of Study')

% Keep on adding features as needed...
colormap hsv %new color map
grid on  % want a grid?
box on   % want a bounding box for the plot?

% Print to string fn the desired eps file name
fn = sprintf('E77N-Figure1.1.eps');
% Print to a color eps file for handout
print( gcf, '-depsc2', fn );
```

2.7 MATLAB code to generate Figure 2.2

```
function PlotMajors()
%--------------------------------------------------------
% Plot major fields in Section 1 of Spring E77
% Written by T.W. Patzek, January 16, 2006
% E77 Class Material, University of California, Berkeley
% Modified by TWP, 01-16-2006
%--------------------------------------------------------

clc         % Clear screen
close all   % Close all open figures

filename = 'Section1ClassInfo.xls';
[Data,Text] = xlsread(filename,'ClassInfo','d1:d150')
% Find the unique names of majors in Text
Major = unique(Text)
[len,dummy] = size(Major);
% Initialize the number of students in each major to zero
Numbers = zeros(len,1);
```

```
% Accumulate the Major frequencies
for ii =1:length(Text)
    localmajor = Text(ii,:);
    for jj=1:len
        if strcmp(localmajor,Major(jj,:))
            % Number of students in each major from the class roster
            Numbers(jj)=Numbers(jj)+1;
        end
    end
end

% Sort the student numbers in descending order
[NS,I] = sort(Numbers);
% Sort the corresponding labels accordingly
MS = Major(I,:);

% Print the results tothe screen
fprintf('\n\nList of majors\n\n')
for ii =1:len
    fprintf('%s  =  %3d\n',char(MS(ii,:)),NS(ii))
end

% Fictitious y-labels: 1,2,3,...
y = 1:1:len;

% Initialize a new figure object with generic properties...
%


% Get the current screen size and set the figure size
% so that it does not cover the bottom screen bar
SS = get(0, 'ScreenSize')

% Initialize a new figure
figure('Visible','on',...
    'Units','Pixels',...
    'Resize','on',...
    'Name','Major fields',...
    'Numbertitle','off',...
    'Position',[1,1,SS(3),SS(4)-75],...
    'BackingStore', 'off');


% Specify plot area
%                                     xmin   ymin   dx    dy
axes('Units','Normalized','Position',[0.28  0.15  0.55  0.80])
% Plot HORIZONTAL bars
barh(y,NS,0.5);

% Replace the y-labels with the strings in the array Major
% gca = get current axes handle.  h now points to a MATLAB
% object, which is full of properties which you can change
h = gca;

% Now change the y-labels by putting the strings from the
% sorted array Major at the y-coordinates dictated by vector y
set(h, 'YTickLabel', MS, 'YTick', y)

% Set the default font size as 12 for all objects inside h
set(h,'fontsize',12)

% So there is no need to specify the font size here...
title('E77 Section 1, Spring 2006 Student Majors')
xlabel('Number of students')

% Keep on adding features as needed...
colormap hsv
grid on
box on

% Print to string fn the desired eps file name
fn = sprintf('E77N-Figure1.2.eps');
% Print to a color eps Level 2 file for handout
print( gcf, '-depsc2', fn );
```

Becoming computer-literate...is a chance to spend your life working with devices smarter than you are, and yet have control over them. It's like carrying a six-gun to the old frontier.

Paul Kalghan, Dean of Computer Science at the Northeastern University, New York Times, *Jan. 13, 1985.*

By 1985, computers will be in position to decide to keep us as pets.

The MIT Artificial Intelligence Agency, *1975.*

Many people erroneously believe that, simply because the computer uses fifteen significant digits, their answers will be accurate to fifteen digits. However, the speed with which some computations can be rendered useless by the cumulative effect of small errors is quite amazing....

Boggs.

Computers can only solve problems which are already understood. Ad hoc *numerical calculations are no substitute for good theory supported by sound experiment.*

Tad Patzek.

I honestly think you ought to calm down; take a stress pill and think things over.

This mission is too important for me to allow you to jeopardize it!

HAL, the Supercomputer of Space Odyssey, *2001.*

It takes twenty-one years to become an adult. We spend most of those years in school, learning the culture, history, skills, and knowledge we expect all citizens to have. After all that, it may still take decades of training to become expert in a particular endeavor. Yet HAL was only ten years old.

Donald A. Norman *Hal's Legacy*, David G. Stork, editor, MIT Press, London, *1997.*

Looking into the distant future, I suppose it's not inconceivable that a semisentient robot-computer subculture could evolve that might one day decide it no longer needed man. You've probably heard the story about the ultimate computer of the future: For months scientists think of the first question to pose to it, and finally they hit on the right one: "Is there a God?" After a moment of whirring and flashing lights, a card comes out, punched with the words: There Is Now.

Stanley Kubrick, *The Playboy Interview*, Playboy Magazine, HMH Publishing Co., Inc., *1968.*

Table 2.1: E77 Course Outline

#	Date	Subject	Reading
1	1/18/2006	Introduction	Palm*, Chapter 1
2	1/23/2006	MATLAB Basics	
3	1/25/2006	Arrays, Data Structures	Chapter 2.1-2.5, Chapter 3.1-3.3
4	1/30/2006	Vector Operations	
5	2/1/2006	I/O, if-then-else, Boolean operators	Chapter 4.1-4.4
6	2/6/2006	Recursion 1	Class Notes
7	2/8/2006	Recursion 2	Class Notes
8	2/13/2006	Iteration 1	Chapter 4.5
9	2/15/2006	Iteration 2/Fractals/Graphics	Class Notes
–	2/20/2006	President's Day	
10	2/22/2006	Midterm I	
11	2/27/2006	Computational Errors/Loss of accuracy	Class Notes
12	3/1/2006	Linear Equations	Chapter 6
13	3/6/2006	Least squares regression	Chapter 5.6
14	3/8/2006	Curve fitting, interpolation	Chapter 5.7
15	3/13/2006	Numerical root finding - first order	Class Notes
16	3/15/2006	Numerical root finding - second order	Class Notes
17	3/20/2006	Numerical differentiation	Chapter 8.1-8.3
18	3/22/2006	Numerical integration	
–	3/27/2006	Spring Recess	
–	3/29/2006	Spring Recess	
19	4/3/2006	ODE	Chapter 8.4-8.9
20	4/5/2006	ODE	
21	4/10/2006	ODE	
22	4/12/2006	Midterm II	
23	4/17/2006	Sorting/search	Class Notes
24	4/19/2006	Big(O)	Class Notes
25	4/24/2006	Trees	Class Notes
26	4/26/2006	PDF/Queues	Class Notes
27	5/1/2006	Simulation	Class Notes
28	5/3/2006	Special Topics (OOP)	Class Notes
29	5/8/2006	Special Topics (OOP)	Class Notes

(*)Palm [25]

Chapter 3

Getting Started

3.1 What Are You Going to Learn?

Your are going to learn how to:

1. Start a MATLAB session, change defaults in the MATLAB Command Window,

2. Start the MATLAB Editor, change defaults in the editor window,

3. Write your first script,

4. Save it to a MATLAB .m file in a work folder (directory) of your choice,

5. Execute the script in MATLAB command window,

6. Use MATLAB Help,

7. Restart MATLAB, change to your present work directory, open the script you just saved and run it,

8. Convert a script into a MATLAB function,

9. Run the function with different input arguments (see the exercises).

3.2 Why Is It Important?

Before you can fly, you must walk. You cannot be serious about programming in MATLAB if you do not have the rudimentary tools in your tool chest.

3.3 Start MATLAB

From your computer desktop, click on the **MATLAB** icon, or go to the **Start** button at the lower left corner of your screen, go to **Programs**, then go to **MATLAB** (or MATLAB Release 12 or similar). At this stage, a new window with **MATLAB** at its upper left corner should appear on your screen. If it has not, please consult your favorite computer expert. Other windows may also appear. Close them one-by-one by clicking on the **x** in the upper right corner of each window.

3.3.1 Change MATLAB window appearance

To change your preferences in the MATLAB command window, click on the menu bar **File**, **Preferences**, **Command Window**, **Font and Colors**. I prefer black **Background color** and white **Text color**.

3.3.2 Set path to your work directory

I will assume that your local disk drive is labelled `C:`. On that drive you will create a new folder, in which you will keep your MATLAB programs. You can do it from the **Windows Explorer**. The new folder may be something like `C:\Smith\GettingStarted` (here substitute your particular path). For example, on my laptop the path to the programs used in this lecture is:

```
C:\MyFolders\E77N\Fall2002\Lectures\Code\GettingStarted
```

MATLAB has no idea where you keep your programs, so you must tell it *every time* you begin your session. Alternatively, you may add your directory to MATLAB's *path* by clicking on the **File** pull down menu, then on **Set Path**, **Add Folder** or **Add with Subfolders**. The first time around you may just type at the MATLAB prompt:

```
>> cd C:\MyFolders\E77N\Fall2002\Lectures\Code\GettingStarted
```

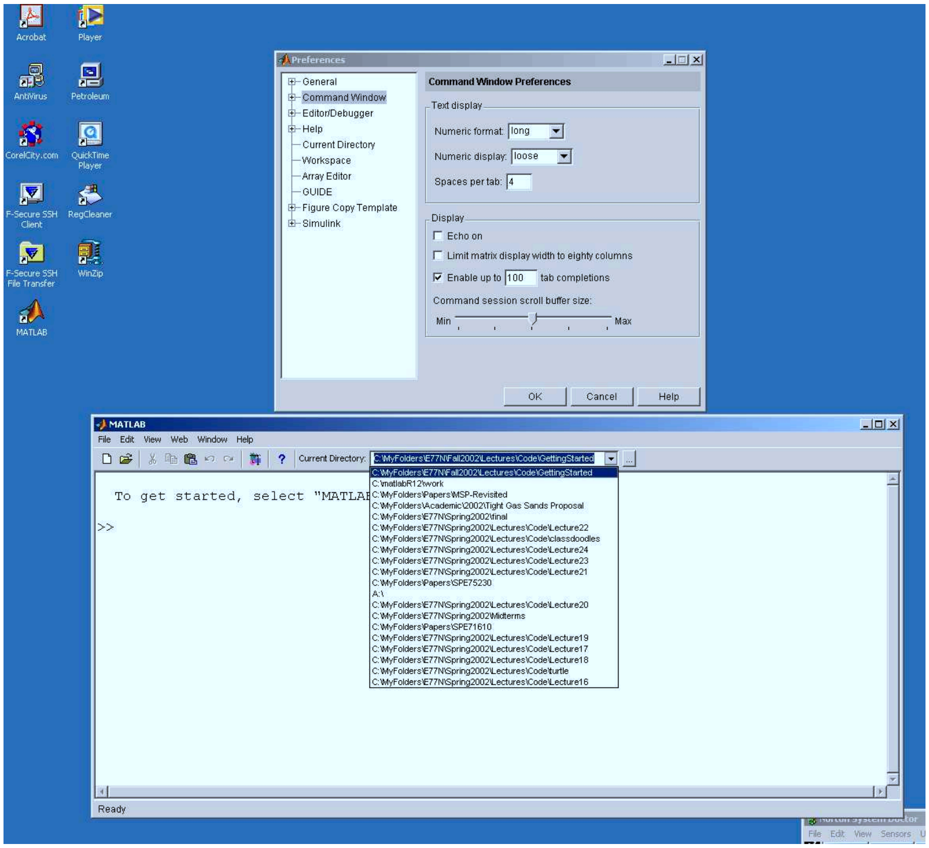
Alternatively, in the middle of the top bar of the MATLAB widow, there is a pull down menu **Current directory**, and there is a push-button with three dots. Click on the three dots, . . . , and find your directory. Next time you start the MATLAB session and you click on the pull down menu, your directory will be there, so highlight it and, voilà, you are in it. Check where you are from the MATLAB prompt by typing in:

```
>> pwd

ans =

C:\MyFolders\E77N\Fall2002\Lectures\Code\GettingStarted
```

`pwd` stands for `Present Work Directory`.



Command Window Preferences

Text display

Numeric format: long

Numeric display: loose

Spaces per tab: 4

Display

□ Echo on

□ Limit matrix display width to eighty columns

☑ Enable up to 100 tab completions

Command session scroll buffer size: Min – Max

Current Directory: C:\MyFolders\E77N\Fall2002\Lectures\Code\GettingStarted

```
C:\MyFolders\E77N\Fall2002\Lectures\Code\GettingStarted
C:\matlabR12\work
C:\MyFolders\Papers\MSP-Revisited
C:\MyFolders\Academic\2002\Tight Gas Sands Proposal
C:\MyFolders\E77N\Spring2002\final
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture22
C:\MyFolders\E77N\Spring2002\Lectures\Code\classdoodles
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture24
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture23
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture21
C:\MyFolders\Papers\SPE75230
A:\
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture20
C:\MyFolders\E77N\Spring2002\Midterms
C:\MyFolders\Papers\SPE71610
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture19
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture17
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture18
C:\MyFolders\E77N\Spring2002\Lectures\Code\turtle
C:\MyFolders\E77N\Spring2002\Lectures\Code\Lecture16
```

```
  To get started, select "MATLAI

>>
```

Figure 3.1: **MATLAB** command window and the child window created by clicking on the **File** pull down menu, and then on **Preferences**.

3.4 Start MATLAB Editor

The MATLAB editor allows you to record your work and store it in the present work directory as the MATLAB .m text files. After your work has been saved in a file, you can execute it by typing the name of the file *without* the .m. Please follow this example:

1. Start the MATLAB editor by clicking on the **File** pull down menu in the MATLAB command window, then on **New**, and **M-file**. Now a new window appears. This is your very nice MATLAB Editor and Debugger.

2. In the **Editor** window, type in something like this:

```
% cylinderscript
%
% A MATLAB script to calculate cylinder volume
% for a given height h and several radii r
h = 50;
r = [10, 20, 30];
A = pi*r.^2;
V = A.*h;
fprintf('\n\nCylinder height h = %g cm\n',h)
for i=1:length(r)
   fprintf('Radius r = %g cm, Volume V = %10g cc\n',r(i), V(i))
end
```

Note that `r` is a *vector* with three elements. The exponentiation caret operator is preceded by a dot to square each element of vector `r` separately, i.e., element-by-element. Get used to this dot.

3. In the **Editor** window click on the **File** pull down menu, then on **Save**. A new **Save file:** window will appear. In the **File name** field, type in `cylinderscript.m` and press the **Save** button. You have now saved your first MATLAB script file.

4. Go to the MATLAB command window and at the prompt `>>` type in `cylinderscript`. This is what will happen:

```
>> cylinderscript

  Cylinder height = 50 cm
  Radius r = 10 cm, Volume V =      15708 cc
  Radius r = 20 cm, Volume V =    62831.9 cc
  Radius r = 30 cm, Volume V =     141372 cc
>>
```

3.5 Using MATLAB Help

Now is a good time to check what the mysterious `fprintf(...)` command[1] does. To learn more about `fprintf` we will use the extensive and extremely useful **MATLAB Help**. To invoke it, go to the MATLAB command window, click on the **Help** pull down menu, then on **MATLAB Help**. A new **Help** window will appear. In the **Search index for**: field type in `fprintf`. Then double-click on the highlighted field. A lengthy description of `fprintf` will appear. Near the bottom, the description will (almost) always have a working example or two. You can cut the MATLAB code from the **Help** window, paste it into the **Editor** or **Command** window and execute it.

Now please go to the MATLAB command window and either click on the `x` in the upper right corner, or click on the **File** pull down menu, then on **Exit**. This will close all open MATLAB windows.

[1] In reality `fprintf` is an internal MATLAB *function*, which may be called with many different *arguments*. The MATLAB `fprintf` roughly follows the C-language function `printf`.

3.6 Restart MATLAB

1. Restart the MATLAB command window as you did the first time around. If the MATLAB Editor and Help windows were previously open, they will open up again. You can close them separately as you wish.

2. Go to the **Current Directory:** pull down menu and highlight your work directory. Double check in the MATLAB command window by typing in `pwd` as before. If you are in the right directory, type `cyliderscript` into the command window, and the script will be executed again.

3.7 MATLAB functions

You have just learned to create a MATLAB script and save it to a disk file. There are many problems with this solution. Suppose, for example, that you want to calculate the cylinder volume for an arbitrary height `h` and arbitrary radius `r`. You could type in a new value of `h` and `r` into the `cylinderscript.m` file, save it again and execute. A better solution will be to open a new M-file window and write the following code:

```
function V=cylvol(h,r)
% A MATLAB function to calculate cylinder volume,V,
% given two inputs: height h and radius r
A = pi*r.^2;
V = A.*h;
fprintf('\n\n')
fprintf('Cylinder height = %10g cm\n',h)
fprintf('Cylinder radius = %10g cm\n',r)
fprintf('Cylinder volume = %10g cc\n',V)
```

Note that when you save the new file, MATLAB will propose a name for it exactly as the name after the MATLAB *keyword* **function**. Adhere to this suggestion.

In the MATLAB command window, type in

```
  c>> cylvol(25.5,48.7)

  Cylinder height =       25.5 cm
  Cylinder radius =       48.7 cm
  Cylinder volume =     189998 cc

  ans =

    1.899975389551056e+005
>>
```

The **function** `cylvol`, which has two *input* arguments, (`h,r`), and one *output* argument, `V`, returns the value of `V` as the default MATLAB variable, `ans`, into the MATLAB command window.

Remark 2

MATLAB will always identify a function with the name of the disk file in which this function is stored.

Remark 3

You should use MATLAB functions to perform all repetitive tasks (many values of many input arguments). You may use MATLAB scripts (although I discourage the use of scripts) to call the respective functions from one place. Each function should do just one task, and do it well. Do not write very long functions; they are difficult to maintain and modify. Put as many comment lines as you can into each function you write.

3.8 Summary

Now, you can open the MATLAB command window and change its appearance. You can open the MATLAB editor, write your own MATLAB script, save it to a directory of your choice, execute it, and use MATLAB Help to guide you further. Finally you have learned how to write a MATLAB function and use it. You cannot progress further until you have mastered every step explained in this chapter.

3.9 Exercises

1. In the script file `cylinderscript.m`, the MATLAB *reserved word*, or *keyword*, **for** appears in blue. Use **MATLAB Help** to learn more about the **for** *loops*.

2. Use **MATLAB Help** to find out what the built-in MATLAB function `length()` does.

3. The word **function** is another MATLAB keyword. From **MATLAB Help** learn more about the **function** syntax.

4. In MATLAB command window, type in the following:

   ```
   >> clear all
   >> cylinderscript
   ... ...
   >> whos
   ```

   What is the result? How many variables reside in MATLAB workspace? (Go to **MATLAB Help** to learn more about `whos`).

5. In MATLAB command window, type in the following:

   ```
   >> clear all
   >> V=cylvol(10,20)
   ... ...
   >> whos
   ```

   How many variables reside in MATLAB workspace now? What is the first striking difference between a MATLAB script and function? Elaborate.

6. Type in the following:

   ```
   >> clear all
   >> h=[10,20,30];
   >> r=[1,2,3];
   >> V=cylvol(h,r)
   ... ...
   >>whos
   ```

   How many variables reside in MATLAB workspace? Note carefully the size and values of each variable.

7. Type in the following:

   ```
   >> clear all
   >> h=[10,20,30];
   >> r=[1,2,3];
   >> cylinderscript
   ... ...
   >> V=cylvol(h,r)
   ... ...
   >>whos
   ```

   What happened to the previous vectors `h` and `r`? Why is it so dangerous to use scripts?

Chapter 4

Basics of MATLAB

4.1 What Are You Going to Learn?

This chapter will teach you the most rudimentary tools you need to master in order to implement the concepts and techniques of this course in the MATLAB programming environment. The objective of this chapter is to provide a practical guide to the essentials of MATLAB and a (necessarily incomplete) list of common mistakes you should avoid. You will see that the material is presented in a brief and informal manner, mostly as examples, with the hope that this will be more helpful than a definite, more general, exposition.

In particular, in this chapter you will learn how to:

1. Use MATLAB as a desktop calculator.
2. Create arrays, the most important concept in MATLAB.
3. Extract portions of arrays and perform operations on arrays.
4. Use predefined mathematical functions, and write your own (user-defined) functions.
5. Use the fundamental constructs of traditional programming, which will allow your code to make decisions and repeat calculations.
6. Present the final output, either in text or graphic format.

4.2 Why Is It Important?

As the course progresses, you will face sophisticated topics that, inevitably, require that algorithms be implemented in the computer. The MATLAB environment makes this step a relatively easy task. However, you will need to be fluent in the MATLAB language from the onset, so that you can focus on the problem at hand, rather than on the implementation.

4.3 What Is MATLAB?

From the official MATLAB documentation: "MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building.

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for MATrix LABoratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB uses software developed by the LAPACK and ARPACK projects, which together represent the state-of-the-art in software for matrix computation."

4.3.1 MATLAB as an Interpreter

MATLAB is not a high-level language but, rather, an interpreter language. High-level languages like Fortran or C consist of a set of commands and instructions which are easy for the programmer to code. The source code is then translated by the compiler to machine language that the CPU can understand and execute. An interpreter language, although it resembles a high-level language, does *not* require the intermediate compilation process. This means that one can write code and obtain an answer from the computer immediately. The essential difference between a *compiled* and an *interpreted* language is expressed schematically in **Figures 4.1** and **4.2**.

4.3.2 MATLAB as a Calculator

It is precisely because MATLAB is an interpreter language that one can use it as a desktop calculator. This capability allows one to type an algebraic expression and obtain an answer right away. For example:

```
>> 2 + 3
ans =
     5
```

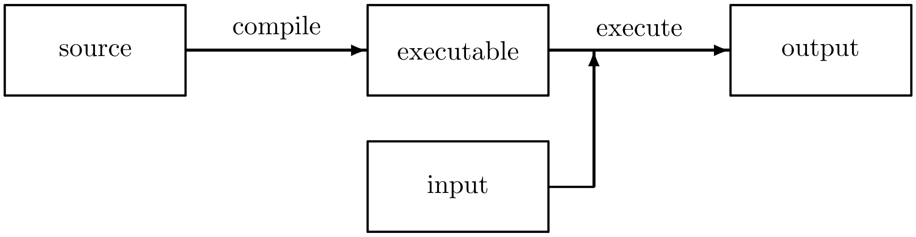


Figure 4.1: Flowchart of a program written in a high-level language, such as Fortran or C

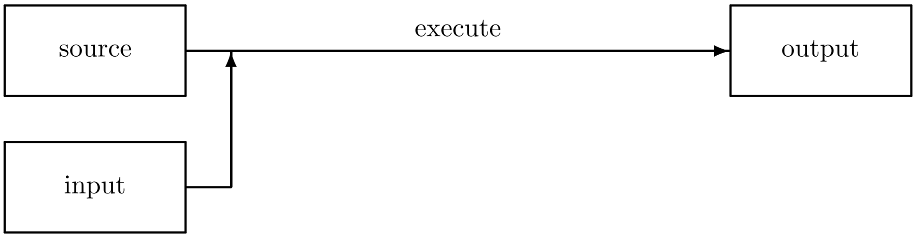


Figure 4.2: Flowchart of a program written in an interpreted language, such as MATLAB

Arithmetic operations with scalars

MATLAB uses the standard symbols for arithmetic operations with scalars:

```
+  -  *  /  ^
```

There is one additional (nonconventional) symbol, `\` , which denotes *left division*. As you will learn later, it is particularly useful when applied to vectors and matrices for solving systems of linear equations. **Table 4.1** is a summary of the scalar arithmetic operations.

The mathematical operations follow from left to right, with standard *precedence rules*, which are common to virtually all programming languages:

Symbol	**Operation**	**MATLAB** form
`^`	Exponentiation: a^b	`a^b`
`*`	Multiplication: ab	`a*b`
`/`	Right division: $a/b = \frac{a}{b}$	`a/b`
`\`	Left division: $b\backslash a = \frac{a}{b}$	`b\a`
`+`	Addition: $a + b$	`a+b`
`-`	Subtraction: $a - b$	`a-b`

Table 4.1: Scalar arithmetic operations

1. (Highest precedence) Parentheses; evaluated starting with the innermost pair.
2. Exponentiation; evaluated from left to right.
3. Multiplication and division, with equal precedence; evaluated from left to right.
4. (Lowest precedence) Addition and subtraction, with equal precedence; evaluated from left to right.

Variables and the assignment operator

Values can be assigned to *variables*. As in many other languages, the symbol of the *assignment operator* is `=` (the equal sign). For example, if we want to assign the value 3 to the variable `x`, we type:

```
>> x = 3
```

It is very important to note that the symbol `=` is *not* the equal sign of mathematical equations. Therefore, the following command is perfectly valid:

```
>> x = x + 2
```

which means that we take whatever value the variable `x` had, we increase it by 2, and we assign the new value to the same variable `x`.

Remark: Variable names are case sensitive!

If the result of any command is not assigned to any variable explicitly, MATLAB assigns it to the default variable `ans`. For example:

```
>> x = 3
x =
      3
>> x + 2
ans =
      5
```

All the variables (including `ans`) used in a MATLAB session are stored in the so-called *workspace*. If a variable is reused or its value changes at any point, the workspace of the MATLAB session will retain the latest value. The following commands are the most important ones for managing the session workspace:

who lists all variables currently in memory

whos lists all variables currently in memory and their size

clear removes all variables from memory

Certain variables are considered “special” because they have predefined default values. These are:

`Inf` Infinity, ∞ (e.g., 1/0)

`NaN` "Not a Number" (e.g., 0/0)

`eps` Accuracy of floating-point precision ($\approx 2.2204 \times 10^{-16}$)

`pi` The number $\pi = 3.1415926\ldots$

`i` The imaginary unit $\sqrt{-1}$ (same as `j`)

To avoid confusion and to follow good programming practice, it is highly recommended *not* to update the value of these variables. Although syntactically valid, commands like the ones below should never be used:

```
>> pi = 3       % never do this!
pi =
     3
>> eps = 1e-10  % never do this!
eps =
  1.0000e-010
```

Miscellaneous

By default, MATLAB prompts on the screen the result of any command. To *suppress the output*, one only has to follow the command by a semicolon (MATLAB still retains the variables's value). For example:

```
>> x = 1 + 2
x =
     3
>> y = 3 + 4;   % issues no output
```

If the command is too long to be typed in a single line, it can be *split into several lines* of code by using `...` (ellipsis) as an indication that the command continues on the following line. For example:

```
>> S = 1 + 1/2 + 1/3 + 1/4 ...
       + 1/5 + 1/6 + 1/7 + 1/8
S =
    2.7179
```

The symbol `%` designates a comment, and whatever follows this symbol on that line is ignored by MATLAB. For example:

```
>> % a full-line comment
>> x = 2 + 3  % and an in-line comment
x =
     5
```

4.4 Creating and Addressing Arrays

In this section we present the most basic use of arrays and vectors (one-dimensional arrays). For more information, refer to Chapter 6: *Arrays and Vectors* and, in particular, to Section 6.7, where useful array functions, special matrices and matrix multiplication are discussed.

Definition. An *array* is a collection of elements, arranged in one or more dimensions.

For our discussion here, we shall restrict our attention to the case when all the elements of an array are numbers.

4.4.1 Creating Arrays

The simplest of all arrays are *vectors*, i.e., one-dimensional arrays. They can be organized as rows or columns. A *row vector* is entered with its elements inside square brackets, and separated by blank spaces or commas:

```
>> r = [1, 0, 6, -1]
r =
     1     0     6    -1
```

To create a *column vector*, the elements are separated by semicolons:

```
>> c = [1; 0; 6; -1]
c =
     1
     0
     6
    -1
```

An "element" of a row (resp. column) vector can be another row (resp. column) vector, which allows us to create new vectors by *appending* one vector to another. For example:

```
>> r = [1, 0, 6, -1];
>> w = [4, 3, r, -2]
w =
     4     3     1     0     6    -1    -2
```

A *matrix* is a rectangular (two-dimensional) array. To create a matrix in MATLAB, we only have to follow the rules above, and separate elements in the same row by spaces (or commas) and one row from another by semicolons:

```
>> M = [1, 0; 6, -1]
M =
     1     0
     6    -1
```

An important operation is the *transpose* of an array (noted by an apostrophe ' following the object to be transposed), which interchanges rows and columns. For example:

```
>> T = M'
T =
     1     6
     0    -1
```

The transpose of a row vector generates a column vector, and *vice versa*:

```
>> c = [1, 0, 6, -1]'
c =
     1
     0
     6
    -1
```

4.4.2 Addressing Arrays and the Colon Notation

In many situations we need to extract or change individual elements elements of an array. Consider the matrix

$$A = \begin{pmatrix} 6 & 9 & 4 \\ 1 & 5 & 7 \end{pmatrix}$$

We create this array as follows:

```
>> A = [6,9,4;1,5,7]
A =
     6     9     4
     1     5     7
```

We can address any element of this array through its indices, that is, its row number and column number. For example, to extract the element positioned on the first row and third column:

```
>> A(1,3)
ans =
     4
```

If we want to replace the value of this element by a different one, we only have to assign the new value:

```
>> A(1,3) = 8
A =
     6     9     8
     1     5     7
```

Sometimes we need to address a group of elements, rather than individual elements. This is done most effectively in MATLAB through the use of the *colon notation*. The colon operator generates vectors with evenly spaced elements. Its general form is

```
>> x = m:q:n
```

where x is the vector created, whose elements are spaced with an increment q, and whose first value is m. The last value of the vector will be less or equal than n. If the increment q is omitted, a default value of 1 is used. When addressing elements of an array, a single colon (:) means "all the elements along this direction". Please study the following self-explanatory examples:

```
>> A(1,1:2)
ans =
     6     9
>> A(:,2)
ans =
     9
     5
>> A(:,1:2:3)
ans =
     6     8
     1     7
```

4.4.3 Element-by-Element Operations

One of the most powerful features of MATLAB is that arithmetic operations can be performed directly on entire arrays, rather than on their individual elements.

Scalar-Array Operations

The arithmetic operations + - * , acting between a scalar and an array, work "element-by-element", that is, they apply to all elements of the array. For example:

```
>> M = [1, 0; 6, -1]
M =
     1     0
     6    -1
>> M + 2
ans =
     3     2
     8     1
>> 2*M
ans =
     2     0
    12    -2
```

The same is true for the left and right division, / and \, if the scalar is in the denominator. For example:

```
>> M/2
ans =
    0.5000         0
    3.0000   -0.5000
```

Array-Array Operations

Element-by-element operations can also be performed between *arrays of exactly the same size*. To distinguish these element-by-element operations from other operations of linear algebra (matrix multiplication, solution to a system of equations, matrix exponentiation), the following new operators are introduced:

.* Array multiplication (instead of *, matrix multiplication)

./ Array right division (instead of /, right matrix division)

.\ Array left division (instead of \, solution of a system of equations)

.^ Array exponentiation (instead of ^, matrix power)

Here are a few examples:

```
>> A = [1, 0; 2, -3]
A =
     1     0
     2    -3
>> B = [4, -1; 2, 1]
B =
     4    -1
     2     1
>> A + B
ans =
     5    -1
     4    -2
>> A.*B
ans =
     4     0
     4    -3
>> B.^A
ans =
     4     1
     4     1
```

4.5 Files and Functions

4.5.1 Files

There are three types of files which are of interest to us:

1. m-files (ASCII): script with MATLAB commands.
2. MAT-files (binary): file with variables from the workspace.
3. Data files (ASCII): file with data arranged in rows and columns, typically including a header.

Please read the MATLAB help to learn how to save/load/import these files.

4.5.2 Mathematical Functions

These are your usual exponential, logarithmic, trigonometric and hyperbolic functions you get in a calculator. The difference is that they apply element-by-element to all components of an array, e.g.,

```
>> A
A =
     1     2
     3     4
>> log(A)
ans =
         0    0.6931
    1.0986    1.3863
>> log10(A)
ans =
         0    0.3010
    0.4771    0.6021
>> sqrt(A)
ans =
    1.0000    1.4142
    1.7321    2.0000
>> sin(A)
ans =
    0.8415    0.9093
    0.1411   -0.7568
```

Be careful with the following:

- Trigonometric functions operate in radian mode
- Decimal (base-10) logarithm is `log10(x)`

4.5.3 User-Defined Functions

- A *function file* is a particular type of m-file. Unlike a script file, all the variables are *local*.
- It is like subroutines in FORTRAN or BASIC, and functions in C.
- The first line looks as follows:

```
function [output vars] = function_name(input vars)
```

Example: The following function computes the volume of a cylinder:

```
% ----------------------------------------
% This function file (saved as cylinder.m)
% computes the volume of a cylinder.
%
% Input variables:
%   r = radius [L]
%   h = height [L]
%
% Output variables:
%   V = volume [L^3]
%
% Author: R. Juanes
% Last modified: 2001.01.30
%
function [V] = cylinder(r,h)
%
A = pi*r*r;     %   A = area of base [L^2]
V = A*h;
% ----------------------------------------
```

Here is a common sequence of MATLAB commands:

```
>> clear all
>> radius = 2;
>> height = 3;
>> [volume] = cylinder(radius,height)
volume =
   37.6991
>> whos
  Name         Size         Bytes  Class

  height       1x1              8  double array
  radius       1x1              8  double array
  volume       1x1              8  double array
```

Remarks:

- All the variables defined in the function file cannot be seen from outside.
- Variable names in the calling procedure do not need to match those defined inside the function.
- Variables can be defined to be `global`, although this is not recommendable.
- Functions can be nested, i.e., a function may call another function.

4.6 Basic Programming Tools

4.6.1 Relational Operators

The six relational operators are:

```
 <  <=  >  >=  ==  ~=
```

The result of a comparison between two variables is either 1 if the comparison is true, or 0 if the comparison is false. When applied to arrays, relational operators work element by element.

Example: Consider the price of the same list of books in two different bookstores:

```
>> price1 = [80, 67, 45, 112];
>> price2 = [75, 67, 52, 99];
```

Study the results of the following commands:

```
>> price1 < price2
ans =
     0     0     1     0
>> price1 >= price2
ans =
     1     1     0     1
>> price1 ~= price2
ans =
     1     0     1     1
```

4.6.2 Logical Operators

The three logical (Boolean) operators are:

`&` AND

`|` OR

`~` NOT

They are usually used together with relational operators.

Example: Consider the book prices above and the availability of the books in both bookstores (1 if it is in stock, 0 otherwise):

```
>> stock1 = [1, 0, 0, 1];
>> stock2 = [1, 1, 1, 0];
```

Books available in *both* bookstores:

```
>> stock1 & stock2
ans =
     1     0     0     0
```

Books available in *any* of the bookstores:

```
>> stock1 | stock2
ans =
     1     1     1     1
```

Books available in bookstore #2 at a price lower or equal than that of bookstore #1:

```
>> stock2 & (price2 <= price1)
ans =
     1     1     0     0
```

4.6.3 Decision-Making Structures

The if structure

The general form of the **if** structure is:

```
if (logical) condition 1
   do this
elseif condition 2
   do that
else
   default option: do something else
end
```

The **if** structures may be nested.

Example: Consider the following two-dimensional function, which is shown in **Figure 4.3** for $(x, y) \in [-1, 1] \times [-1, 1]$:

$$f(x,y) = \begin{cases} \sin(\pi x)\sin(\pi y) & \text{if } x > 0, y > 0, \\ 0 & \text{if } x < 0, y > 0, \\ -|xy| & \text{if } y < 0. \end{cases}$$

The following MATLAB function evaluates $f(x, y)$ using **if** statements:

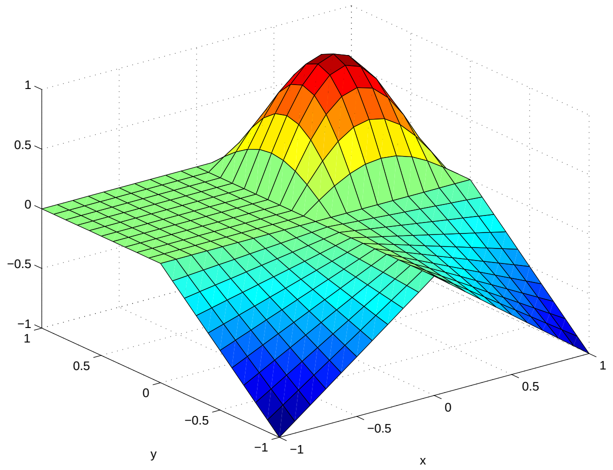


Figure 4.3: Matlab plot of the function $f(x, y)$

```
function [f] = pwfun(x,y)
%
if y < 0
    f = -abs(x*y);
else
    if x > 0
        f = sin(pi*x)*sin(pi*y);
    else
        f = 0;
    end
end
```

The **switch** structure

The general form of the **switch** structure is:

```
switch variable
case value 1
       do this
case value 2
       do that
otherwise
       default option: do something else
end
```

Example: The following function uses the **switch** construction to output a variable with the name of the day of the week:

```
function dayofweek = whatday(daynumber)
%           Output     = functionname(Input)
%
dayofweek = 'None of your business';
switch daynumber
case 1
  dayofweek = 'Monday';
case 2
  dayofweek = 'Tuesday';
case 3
  dayofweek = 'Wednesday';
case 4
  dayofweek = 'Thursday';
case 5
  dayofweek = 'Friday';
case 6
  dayofweek = 'Saturday';
case 7
  dayofweek = 'Sunday';
otherwise
  %dayofweek = '';
  fprintf('daynumber is invalid')
end
```

The function produces the following output:

```
>> whatday(3)
ans =
Wednesday
>> whatday(8)
daynumber is invalid
ans =
     ''
```

4.6.4 Loops

A *loop* is a structure for repeating a calculation a number of times. There are two classes of loops in MATLAB:

for loop: when the number or iterations is known.

while loop: when the process terminates if a specified condition is satisfied.

for loop

Assume we want to compute the series

$$S = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6}.$$

Of course, only a sum with a finite number of terms can be calculated. The following MATLAB function will compute it with a specified number of terms n:

```
function [S] = pi2over6(n)
%
% initialize sum
S = 0;
%
% loop
for i = 1:n
    S = S + 1/i/i;
end
```

Compare the results for different values of n:

```
>> pi*pi/6
ans =
   1.64493406684823
>> pi2over6(10)
ans =
   1.54976773116654
>> pi2over6(100)
ans =
   1.63498390018489
>> pi2over6(1000)
ans =
   1.64393456668156
>> pi2over6(10000)
ans =
   1.64483407184807
```

Remarks:

1. The increment of the iteration variable may be positive or negative.

2. It is highly recommended to use integers for the iteration variable.

while loop

Imagine that for the series above we want to specify a measure of the error (given by the last computed term), rather than the number of terms. A **while** loop is more convenient:

```
function [S,n] = pi2over6w(err)
%
% initialize sum and first term
S = 0; n = 1; term = 1;
%
% loop
```

```
while term > err
    term = 1/n/n;
    S = S + term;
    n = n+1;
end
```

Compare the output for different specified errors:

```
>> pi*pi/6
ans =
   1.64493406684823
>> [S,n] = pi2over6w(1e-1)
S =
   1.42361111111111
n =
     5
>> [S,n] = pi2over6w(1e-3)
S =
   1.61416726282792
n =
    33
>> [S,n] = pi2over6w(1e-5)
S =
   1.64178446315268
n =
   318
>> [S,n] = pi2over6w(1e-7)
S =
   1.64461796127160
n =
        3164
```

Avoiding loops

In modern programming, one should avoid loop structures, by using the so-called *implied loops* or *vectorized code*. Every time you remove a loop, your code will be (often, but not always):

1. shorter,
2. easier to understand,
3. faster,
4. more *object oriented*.

Most MATLAB functions work for arrays, so you don't have do your calculations element by element (viz. matrix multiplication). Remember, almost everything can be vectorized.

Example: Here is a vectorized implementation of the approximate series considered in the previous examples.

```
function [S,n] = pi2over6v(err)
%
% compute number of terms needed
n = ceil(sqrt(1/err));
%
% vector of terms
term = 1:n; term = 1./term./term;
%
% series
S = sum(term);
```

And here is the output:

```
>> pi*pi/6
ans =
   1.64493406684823
>> [S,n] = pi2over6v(1e-7)
S =
   1.64461796127160
n =
        3163
```

Example: Recall the function in Figure 4.3. Here is the naïve way to generate it (using the function `pwfun.m` and **for** loops):

```
x = -1:.1:1; y = x;
%
% loop to compute function
for ix = 1:length(x)
    for iy = 1:length(y)
        xx = x(ix);
        yy = y(iy);
        f(ix,iy) = pwfun(xx,yy);
    end
end
%
% plot
figure;
surf(x,y,f')
```

And here is the vectorized code using array operations, which runs orders of magnitude faster. (Can you understand it?)

```
x = -1:.1:1; y = x;
%
```

```
% compute function
f = pwfun2(x,y);
%
% plot
figure;
surf(x,y,f')
% ---------------------------------------
function [f] = pwfun2(x,y)
%
f = - abs(x'*(y.*(y<0))) ...
    + sin(pi*x.*(x>0))'*sin(pi*y.*(y>0));
```

4.7 Output of Results

4.7.1 Formatted Text Output

The **fprintf** command allows one to control the output format. For a detailed description of this command, please go to the online MATLAB help. The syntax is:

```
fprintf(fid,format,A,...)
```

fid is an integer file identifier obtained from **fopen**. If omitted, the default value of 1 is used, and the output is displayed on the screen.

format is a string with the output format specifications.

A,... are the variables to be displayed. They can be scalars, vectors or matrices. If **A** is a matrix, its transpose will be displayed.

Example 1:

```
tol = 0.01;
maxiter = 200;
fprintf(' Warning!\n')
fprintf(' Tolerance of %6.3f not met within %g iterations\n',...
        tol,maxiter)
```

will produce

```
Warning!
Tolerance of  0.010 not met within 200 iterations
```

Example 2:

```
x = 0:.1:1;
y = exp(x);
fprintf('    x       exp(x) \n')
fprintf(' -----  -----------\n')
fprintf('%6.2f %12.8f\n',[x;y])
```

will produce

```
    x       exp(x)
 -----  -----------
  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

Example 3:

```
tolfrac = [0.01 0.05 0.1 0.2];
tolperc = tolfrac*100;
tolmat = [tolfrac;tolperc];
fprintf(' Tolerance of %12.3e is equal to %4.0f%%\n',tolmat)
```

will produce

```
 Tolerance of   1.000e-002 is equal to    1%
 Tolerance of   5.000e-002 is equal to    5%
 Tolerance of   1.000e-001 is equal to   10%
 Tolerance of   2.000e-001 is equal to   20%
```

4.7.2 Basic Plotting

Here is an example that shows the most basic features of plotting with MATLAB. The code below generates **Figure 4.4**.

```
%-------------------------------------------------------------
% bplot.m: the most basic plot in MATLAB
% Author: Ruben Juanes
% Date: 2/13/01
%-------------------------------------------------------------
clear all
close all
%
% parameters
%
x_min = 0;
x_max = 5;
x_inc = 0.1;
```

```
%
% vectors
%
x = [x_min:x_inc:x_max];
y = sin(x);
%
% plot
%
plot(x,y)
xlabel('x','FontSize',14)
ylabel('y','FontSize',14)
title('y = sin(x)','FontSize',18)
grid
%
```

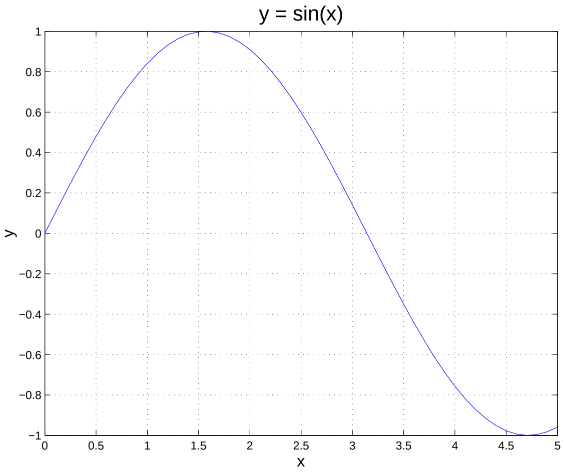


Figure 4.4: Matlab plot of the function $y = \sin(x)$

Chapter 5

Introduction to Computers

5.1 What Are You Going to Learn?

In its most rudimentary form a computer is a device that remembers what it counts, counts what it remembers, and retrieves whatever is stored in it with a push of a button or a click of a mouse. This chapter will be an introduction to computers. First, you will refresh your high school knowledge of numbers, powers and logarithms. Here I will follow closely the introduction by DONALD E. KNUTH [20]. Second, you will learn what are the simplest means of instructing a computer to do something, and how the computer executes your instructions. To illustrate the principles of instructing the computer, I will use "the file clerk model" of RICHARD P. FEYNMAN [12].

Chapter 5 folder on fttp:\\petroleum.berkeley.edu has the following MATLAB files:

Add.m Compare two strings

AddSales.m Add sales from several salesmen, by calling `Add.m`.

Each file contains a MATLAB *function* with the same name.

5.2 Why is it Important?

When you are an engineer or a scientist, you must know what a computer can or cannot do. You must also know how to instruct a computer to do simple things. MATLAB makes your instructions particularly simple.

5.3 What is Computer?

You often hear statements similar to this one: "The United States has become an information society dependent on the creative use and communication of information for its economic and social well-being[1]". The National Committee on Excellence in Education agrees[2] and recommends that all students in the U.S. be put through at least half year of study in computer science. With all this hoopla, you may start wondering what is so special about computers that gives them such a central role in our society. Why are all people divided into two categories, those who are fluent with computers and those who are not?

Computers can do many things. They add millions of numbers faster that you can blink an eye. They guide smart bombs to their targets. They operate pumps at gas stations. They reserve tickets to rock concerts and to movie theaters. They check you out at a grocery store. They keep all your medical, tax and financial records. They connect you to the "Information Highway" or the World Wide Web. They allow me to type in this lecture. They allowed Enron Corporation to reach in the year 2001 $100 billion in stock value, and their misuse caused Enron's instantaneous and catastrophic failure (yes, computers operated by the crooks also work fast). Obviously computers are important to every aspect of life in America. So if we are going to talk about computers, we should decide which ones we are going to look at and how.

In fact, it makes no sense to concentrate on individual machines. The reason is that once you get down to the guts of computers, you will find out that they tend to be more or less alike. They can differ in their functions, and in the nature of their inputs and outputs (I/O): one computer can produce music, another one stunning graphics; while one can be run from a keyboard, another one runs from a touch screen. In essence all these computers are very similar. We will hence dwell on what their innards can or cannot do. Furthermore we will not assume anything specific about their Input/Output, I/O, structure; all we care is that however the input gets in, it is in digital form, and whatever happens to the output, the last the innards see it, it is digital too. By digital we will mean binary numbers: 1's and 0's.

MATLAB is very good at handling I/O, and producing very high quality graphs on the fly. Because of the ease of outputting anything we will do, we will have more time to concentrate on the code implementation details.

5.4 Numbers, Powers and Logarithms

Before we embark on the study of *Matlabese*[3], the official language of E77N, a short reminder of the numbers we will deal with is in order. Here, I will follow almost literally Section 1.2 in KNUTH's monograph [20].

[1] The U.S., Congress Office of Technology Assessment, *Information Technology and Its Impact on American Education*, Washington, D.C.: Government Printing Office, 1982.

[2] *A Nation at Risk*, Washington, D.C.: Government Printing Office, 1983.

[3] Or should we call it *Matlabish*?

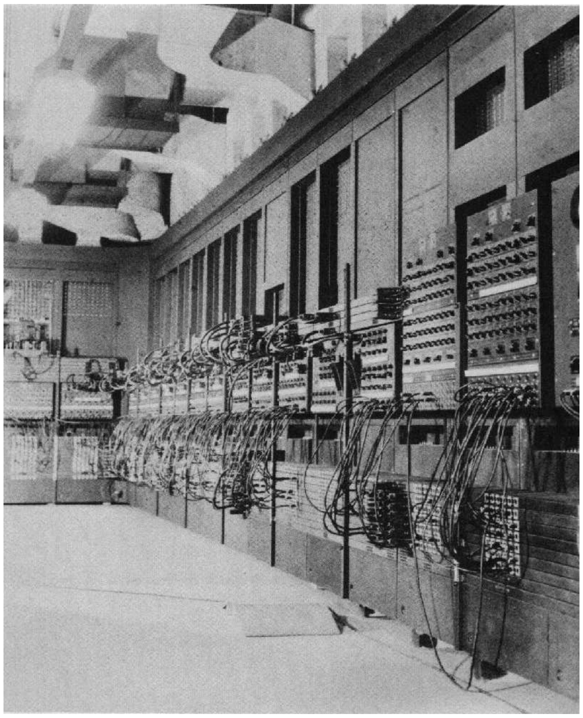


Figure 5.1: 1950: Not all computers have their programs in their memory. ENIAC and Colossus, the first electronic computers, were programmed by placing patch wires into plugboards. The tangles of wires in the picture are the program. (Only part of ENIAC is in the picture - it was 100 feet long.)

5.4.1 Numbers

1. **An integer** is the whole number, negative, zero, or positive:

$$\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \ldots \tag{5.1}$$

2. **A rational number** is the ratio (quotient) of two integers:

$$\frac{p}{q}, \quad \text{where } q \text{ is positive.} \tag{5.2}$$

3. **A real number**, x, can always be expressed as a sum of an integer, n, and a *decimal expansion*:

$$0.d_1 d_2 d_3 \ldots \tag{5.3}$$

$$x = n + 0.d_1 d_2 d_3 \ldots \tag{5.4}$$

where each digit d_i is an integer between 0 and 9, and the sequence of digits does *not* end with infinitely many 9s. The representation (5.4) means that

Figure 5.2: 1960: The IBM 360 was a revolutionary advance in computer system architecture, enabling a family of computers covering a wide range of price and performance.

Figure 5.3: 1990: Cray Y-MP/432, a four-processor supercomputer with 256 MB of central memory.

any real number can be bracketed by rational numbers:

$$n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} \le x < n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} + \frac{1}{10^k} \quad (5.5)$$

for all positive integers k. The bracketing procedure (5.5) will play an important role in the finite computer representation of real numbers.

4. **A complex number**, z, has the form:

$$z = x + iy \tag{5.6}$$

where x and y are real and i is a special number that satisfies the equation $i^2 = -1$. We call x and y the *real* and *imaginary* parts of z and we define the absolute value of z as

$$|z| = \sqrt{x^2 + y^2} \tag{5.7}$$

The *complex conjugate* of z is $\bar{z} = x - iy$ and, from the definition of i, $z\bar{z} = x^2 + y^2 = |z|^2$

Remark 4 A *decimal*, or *base-10* ten, number is a series of powers of ten. For example, $10 = 1 \times 10^1 + 0 \times 10^0$, $0.1 = 0 \times 10^0 + 1 \times 10^{-1}$.

A *binary*, or *base-2*, number is a series of powers of two. For example, the binary 1 is 1×2^0, the binary $11 = 1 \times 2^1 + 1 \times 2^0$.

A decimal number can be converted in to a binary one and *vice versa*. However, a nice rational number in one base may become a nasty real number in the other one. The decimal 2 is $1 \times 2^1 + 0 \times 2^0$, or the binary 10; the decimal 3 is $1 \times 2^1 + 1 \times 2^0$, or the binary 11; the decimal $0.5 = 1/2$ is $0 \times 2^0 + 1 \times 2^{-1}$, or the binary 0.1.

Question: What is the decimal 10 in binary? What is the binary equivalent of the decimal 0.1? □

5.4.2 Powers

If b is a *positive real* number and n is an integer, then b^n is defined using the old tried rules:

$$b^0 = 1, \quad b^n = b^{n-1}b \text{ if } n > 0, \quad b^n = b^{n-1}/b \text{ if } n < 0. \tag{5.8}$$

Soon we will be able to prove by *induction* that the *laws of exponents* hold:

$$b^{n+m} = b^n b^m, \quad (b^n)^m = b^{mn}, \tag{5.9}$$

where n and m are integers.

For rational numbers, $r = p/q$, we define:

$$b^r = b^{p/q} = \sqrt[q]{b^p}. \tag{5.10}$$

Finally, we define b^x for real exponents x given by Eq. (5.4):

$$b^{n+d_1/10+d_2/100+\cdots+d_k/10^k} \leq b^x < b^{n+d_1/10+d_2/100+\cdots+d_k/10^k+1/10^k} \tag{5.11}$$

When $b < 1$, we define b^x as

$$\begin{aligned} b^x &\equiv (1/b)^{-x} \\ (1/2)^2 &= 2^{-2} = 1/4 \end{aligned} \tag{5.12}$$

When $b = 1$, $b^x = 1$ for any x.

5.4.3 Logarithms

Now we come to an important question. Suppose that a positive real number y is given; can we find a real number x such that $y = b^x$, $b \neq 1$? The answer is yes, for we simply use Eq. (5.11) in "reverse" to determine the decimal expansion of x, i.e., n, d_1, d_2,...d_k, when $b^x = y$ is given. The resulting number x is called the *logarithm of y to the base of b*, and we write it as

$$x = \log_b y \tag{5.13}$$

By this definition we have

$$x = b^{\log_b x} = \log_b(b^x) \tag{5.14}$$

For example,

$$10^{0.3102999} = 1.9999999739\ldots, \quad \log_{10} 2 = 0.3102999\ldots. \tag{5.15}$$

From the laws of exponents it follows that

$$\log_b(xy) = \log_b x + \log_b y, \quad \text{if} \quad x > 0,\ y > 0 \tag{5.16}$$

and

$$\log_b\left(c^y\right) = y \log_b c. \tag{5.17}$$

Equation (5.15) illustrates the so-called *common* or *base-10* logarithm. This is *not* the most convenient logarithm to work with. There is a real number $e = 2.718281828459\ldots$, for which the logarithms have simpler properties. Logarithms to the base e are conventionally called *natural logarithms*, and denoted as:

$$\ln x = \log_e x. \tag{5.18}$$

5.5 Talking to a Computer

For today's computers to perform a complex task, they need a precise and complete description of how to do that task in terms of a sequence of simple procedures, called the "software," and we need a machine to carry out these procedures in a specifiable order, the "hardware." Our instructions must be exact and unambiguous. In real life we never tell each other *exactly* what we want to say. Such excruciating exactness is never needed; the context, body language, familiarity with the speaker allow us to *fill in the gaps* and resolve any ambiguities in what we said. Computers are really dumb and can never guess what is being said, the way a person does. Our specific instructions are communicated to a computer through a *programming language*. As there are many human languages and alphabets, there are many computer languages.

Let us begin to examine how we might build complex instructions from a set of simple elements. If an instruction set B (this abbreviation is just for convenience) is very simple, then a complex instruction is going to take an awful lot of description, and the resulting *programs* will be very long, complicated and almost unreadable to anyone but the author. For example, we may want to perform all sorts of numerical calculations on our computer, but we have at our disposal only a set B which does not recognize multiplication. If we tell this computer to multiply 3 by 33, it returns error. Suppose now that B does have addition; if you think about it, you will see that we can emulate multiplication by adding lots of times. In this example we add 33 to itself two more times. If we need multiplication often, we can shorten the writing of B-programs if we augment the set B with a separate "multiply" instruction, defined as a chunk of basic B instructions that are required to preform multiplication. Then when we want to perform multiplication, we say "computer, 3 times 33", and it now recognizes the word "times" and uses addition to accomplish this task. The computer still breaks compound instructions into their basic components, but from now on we avoid getting bogged in the low-level details all the time. A very similar process takes place in everyday life when one replaces with one word a set of ideas and the connections among them. From then on, referring to these ideas we can use a single word and do not have to go back and work through all the lower level concepts. For example, think how many concepts are summarized with these two words, "derivative" and "integral." We no longer go to the lower level when we say the derivative of x^2 is $2x$ or the integral of $\cos x$ is $\sin x + C$.

Remark 5 Computers are such complex machines that a simplification of ideas, like the ones above, is usually necessary, and good design is essential if you want to avoid getting completely lost in details. In short, good programming practice is a must.

The word "computer" makes you think about arithmetic: add, subtract, multiply, divide and so on, and it is easy to assume that this is all a computer does. In fact this is not entirely true, but most conventional computers have only one place do do arithmetic (their CPU) and the rest of the machine is for the computer's main task which is to shuffle bits of paper around, only in the computer bits of paper are electronic signals. Most computers are reminiscent of and army of file clerks, dashing back and forth to their file cabinets, taking files out and putting them back, scribbling notes on pieces of paper and passing them on; erasing their notes, and so on. The file clerk metaphor is a good place to start to get to you some of the basic elements of computer structure.

5.6 The File Clerk Model

This model was first used by Richard Feynman, the great physicist and Nobel prize laureate, in his seminal lectures on computation [12]. Suppose that you have a big company, employing lots of salesmen[4]. Lots of information about these salesmen is stored in a big filing system, which is administered by a *single clerk*. We begin with the idea that the clerk knows how get the information from the filing system. The data are stored on cards, and each card has the name of the salesman, his location in a given state, the sum total of sales he has made, his salary, etc., see **Figure 5.4**.

Salesman: ....
Sales: ....
Salary: ....
Location: ....
.......: ....

Figure 5.4: Records on a salesman data card.

Now suppose that we want to answer a specific question: What are the total sales in California? This is a pretty simple question, and that is why we start from it. To find the sales in California, our clerk could do the following:

```
Take out a card
If the ‘‘location’’ says California, then
    Add the number under ‘‘sales’’ to a running counter
    called ‘‘total’’
Put ‘‘sales’’ card back
Take next card and repeat
```

You have to keep repeating this procedure until you've gone through all the cards.

No let's suppose that we have hired a particularly stupid clerk, who can still read, but does not know how to keep a running count. We need to help this poor soul a little more by inventing the “total card for our clerk to use. He will use our new procedure to keep a running total in the following way:

```
Take out a next ‘‘sales’’ card
If California, then
    Take out ‘‘total’’ card
    Add sales number to number on card
```

[4] Or *saleswomen*, but this is irrelevant here.

```
   Put ‘‘total’’ card back
Put ‘‘sales’’ card back
Take next ‘‘sales’’ card and repeat
```

Of course this is a very crude solution to a computer adding problem. Obviously, the data would not be stored on cards, and the machine would not have to “take out a card”. Instead it would read the stored information form a register. It would also write from a register to a “card” without physically putting something back.

In MATLAB, the latter procedure could be implemented as follows:

```
function [Total] = Add(Salesman, Sales, Salary, Location, Total)
%            ^      ^                        ^
%            |    Function name              |
%        Output            Input variables: we cannot change their values
%
% Written by T.W. Patzek, January 15, 2002
% E77N Class Material, University of California, Berkeley
%-------------------------------------------------------------------

% Compare strings to one another
if strcmp(Location,'CA') == 1
   Total = Total+Sales;
end
```

The code fragment above requires some explanation.

- First of all, the whole procedure of finding the total of sales of all salesmen located in California is gathered in one place, which in MATLAB has a reserved *keyword*, function, and is stored in a file with the same name, i.e., Add.m. To type in the contents of this file, we click on File, New, and M-file in MATLAB command window. When done, we store our new “m-file” Add.m in a directory of our choice. We change MATLAB path to that directory, and we invoke the function by typing in, e.g., Total = 0 ↰ Total = Add('Dick', 400000, 40000, 'CA', Total) ↰.
- Each field of every card corresponds to a single register, and is called a *variable* because we can overwrite the contents of this register as needed.
- Each function may have a list of *output* variables or *arguments*, here a single argument Total in square parentheses to the left of the function’s unique name, here Add, and a list of *input* arguments in round parentheses, here the fields of a single “sales” card, i.e.,, Salesman, Sales, Salary and Location. The last input argument is the “total” card with the previous value of the total sales, or the input variable Total.
- We can only look at the input variables, but we cannot change their values.
- We can change the values of the output variables, hence the variable Total appears both as input and output.

- Because strings are not numbers and have variable lengths, we must use a special MATLAB function strcmp to compare them. If the first and second argument of strcmp are identical, it returns 1, otherwise it returns 0. Note that the logical "=" is denoted as "==" to differentiate it from the variable assignment "=".

- In the statement Total=Total+Sales, we update the output register Total by adding the contents of register Sales to a number already stored in it. We follow here the "right-to-left" convention used in all standard computer languages, i.e., the variables on the right must have values assigned to them, and the result is stored to the left of the "=" sign[5].

- Later on, you will learn that in fact MATLAB stores each variable in an array of contiguous registers. This is because *every* variable in MATLAB is treated as an array with some dimensions. So a single number is an array with one element, and in that case it does fill a single register that is 64 bits long.

To process many cards in MATLAB, we could proceed as follows:

```
function AddSales()
%-------------------------------------------------------------------
% Written by T.W. Patzek, January 15, 2002
% E77N Class Material, University of California, Berkeley
%
% Note that this function has no input or output arguments
% We use this function to isolate what we do from the outside
% world
%-------------------------------------------------------------------

% Card 1
Salesman1 = 'Dick';
Sales1    = 120000;
Salary1   = 35000;
Location1 = 'CA';

% Card 2
Salesman2 = 'Harry';
Sales2    = 220000;
Salary2   = 45000;
Location2 = 'UT';

% Card 3
Salesman3 = 'John';
Sales3    = 820000;
```

[5] The assignment operator should be denoted by ←, i.e., the statement $x \leftarrow n + 1$ should read: "the value of variable x is replaced by the current value of variable n and a constant 1 is added". In computer languages this statement is written as $x = n + 1$, mostly because there is no left arrow on our keyboards. A statement $n \leftarrow n + 1$ is read as "n gets n+1" or "n is replaced by $n + 1$".

```
Salary3   = 95000;
Location3 = 'CA';

% Prepare a register to hold the total sales in Califonia
Total = 0;
%Process first card
[Total] = Add(Salesman1, Sales1, Salary1, Location1, Total)
%Process second card
[Total] = Add(Salesman2, Sales2, Salary2, Location2, Total)
%Process third card
[Total] = Add(Salesman3, Sales3, Salary3, Location3, Total)
```

In practice, the individual cards will be stored in a relational database or in a text file and will be read into an array of *cells* and then *structures*. In a couple of lectures you will learn how to do it.

5.7 Exercises

1. What is the smallest positive rational number?

2. Is $1 + 0.239999999\ldots$ a decimal expansion?

3. What is $(-3)^{-3}$?

4. What is $(0.125)^{-2/3}$?

5. Using Eqs. (5.14) and (5.9), prove Eqs. (5.16) and (5.17).

6. We shall define $\lg x \equiv \log_2 x$, following a suggestion of Edward M. Reingold [20]. Show that

   $$\log_{10} x = (\lg x)(\log_{10} 2), \tag{5.19}$$

   and in general that

   $$\log_c x = \frac{\log_b x}{\log_b c}. \tag{5.20}$$

7. (Knuth [20]) In Eq. (5.5) we defined real numbers in terms of decimal expansion. Discuss how we could have defined these numbers in terms of binary expansion and replace Eq. (5.5) with its binary equivalent.

8. Decimal fraction, 0.1[decimal], is approximated by the periodic binary fraction $0.00011001100110011\cdots = 0.0(0011)$[binary]. Open the MATLAB window and by typing:

   ```
   >> x = 2^(-4)+2^(-5)
   x =
      0.09375000000000
   ```

etc., calculate the relative error, $y = (x - 0.1)/0.1$, between the 5-digit and 10-digit binary expansions (not counting "0.") of 0.1[decimal], and 0.1[decimal]. Then go back to Item 7 and think again if necessary.

9. Explain in words, what a MATLAB function does, what are the properties of the input and output arguments, and how and where do we place them?

10. Type in the function Add.m, store it, and invoke it properly from the MATLAB command window. Repeat with a couple of different input arguments.

11. Suppose that we would like to pay a 5% commission on sales of all salesmen in California. How would you expand function Add.m and its calling function AddSales.m to accomplish this task?

12. As in the previous item, but now you want to pay a 7.5% commission to salesmen from Utah.

5.8 Summary

In this chapter we already got a bit technical. We talked a little about the types of numbers used in computer applications, and about the only numbers computers understand: the *binary numbers*. Using the *clerk analogy*, we explained how a computer processes simple instructions. Most importantly, we stressed the importance of packaging your instructions in such a way that the next guy does not have to go all the way to the lowest level to extend what you have already done. You should strive to develop a good programming practice from the very beginning. So, please, *no* MATLAB scripts, only MATLAB functions; and *no* MATLAB do-loops, only vector operations!

Chapter 6

Arrays and Vectors

6.1 What Are You Going to Learn?

In this chapter, you are going to learn about multidimensional *arrays* and one-dimensional arrays, loosely called "vectors," and their geometrical counterparts, *matrices* and true *vectors*. You will learn how to rotate a two-dimensional vector and how to use *matrix notation* to shorten the lengthy algebra. We will also revisit the standard notation used in this book.

The Chapter 6 folder on fttp:\\petroleum.berkeley.edu has the following MATLAB programs:

threedarray.m generates a plot of a three-dimensional MATLAB array.

vectoralgebra.m defines a pair of two-dimensional vectors issuing from the origin, plots them, calculates their scalar product, lengths, etc.

myarrow.m plots a a single vector with an arrow of variable shape. Uses extensively vector translation and rotation two calculate the coordinates of the vector elements

6.2 Why Is It Important?

Arrays and the operations on them are *the* essential building blocks of MATLAB. You can't learn MATLAB without mastering its arrays first.

6.3 Arrays and Vectors

We start from a definition of *array*, the most fundamental MATLAB object.

Definition 1 Array:

- An arrangement of elements in one or more dimensions.

- In a programming language, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripts. □

Remark 6 A two-dimensional rectangular array has ordered rows and columns of objects (e.g., numbers). An m-by-n array has m rows of n objects per row. The first index, $i = 1, 2, \ldots, m$ denotes the row number, and the second one, $j = 1, 2, \ldots, m$, the column number in the array. □

An $m \times n$ array, A, is written as

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1,n} \\ a_{21} & a_{22} & \ldots & a_{2,n} \\ \vdots & \vdots & \ldots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}. \tag{6.1}$$

Remark 7 A three-dimensional array is an orthogonal parallelepiped of ordered objects, whose ordered (e.g., horizontal) planes are two-dimensional arrays, **Figure 6.1**. □

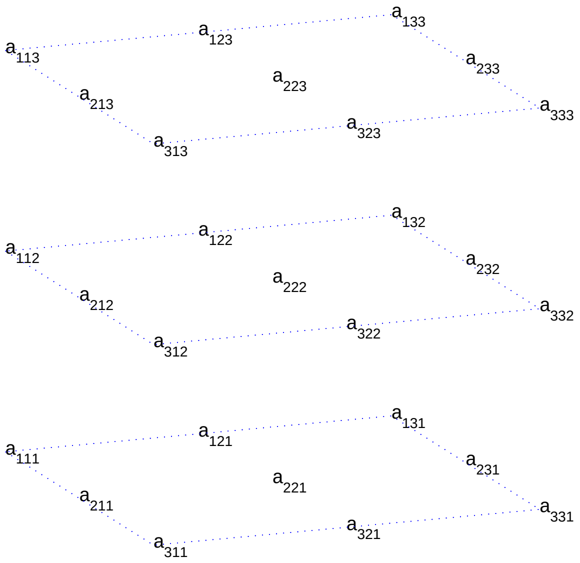


Figure 6.1: A 3×3×3 array, $A = [a_{ijk}]$, whose elements are ordered as horizontal arrays. In MATLAB, each element of this array can be (almost) anything: a real number, a complex number, a string, a cell, a structure, or a user-defined object. Isn't this cool?!

So, in fact, a row "vector" is equivalent to a single-row array:

$$X = (x_{11}, x_{12}, \ldots, x_{1n}), \tag{6.2}$$

and a column “vector” is equivalent to a single-column array:

$$Y = \begin{pmatrix} y_{11} \\ y_{21} \\ \vdots \\ y_{m1} \end{pmatrix}. \tag{6.3}$$

An $m \times n$ array can be thought of as m row vectors of length n, or as n column vectors of length m.

We often skip the constant first or second index and write:

$$X = (x_1, x_2, \ldots, x_n), \tag{6.4}$$

and

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}. \tag{6.5}$$

Remark 8 *Single*-row or single-column arrays are loosely called *vectors*, but in general they are *not*. □

Definition 2 Physically, a *vector* is an object characterized by length (magnitude) and direction. Velocity, acceleration, force, etc., can all be visualized as vectors. An algebraic aspect of a vector is suggested by the one-to-one correspondence between the unique vectors (issuing from the origin of, say, a Cartesian coordinate system, **Figure 6.2**), and the coordinates of their terminal points, the ordered pairs of numbers, (x, y), in two dimensions, or three numbers, (x, y, z) in three dimensions. In higher-dimensional Euclidean spaces, this analogy is extended to an ordered *n-tuple* of numbers, $(x_1, x_2, x_3, \ldots, x_n)$. We can no longer visualize these *n-dimensional* vectors, but the definition works. Of course there is more to a vector than just n numbers. For these numbers to represent a *vector object*, they must change appropriately (transform) when we rotate the coordinate system. □

We follow with the informal definition of a matrix:

Definition 3 *Matrix* is a two-dimensional rectangular table of ordered rows and columns of numbers, and a transformation rule for every number when we rotate the coordinate system. □

For example, a *row* vector x, of *length* n, is written as

$$x = (x_1, x_2, \ldots, x_n), \tag{6.6}$$

and a *column* vector y, of *length* m, as

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}. \tag{6.7}$$

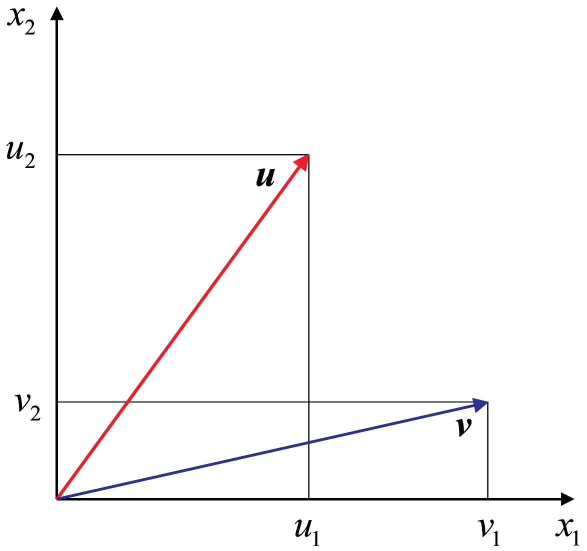


Figure 6.2: Coordinates of two unique vectors in 2D. Note that a Cartesian coordinate system is x_1 and x_2, not x and y.

The same vector can be written in the column and the row form; one is the *transpose* of the other:

$$
\begin{aligned}
x &= (x_1, x_2, \dots, x_n) \\
x^T &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}
\end{aligned}
\tag{6.8}
$$

Remark 9 All vectors and matrices are arrays, but not all one- or two-dimensional arrays are vectors or matrices. □

6.4 Scalar product

The scalar or "inner" or "dot" product of two vectors is the real number defined by the equation

$$\boldsymbol{x} \cdot \boldsymbol{y} = |x||y| \cos\theta \tag{6.9}$$

where θ is the angle between the two vectors measured from $\boldsymbol{x}$ to $\boldsymbol{y}$. Geometrically, the scalar product is equal to the length of vector $\boldsymbol{x}$ multiplied by the length of projection of vector $\boldsymbol{y}$ onto vector $\boldsymbol{x}$. Of course $\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{y} \cdot \boldsymbol{x}$. In particular, the length of a vector $\boldsymbol{x}$ is:

$$
\begin{aligned}
\boldsymbol{x} \cdot \boldsymbol{x} &= |x||x| \\
|x| &= \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}}
\end{aligned}
\tag{6.10}
$$

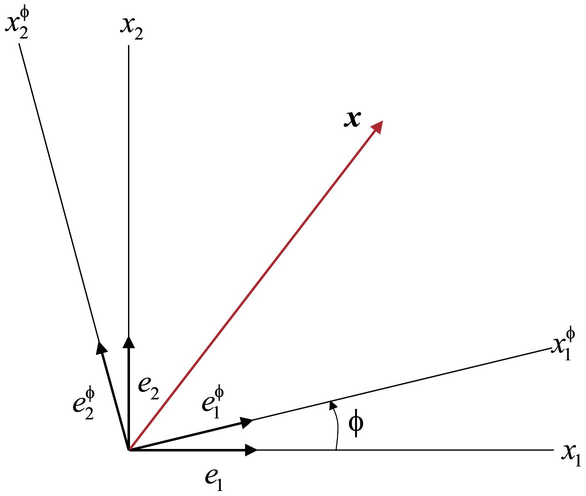


Figure 6.3: Counterclockwise rotation in two dimensions. The vector $\boldsymbol{x}$ sits passively as the basis with respect to which it is expressed is rotated beneath it [3].

Example 1 Suppose that we resolve two arbitrary vectors issuing from the origin, $\boldsymbol{x}$ and $\boldsymbol{y}$, into Cartesian unit vectors, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, such that $\boldsymbol{e}_1 \cdot \boldsymbol{e}_1 = 1$, $\boldsymbol{e}_2 \cdot \boldsymbol{e}_2 = 1$, $\boldsymbol{e}_1 \cdot \boldsymbol{e}_2 = 0$. Then

$$\begin{aligned}
\boldsymbol{x} &= x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2 \\
\boldsymbol{y} &= y_1\boldsymbol{e}_1 + y_2\boldsymbol{e}_2 \\
\boldsymbol{x}\cdot\boldsymbol{y} &= x_1y_1(\boldsymbol{e}_1\cdot\boldsymbol{e}_1) + x_2y_2(\boldsymbol{e}_2\cdot\boldsymbol{e}_2) + x_1y_2(\boldsymbol{e}_1\cdot\boldsymbol{e}_2) + x_2y_1(\boldsymbol{e}_2\cdot\boldsymbol{e}_1) \\
\boldsymbol{x}\cdot\boldsymbol{y} &= x_1y_1 + x_2y_2 + 0 + 0 \qquad (6.11) \\
|x| &= \sqrt{x_1^2 + x_2^2} \qquad \square \\
|y| &= \sqrt{y_1^2 + y_2^2}
\end{aligned}$$

A rudimentary MATLAB implementation of vector algebra is listed Section 6.9, see also **Figure 6.4**:

6.5 Rotation

As we said before, there is a relationship between the components of a vector expressed with respect to two different Cartesian bases with the same origin. A Cartesian base $(\boldsymbol{e}_1, \boldsymbol{e}_2)$ is rotated counterclockwise by an angle ϕ, and denoted as $(\boldsymbol{e}_1^\phi, \boldsymbol{e}_2^\phi)$, **Figure 6.3**. The same vector $\boldsymbol{x}$ can be resolved with respect to the

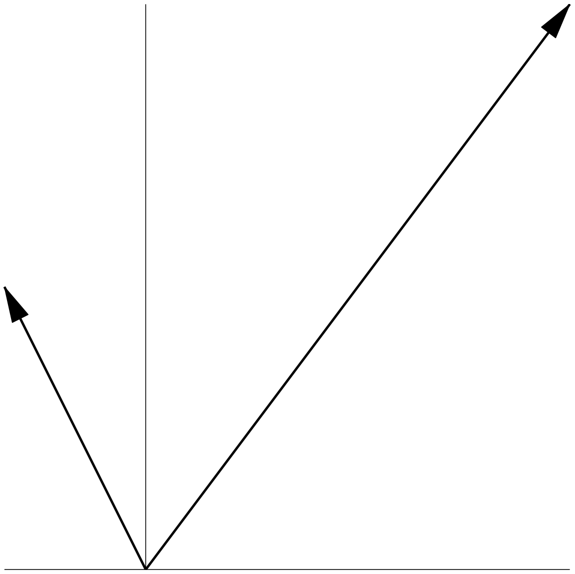


Figure 6.4: Output of function vectoralgebra.m.

two bases:

$$\begin{aligned} \boldsymbol{x} &= x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2 = (\boldsymbol{x}\cdot\boldsymbol{e}_1)\boldsymbol{e}_1 + (\boldsymbol{x}\cdot\boldsymbol{e}_2)\boldsymbol{e}_2 \\ \boldsymbol{x} &= x_1^\phi\boldsymbol{e}_1^\phi + x_2^\phi\boldsymbol{e}_2^\phi = (\boldsymbol{x}^\phi\cdot\boldsymbol{e}_1^\phi)\boldsymbol{e}_1^\phi + (\boldsymbol{x}^\phi\cdot\boldsymbol{e}_2^\phi)\boldsymbol{e}_2^\phi \end{aligned} \tag{6.12}$$

In particular, we can resolve the rotated unit vectors $\boldsymbol{e}_1^\phi$ and $\boldsymbol{e}_2^\phi$ with respect to the original unit vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. To do so, set $\boldsymbol{x} = \boldsymbol{e}_1^\phi$ and $\boldsymbol{x} = \boldsymbol{e}_2^\phi$, and use Eq. $(6.12)_1$ twice:

$$\begin{aligned} \boldsymbol{e}_1^\phi &= (\boldsymbol{e}_1^\phi\cdot\boldsymbol{e}_1)\boldsymbol{e}_1 + (\boldsymbol{e}_1^\phi\cdot\boldsymbol{e}_2)\boldsymbol{e}_2 = \cos[\angle(\boldsymbol{e}_1^\phi,\boldsymbol{e}_1)]\boldsymbol{e}_1 + \cos[\angle(\boldsymbol{e}_1^\phi,\boldsymbol{e}_2)]\boldsymbol{e}_2 \\ \boldsymbol{e}_2^\phi &= (\boldsymbol{e}_2^\phi\cdot\boldsymbol{e}_1)\boldsymbol{e}_1 + (\boldsymbol{e}_2^\phi\cdot\boldsymbol{e}_2)\boldsymbol{e}_2 = \cos[\angle(\boldsymbol{e}_2^\phi,\boldsymbol{e}_1)]\boldsymbol{e}_1 + \cos[\angle(\boldsymbol{e}_2^\phi,\boldsymbol{e}_2)]\boldsymbol{e}_2 \end{aligned} \tag{6.13}$$

By looking at **Figure 6.3**, we can easily calculate the angles between the respective pairs of unit vectors:

$$\begin{aligned} \boldsymbol{e}_1^\phi &= \cos(\phi)\boldsymbol{e}_1 + \cos(\pi/2-\phi)\boldsymbol{e}_2 = \cos(\phi)\boldsymbol{e}_1 + \sin(\phi)\boldsymbol{e}_2 \\ \boldsymbol{e}_2^\phi &= \cos(\pi/2+\phi)\boldsymbol{e}_1 + \cos(\phi)\boldsymbol{e}_2 = -\sin(\phi)\boldsymbol{e}_1 + \cos(\phi)\boldsymbol{e}_2 \end{aligned} \tag{6.14}$$

In matrix notation, the unit vectors of the rotated coordinate system can be represented as the product of the two-dimensional *rotation matrix*:

$$\boldsymbol{R} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix}, \tag{6.15}$$

and the original unit vectors:

$$\begin{aligned} \begin{pmatrix} \boldsymbol{e}_1^\phi \\ \boldsymbol{e}_2^\phi \end{pmatrix} &= \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \end{pmatrix} \\ \boldsymbol{E}^\phi &= \boldsymbol{R}\boldsymbol{E}, \end{aligned} \tag{6.16}$$

where E is a column array with unit vectors as its elements.

The coordinates of any vector $\boldsymbol{x}$ are expressed in the rotated coordinate system as:

$$\begin{aligned} \begin{pmatrix} x_1^\phi \\ x_2^\phi \end{pmatrix} &= \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ \boldsymbol{x}^\phi &= \boldsymbol{R}\boldsymbol{x}, \end{aligned} \tag{6.17}$$

The inverse transformation, $\boldsymbol{R}_{\text{inv}}$, is obtained by a back-rotation, $-\phi$:

$$\begin{aligned} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} &= \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x_1^\phi \\ x_2^\phi \end{pmatrix} \\ \boldsymbol{x} &= \boldsymbol{R}_{\text{inv}}\boldsymbol{x}^\phi, \end{aligned} \tag{6.18}$$

A rudimentary MATLAB implementation of a vector drawing program uses rotation and translation is listed in Section 6.10.

6.6 MATLAB Arrays

Remark 10

> The MATrix LABoratory (MATLAB) language works with a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cells, structures, and user-defined objects, are stored as MATLAB arrays.

6.7 Arrays

6.7.1 Some useful array functions

Consider the following array:

$$A = \begin{pmatrix} 6 & 9 & 4 \\ 1 & 5 & 7 \end{pmatrix}$$

```
>> A = [6,9,4;1,5,7]
```

You should be familiar with the array functions:

- size(A) returns a row vector `[m,n]` with the dimensions $m \times n$ of array A

```
>> size(A)
ans =
     2     3
```

- length(A) returns the largest dimension (m or n) of array A

```
>> length(A)
ans =
     3
```

- sum(A) returns a row vector containing the sums over each column (along rows) of array A

```
>> sum(A)
ans =
     7    14    11
>> sum(sum(A))
ans =
    32
>> sum(A’)’
ans =
    19
    13
```

- Other functions work in a similar way (max, min, sort)

```
>> max(A)
ans =
     6     9     7
>> min(A’)’
ans =
     4
     1
>> sort(A)
ans =
     1     5     4
     6     9     7
```

6.7.2 Special matrices

- zeros(m,n) creates a $m \times n$ matrix of zeros
- ones(m,n) creates a $m \times n$ matrix of ones
- eye(n) creates a $n \times n$ identity matrix

```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

6.7.3 Matrix multiplication

This topic is of particular relevance, since it shows up very often in engineering calculations. General matrix multiplication is defined as follows. Let $A = [a_{ik}]$ be a $m \times p$ matrix, $B = [b_{kj}]$ be a $p \times n$ matrix, we define matrix $C = [c_{ij}]$ as the following $m \times n$ matrix:

$$C = A * B \quad \Longleftrightarrow \quad c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

For example, if

$$A = \begin{pmatrix} 6 & -2 \\ 10 & 3 \\ 4 & 7 \end{pmatrix}, \qquad B = \begin{pmatrix} 9 & 8 \\ -5 & 12 \end{pmatrix}$$

then

$$\underbrace{C}_{(3\times 2)} = \underbrace{A}_{(3\times 2)} * \underbrace{B}_{(2\times 2)} = \begin{pmatrix} (6)(9)+(-2)(-5) & (6)(8)+(-2)(12) \\ (10)(9)+(3)(-5) & (10)(8)+(3)(12) \\ (4)(9)+(7)(-5) & (4)(8)+(7)(12) \end{pmatrix} = \begin{pmatrix} 64 & 24 \\ 75 & 116 \\ 1 & 116 \end{pmatrix}$$

```
>> A = [6,-2;10,3;4,7]
A =
     6    -2
    10     3
     4     7
>> B = [9,8;-5,12]
B =
     9     8
    -5    12
>> C = A*B
C =
    64    24
    75   116
     1   116
```

Particular cases are:

- Matrix-vector multiplication

```
>> A = [2,7;6,-5]
```

```
A =
     2     7
     6    -5
>> v = [3;9]
v =
     3
     9
>> A*v
ans =
    69
   -27
```

- Vector-vector multiplication: dot product

```
>> f = [7,-3,5]
f =
     7    -3     5
>> r = [4,2,9]'
r =
     4
     2
     9
>> f*r
ans =
    67
```

6.7.4 Multidimensional arrays

Arrays are not restricted to be vectors or two-dimensional matrices with rows and columns only. You can think of:

3D arrays as 2D matrices organized in a row vector, e.g.,

$$C = \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right)$$

4D arrays as 2D matrices organized in rows and columns, e.g.,

$$D = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \\ & \\ \begin{pmatrix} 9 & 10 \\ 11 & 12 \end{pmatrix} & \begin{pmatrix} 13 & 14 \\ 15 & 16 \end{pmatrix} \end{pmatrix}$$

We can address portions of a multidimensional array as usual:

```
>> A = [1,2;3,4]
A =
     1     2
     3     4
>> C = A
C =
     1     2
     3     4
>> C(:,:,2) = A+4
C(:,:,1) =
     1     2
     3     4
C(:,:,2) =
     5     6
     7     8
>> size(C)
ans =
     2     2     2
>> C(1,1,:)
ans(:,:,1) =
     1
ans(:,:,2) =
     5
```

6.8 Exercises

1. In function vectoralgebra.m change the two unit vectors from row vectors to column vectors. Check what happens. Did you expect it? How would you change the code to use column unit vectors?

2. Derive in detail Eqs. (6.12).

3. Derive Eqs. (6.15) and (6.16).

4. Derive Eq. (6.17).

6.9 MATLAB Code for Vector Algebra

.

```
function vectoralgebra()
%---------------------------------------------------------------
% Rudiments of vector algebra, TWP, 2002-02-02
% Define two 2d vectors, plot them, calculate theirt scalar
% product, lengths, etc.
%---------------------------------------------------------------
close all
clc

% Define Cartesian unit vectors
e_1 = [1,0];
e_2 = [0,1];

% Resolve two arbitrary vectors into the unit vectors
% I chose their coordinates to be (3,4,) and (-1,2)
vectorx =  3*e_1 + 4*e_2;
vectory = -1*e_1 + 2*e_2;


figHndl=figure;

hold on % otherwise only the last plotted object will show up

% Give espect ratio 1:1 to the axes. Otherwise, the vector arrow
% will be deformed. Comment the next line out and rerun
axis equal

% Turn off the axes

axis off

% Use my function myarrow.m to plot the vectors.
% To plot the arrow, this function uses rotation and translation
% of a Cartesian coordinate system in which the vector's line is
% plotted.
myarrow([0,0],[vectorx(1),vectorx(2)],0.25);
myarrow([0,0],[vectory(1),vectory(2)],0.25);


%Scalar product of vectors x and y
sp=vectorx*vectory';
sp1 = dot(vectorx,vectory);

% length of vector x
xlen= sqrt(vectorx*vectorx');
```

```
% length of vector y
ylen = sqrt(vectory*vectory');

% Add coordinate lines
plot([min(vectorx(1),vectory(1)), max(vectorx(1),vectory(1))],...
      [0,0], 'Color', 'k');
plot([0,0],[0, max(vectorx(2),vectory(2))],'Color', 'k');

% Print to string fn the desired eps file name
fn = sprintf('E77N-Figure3.4.eps');
% Print to a color eps Level 2 file for handout
% print( figHndl, '-depsc2', fn );


%print results to the screen
fprintf('Vector x: x(1)=%g, x(2)=%g\n',vectorx(1),vectorx(2))
fprintf('Vector y: x(1)=%g, y(2)=%g\n',vectory(1),vectory(2))
fprintf('Scalar product x*y=%g or %g\n',sp,sp1);
fprintf('Length of vector x=%g\n',xlen);
fprintf('Length of vector y=%g\n',ylen);
```

6.10 MATLAB Code to Draw Vectors

.

```
function [L_h,F_h]=arrowgen(Start,Stop,ALen,Base,SLen,Width,Color,BLineStyle);
%---------------------------------------------------------
% MYARROW        plots a a single vector with an arrow
%                plot an arrow with the following properties:
%
%                A-D = SLen                  ^ A
%                A-C = ALen                 / \
%                A   = Stop point          /   \
%                B   = Start point        /     \
%                E-F = Base              / ..D.. \
%                                      E/..  |C ..\F
%                                            |
%                                            |B
% Input  : - Start = point [X Y]
%          - Stop  = point [X Y]
%          - Alen  = arrow length (default is 0.05 of total length).
%          - Base  = length  (default is 1/2 of Alen).
%          - Slen  = short length (default is Alen).
%          - Width = line width (default is 1.5).
%          - Color = plot color(default is 'k').
%          - BLineStyle = boundary line style (default is '-').
% Output : - Line handle.
%          - Arrow (filled) handle.
%
```

```
% By : Eran O. Ofek, October 1999
% Modified by TWP, 05-04-2002
%-------------------------------------------------------

% Set the defaults if the user does not specify them.
% nargin = Number of Arguments In.  if nargin==8, means
% if there are eight input arguments.
%
if (nargin==8),
   % no defaults left to set.
elseif (nargin==7)
   BLineStyle = '-';
elseif (nargin==6)
   Color      = 'k';
   BLineStyle = '-';
elseif (nargin==5)
   Width      = 1.5; % changed, TWP
   Color      = 'k'; % changed, TWP
   BLineStyle = '-'; % changed, TWP
elseif (nargin==4)
   SLen       = ALen;
   Width      = 1.5;
   Color      = 'k';
   BLineStyle = '-';
elseif (nargin==3)
   Base       = 0.5*ALen;
   SLen       = ALen;
   Width      = 1.5;
   Color      = 'k';
   BLineStyle = '-';
elseif (nargin==2)
   Length     = sqrt((Stop(2)-Start(2)).^2+(Stop(1)-Start(1)).^2);
   ALen       = 0.05*Length;
   Base       = 0.5*ALen;
   SLen       = ALen;
   Width      = 1.5;
   Color      = 'k';
   BLineStyle = '-';
else
   error('illegal number of input arguments');
end

% ATAN2 is the four quadrant inverse tangent (arc tangent)
Theta  = atan2(Stop(2)-Start(2),Stop(1)-Start(1));

% Arrow angle (between the vector and arrow side)
Phi  = atan(0.5*Base./ALen);

% Length of arrow side
BC   = sqrt(ALen.^2 + (0.5*Base).^2);
```

```
% Plot the line segment for the vector. Do not plot the
% last 0.15 of the line segment covered by the arrow.
% This avoids the ugly-looking blunt arrow tip
L_h=plot([Start(1),Stop(1)-0.85*ALen*cos(Theta)],...
         [Start(2),Stop(2)-0.85*ALen*sin(Theta)],...
         Color);
set(L_h,'LineWidth',Width);
hold on;

%-----------------------------------------------------------
% Rewrote the rest completely, TWP
%-----------------------------------------------------------
% Coordinates of the upper and lower wing of the arrow in a
% basis whose x-axis is aligned with the vector,
% and whose origin is at the tip of the vector
E = BC*[cos(pi-Phi), sin(pi-Phi)]';
F = BC*[cos(pi+Phi), sin(pi+Phi)]';

% Go back to the original coordinate system, rotate by
% (-Theta), and shift left by adding X.
Rinv = [cos(Theta), -sin(Theta); sin(Theta), cos(Theta)];
X = [Stop(1), Stop(2)]';
E = Rinv*E+X;
F = Rinv*F+X;
% Fill the arrow with Color
F_h=fill([Stop(1);E(1);F(1)],...
         [Stop(2);E(2);F(2)],Color);

% Add line contour to the fill, to hide the flat tip of the
% vector line segment that otherwise shows up
set(F_h, 'LineWidth',[0.5]);
set(F_h,'LineStyle',BLineStyle);
%% end main program-------------------------------------------
```

Chapter 7

How to Teach a Computer: Binary Addition for Idiots

7.1 What Are You Going to Learn?

In this chapter you will learn how to instruct a very dumb computer on how to perform addition. The material here has been strongly influenced by RICHARD FEYNMAN's astounding book [12]. The addition will be carried out bit-by-bit on binary, or base two numbers. A base two number, say, $(110)_2$ (read binary one-one-zero, *not* one hundred ten) is equal to $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2$, or the decimal $(6)_{10}$. In particular, you will learn more about the MATLAB functions, the logical instruction blocks if, elseif, else...end, the logical operators NOT ($\sim$), AND (&) and OR (|), and the for...end loops. As a byproduct, you will also learn more about the formatted output, fprintf.

Chapter 7 folder on fttp:\\petroleum.berkeley.edu has the following MATLAB files:

BinaryAdd.m performs binary addition of two input bits, given an input carry bit from previous addition. The outputs are the new carry bit, if any, and the result bit.

AddBits.m calls `BinaryAdd` with all bit combinations you will encounter in binary addition. This is how you test a new function for implementation errors.

AddNumbers.m is a rudimentary adder of arbitrary binary numbers of *equal* length. If the two numbers have unequal lengths, the shorter one must be padded with *leading zeros.* The binary numbers are input as they appear in writing, i.e., they are *un*-flipped left-right.

binarray2dec.m converts a binary row array with the least significant bit first (i.e., flipped left-right) into a decimal integer.

decnum2bin.m converts a decimal integer to a row binary array with the least significant bit first (i.e., flipped left-right).

7.2 Why Is It Important?

Computers are quite dumb; they will not understand plain English, but only zeros (off) and ones (on). They require most exact instructions on what to do. These instructions must have no ambiguities whatsoever and be restricted to a particular vocabulary translated by appropriate software, here the MATLAB interpreter, into *finite* sequences of zeros and ones. The logical instructions and operators allow computers to perform conditional operations. The do-loops instruct the computer to repeat the same operations a fixed number of times. It is important to learn the most fundamental (key)words and sentences of any computer language.

7.3 Binary addition

Computers work only with binary (base two) numbers that consist of zeros and ones, **Table 7.1** and **7.2**. Let us depict a binary number as a strip of plastic made of little compartments, just like an ice tray. Each compartment corresponds to a digit; when the digit is 0, a compartment is empty, when the digit is 1, we fill the compartment with water or put a pebble in it. Now let us take two such strips and pretend that these are the numbers to be added, or the "summands". Underneath these two strips we put a third one that will hold the answer. The basic rules of binary addition are:

$$\begin{aligned} &0+0=0\\ &0+1=1\\ &1+0=1\\ &1+1=0 \text{ plus a carry bit or one in the next compartment} \end{aligned} \tag{7.1}$$

Now you can imagine giving instructions on how to move the pebbles to someone who is a complete idiot, i.e., a computer: if you have two pebbles in two aligned compartments of the summands, you put no pebble in the answer compartment beneath, but you carry one pebble to the answer compartment to the left, etc. The amazing thing is that with very specific rules the idiot computer can add binary numbers of any size. With a slightly more detailed set of rules our computer can even do multiplication. As we extend the rules the computer can do very complicated calculations indeed. Because the machine is very fast, we tend to forget how incredibly stupid it is. Once we forget, we become the enchanted consumers of the "Information Society," ready to purchase the Brooklyn Bridge from a passer-by or Windows XP® from Microsoft a week after their release.

Table 7.1: An English oral numbering system using base 2 [19].

1	one	10	eightsome twosome
2	twosome	11	eightsome twosome-one
3	twosome-one	12	eightsome foursome
4	foursome	13	eightsome foursome-one
5	foursome-one	14	eightsome foursome-twosome
6	foursome twosome	15	eightsome foursome twosome-one
7	foursome twosome-one	16	sixteensome
8	eightsome	17	sixteensome-one
9	eightsome-one		etc.

Table 7.2: Decimal integers and their binary equivalents

Decimal	Binary	Binary Expansion
1	1	
2	10	
3	11	$1 \times 2^1 + 1 \times 2^0$
4	100	$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
5	101	$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
6	110	$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
7	111	$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
8	1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
9	1001	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
10	1010	$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
...	...	...
16	10000	$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

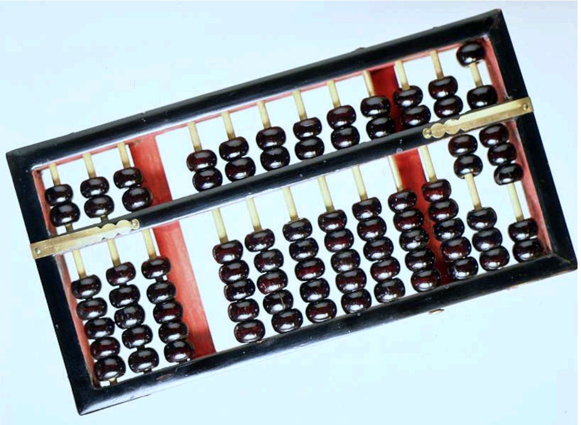


Figure 7.1: Abacus is an ancient tool used for counting; it consists of a frame with rods on which beads or balls get moved. The abacus has been in existence in China since the second century BCE. The oldest abaci come from the fourth century BCE and were found in Rome.

Of course, computers today[1] do not move pebbles, they use high voltage for 1 and low voltage for 0. Each strip is a register and the number of compartments is the register length in bits. So 64 bits correspond to 64 compartments. Thus far we have been representing integers as binary numbers. Later on we will learn how to *approximate* the *floating-point* decimals as 64 pebbles.

Our task is to design a binary adder in MATLAB using the rules in Eq. 7.1. The two strips along an ice tray will be one dimensional arrays. Each compartment will contain a digit. So to represent 22 as a binary number, we

[1] The computers of yesterday, named abaci, actually used beads that slid along the rods, see Figure 7.1.

Figure 7.2: Binary addition of pebbles in compartments of an ice tray. The first two strips are summands, and the third strip is an accumulator or sum.
We are adding the binary equivalents of $(13)_{10}$ and $(22)_{10}$.

		●	●		●

$+$

	●		●	●	

$=$

●				●	●

will say s1=[1 0 1 1 0] or s1=[1, 0, 1, 1, 0].

The MATLAB response will be

```
>> s1=[1 0 1 1 0]

s1 =
     1     0     1     1     0

>> s1(1)

ans =
     1

>> s1(2)

ans =
     0

>> s1(3)

ans =
     1

>> s1(4)

ans =
     1

>> s1(5)
ans =
     0
```

Now, as you see the elements of an array are read from left to right, whereas the actual numbers are the increasing powers of the base (2^0, 2^1, 2^2,...) from right to left, so we flip the array left to right:

```
>> s1=fliplr(s1)
s1 =
     0     1     1     0     1
```

Note that fliplr(Some array) is a built-in function of MATLAB. Now the first element of the array is the lowest bit of the binary number, and the last element is the highest bit.

7.3.1 Logical Instructions

Before implementing the adder, we need to remind you about the logical operators AND, OR , XOR and NOT, see **Table 7.3**:

Table 7.3: The basic logical operators

Name	Logic	MATLAB
AND	$\wedge$	&
OR	$\vee$	|
NOT	$\sim$	$\sim$
eXclusive OR (XOR)	$\bigoplus$	xor(a,b)

$$\begin{aligned} \text{Logical AND operator :} \quad & X = A \wedge B \\ \text{MATLAB AND operator :} \quad & x = a \,\&\, b \end{aligned} \tag{7.2}$$

$$\begin{aligned} \text{Logical OR operator :} \quad & X = A \vee B \\ \text{MATLAB OR operator :} \quad & x = a \mid b \end{aligned} \tag{7.3}$$

$$\begin{aligned} \text{Logical eXclusive OR (XOR) operator :} \quad & X = A \oplus B \\ \text{MATLAB XOR operator :} \quad & x = \text{xor}(a, b) \end{aligned} \tag{7.4}$$

The logical Truth Tables for these three operators are listed in **Tables 7.4-7.6**. For example, the first row of Table 7.4 reads as follows: if A is false and B is false, then $A \wedge B$ is false. The third row of Table 7.5 reads as follows: if A is true and B is false, then $A \vee B$ is true.

Remark 11 Note that the *logical assignment* is expressed in MATLAB with a "==" operator, as opposed to a *value assignment* operator "=" discussed in Chapter 5. So if x==0 is read if x is false. □

Remark 12 The logical value of any positive number is true, while that of any non-positive number is false. □

Now let us implement a full binary adder. We start from planning the number and types of inputs and outputs. We conclude that our adder should have three inputs: the first bit to be added, the second bit to be added, and the carry bit from the previous addition **Figure 7.3**. It should also have two outputs: the sum and the carry bit if any. Here is how a binary adder might look like:

Table 7.4: The Truth Table for the logical AND operator

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

$$X = A \wedge B$$

Table 7.5: The Truth Table for the logical OR operator

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

$$X = A \vee B$$

Table 7.6: The Truth Table for the logical XOR operator

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

$$X = A \bigoplus B$$

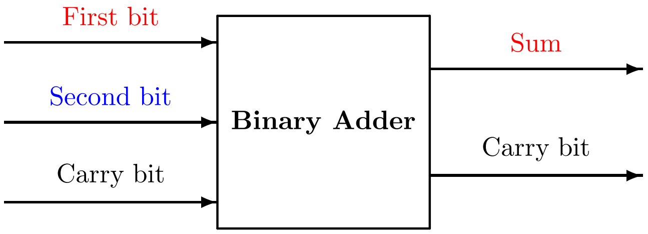


Figure 7.3: Block diagram of the binary addition function. The innards of this function are hidden from the users. For the casual user, this function becomes a black box.

```
function [cout,sum]=BinaryAdd(b1,b2,cin)
%-----------------------------------------------------------------------
% Function BinaryAdd performs binary addition of input bits,
%     b1 and b2,
% given the input carry bit from previous addition,
%     cin.
% The outputs are the new carry bit, cout, if any, and the binary
% result, sum.
%
% Written by T.W. Patzek, 12-30-2001
% U.C. Berkeley, patzek@patzek.berkeley.edu
% E77N Class Material, University of California, Berkeley
%
% Last revised, TWP, 01-01-2002
%-----------------------------------------------------------------------

% Assume that the previous carry bit is zero.
% The symbol ~ is read "not," e.g., if b1=0, ~b1=1.

if ~b1 & ~b2        % b1 = 0 and b2 = 0, i.e., both are false
   sum  = 0;
   cout = 0;
elseif (~b1 & b2) | (b1 & ~b2)  % b1 or b2 = 1, the other = 0
   sum  = 1;
   cout = 0;
else                            % b1 and b2 = 1
   sum  = 0;
   cout = 1;
end

% If the input carry bit is 1, flip the results, but only if
% b1 and b2 are NOT both ones.  In the last case, we need to
% set both the sum and cout to one.
```

```
if cin
   if sum           % sum = 1
      sum  = 0;
      cout = 1;
   else             % sum = 0
      if ~b1 & ~b2  % b1 and b2 = 0
         sum  = 1;
         cout = 0;
      else          % b1 = 1 and b2 = 1
         sum  = 1;
         cout = 1;
      end
   end
end

% A simpler way of writing the last logical instructions block is:
%if cin
%   sum  = 1;
%   cout = 1;
%   if sum           % sum = 1
%      sum  = 0;
%   else             % sum = 0
%      if ~b1 & ~b2  % b1 and b2 = 0
%         cout = 0;
%      end
%   end
%end
```

The syntax of the **if** . . . **elseif** . . . **else** . . . **end** programming structures have been described in Chapter 4.

We can test out new binary adder by writing the following function:

```
function AddBits()
%------------------------------------------------------------------
% AddBits calls BinaryAdd with all bit combinations
%         in binary addition.  This is how you test a new function
% Written by T.W. Patzek, January 2, 2002
% E77N Class Material, University of California, Berkeley
%------------------------------------------------------------------

clc; % Clear MATLAB window screen

s1  = 0;
s2  = 0;
cin = 0;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);
```

```
s1  = 1;
s2  = 0;
cin = 0;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);


s1  = 0;
s2  = 1;
cin = 0;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);

s1  = 1;
s2  = 1;
cin = 0;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);


fprintf('\n\n')

s1  = 0;
s2  = 0;
cin = 1;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);


s1  = 1;
s2  = 0;
cin = 1;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);


s1  = 0;
s2  = 1;
cin = 1;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);

s1  = 1;
s2  = 1;
```

```
cin = 1;
[cout, s] = BinaryAdd(s1,s2,cin);
fprintf('s1 =%d, s2  =%d, cin=%d\n',s1,s2,cin);
fprintf('sum=%d, cout=%d\n\n', s,cout);
```

The features of the `fprintf` function have been described in Chapter 4.

7.3.2 For-Loops

In order to add two binary numbers of equal, but arbitrary length, we need to call our single-bit adder repeatedly. In MATLAB this is done in a **for** loop, see Chapter 4:

```
function AddNumbers()
%---------------------------------------------------------------------
% AddNumbers is the first adder of arbitrary binary numbers of EQUAL
% length.  If the two numbers have unequal lengths, the shorter one
% must be padded with LEADING zeros
%
% Written by T.W. Patzek, 12-30-2001
% U.C. Berkeley, patzek@patzek.berkeley.edu
% E77N Class Material, University of California, Berkeley
%
% Last revised, TWP, 01-01-2002
%---------------------------------------------------------------------

clc

% Add the two rows of ice tray in Figure 3.1
% NOTE that each number is PREPADDED with a leading zero
% to make allowance for the carry bit in the sum<<<<<<<<<<<<<<<<
%     |
%     V
n1 = [0 0 1 1 0 1];
n1 = fliplr(n1); % flip the number lef-right
n2 = [0 1 0 1 1 0];
n2 = fliplr(n2);

len = length(n1); % find the commnon length of BOTH numbers
% This is how you declare an array of zeros with one row
% and len columns
sum = zeros(1,len) % prepare and empty tray for the results

c = 0; % set the first input carry bit to zero

% Repeat the same operation several times in a do-loop
% i is the loop running index, 1 is the starting value of i
% and len is the final value of i.
for i = 1:len
   fprintf('n1(%d)=%d, n2(%d)=%d, cin=%d\n',i,n1(i), i,n2(i), c);
   % Reuse our old-and-tried binary adder of single bits:
   [c,sum(i)] = BinaryAdd(n1(i),n2(i),c);
   fprintf('sum(%d)=%d, cout=%d\n',i,sum(i), c);
end

sum = fliplr(sum)
```

7.4 Computer Representation of Numbers

We represent numbers using the *decimal* system, i.e., powers of 10. For example:

$$N = \mathbf{2} \times 10^2 + \mathbf{5} \times 10^1 + \mathbf{3} \times 10^0 = (253)_{10}.$$

In a computer that uses the *binary* system, i.e., powers of 2, the same number 253 is represented as:

$$N = \mathbf{1}\times 2^7 + \mathbf{1}\times 2^6 + \mathbf{1}\times 2^5 + \mathbf{1}\times 2^4 + \mathbf{1}\times 2^3 + \mathbf{1}\times 2^2 + \mathbf{0}\times 2^1 + \mathbf{1}\times 2^0 + = (11111101)_2.$$

We can also think of *real* numbers in both systems. For example:

$$\begin{aligned}(14.75)_{10} &= 1 \times 10^1 + 4 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2} \\ &= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= (1110.11)_2\end{aligned}$$

MATLAB follows the IEEE Standard for floating-point arithmetic. The IEEE floating-point format includes three components: the sign bit, the exponent field, and the significand (mantissa) field. Single-precision floating-point values consist of 32 bits, and the value is given by

$$\text{value} = (-1)^{\text{sign}} \left(2^{\text{exp}-127}\right) (1.\text{significand})$$

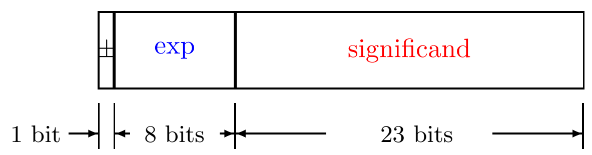


Figure 7.4: Binary representation of a number with single precision (32 bits)

Double-precision floating-point values consist of 64 bits, and the value is given by

$$\text{value} = (-1)^{\text{sign}} \left(2^{\text{exp}-1023}\right) (1.\text{significand})$$

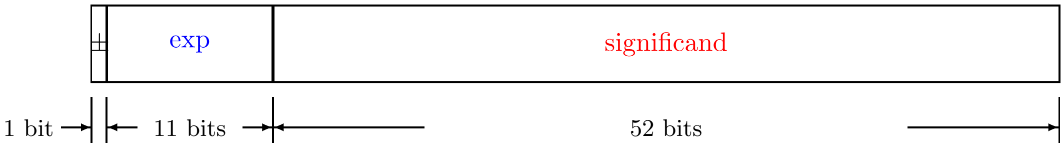


Figure 7.5: Binary representation of a number with double precision (64 bits)

The significand is *normalized* if it has the form $1.f$, where 1 is always the leading bit (known and not stored), and f is the stored fraction with arbitrary bits (0's or 1's in any combination.) A *denormalized* significand has 0 as the leading bit.

The fixed number subtracted to the exponent allows an accurate representation of both small and big numbers. For the double-precision case, the minimum value of the exponent is

$$(00000000000)_2 = 0,$$

and the maximum value

$$(11111111111)_2 = 2^{11} - 1 = 2047.$$

Thus, $(\exp - 1023)$ ranges from -1023 to 1024.

For the largest IEEE normalized numbers, the least-significant bit of the exponent is set to 0, and the largest value of the exponent is 1023. Conversely, for the smallest normalized numbers the first bit is set to 1, and the smallest value of the exponent is -1022:

$$\begin{aligned} &(11111111110)_2 - 1023 = 2^{11} - 2 - 1023 = 1023 && \text{Largest exponent of normalized number} \\ &(00000000001)_2 - 1023 = 2^0 - 1023 = -1022 && \text{Smallest exponent of normalized number} \end{aligned}$$

The biggest normalized number (in absolute value) which can be represented in MATLAB is

$$R_{\max} = (2 - 2^{-52}) \times 2^{1023} = 1.797693134862316 \times 10^{308},$$

i.e., all 52 bits of this number's significand are set to 1.

The minimum denormalized number is degraded to the last significant digit in the fraction (only the 52$^{\text{nd}}$ bit is set to 1):

$$R_{\min} = 2^{-52} \times 2^{-1022} = 2^{-1074} = 4.940656458412465 \times 10^{-324}$$

The floating point relative accuracy, i.e., the distance from 1.0 to the next larger floating point number is

$$\epsilon = 2^{-52} = 2.220446049250313 \times 10^{-016}.$$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double-precision), and the significand is filled with 1s (including the normalizing 1 bit).

7.4.1 Putting it All Together

In summary:

1. The sign bit is 0 for positive, 1 for negative.

2. The exponent's base is two. The exponent field contains -127 plus the true exponent for single-precision numbers, or -1023 plus the true exponent for double-precision numbers.

3. The significand (mantissa) is typically assumed to be *normalized*, $1.f$, i.e., the first bit is always 1, and f is the field of fraction bits (23 in single precision and 52 in double precision).

4. For the largest normalized numbers, the exponent is restricted to 2046-1023, by setting its first bit to 0.

5. For the smallest normalized numbers the exponent is restricted to $1 - 1023 = -1022$, by setting its first bit to 1.

Precision	Denormalized	Normalized	Approximate Decimal
Single	$\pm 2^{-149}$ to $\pm(1 - 2^{-23}) \times 2^{-126}$	$\pm 2^{-126}$ to $\pm(2 - 2^{-23}) \times 2^{127}$	$\sim \pm 10^{-44.85}$ to $\sim \pm 10^{38.53}$
Double	$\pm 2^{-1074}$ to $\pm(1 - 2^{-52}) \times 2^{-1022}$	$\pm 2^{-1022}$ to $\pm(2 - 2^{-52}) \times 2^{1023}$	$\sim \pm 10^{-323.3}$ to $\sim \pm 10^{308.25}$

The Chapter 7 function `decnum2bin` converts a decimal *integer* to its binary representation with the least significant bit as the first element. To make the binary number appear as written by hand do the following:

```
>> fliplr(decnum2bin(1000))
ans = 1111101000
```

7.4.2 Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized Number

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

7.5 Exercises

1. In MATLAB command window, go to Help → Matlab Help. Click on the Index tab, and by typing "operator precedence," find the precedence rules for MATLAB operators. These are shown as a list, ordered from highest precedence level to lowest precedence level. You need to memorize this list.

2. Now type fprintf into the Index tab, and familiarize yourself with the other typical formatting strings, such as '%e', '%12.4e', '%f', '%10.4f', etc. Try to use fprintf in your own code as soon as possible. It pays to communicate your output clearly.

3. Simplify the BinaryAdd.m function using the eXclusive OR (XOR) operator, see **Table 7.6**, which in MATLAB is implemented as s=xor(b1,b2).

4. Write a function [d]=bin2dec(b) that converts an array representation of a binary number to the corresponding decimal number. Use the fact that if the left-right flipped binary form is $b = [b_1, b_2, b_3, \ldots, b_N]$, the corresponding decimal number, d, is

   $$d = \sum_{i=1}^{N} b(i) 2^{i-1} \tag{7.5}$$

   Use the for...end construct to perform the conversion and return the result from the function.

   Bonus: By using MATLAB help find out how MATLAB can do the summation more efficiently. It can, but you must first convert the powers of 2 in the sum to an array of length equal to b.

5. **Bonus**: Write a more sophisticated black-box function [sum] = AddNumbers2(n1,n2,flag), where n1 and n2 are the input bit arrays of arbitrary lengths, and sum is the binary result. The flag tells the computer if the inputs should be flipped.

   Hint: Work with the flipped forms of the arrays. Suppose that n1 is longer. Pre-pad n2: n2 = [zeros(1,length(n1)-length(n2),n2)]. Also allow for n2 to be longer. Make sure that both n1 and n2 are row arrays. To achieve this, put n1=n1(:)', and the same for n2. Explain in words what are the effects of this operation.

6. **Bonus**: Write a decimal-to-binary integer converter. The input is the positive decimal integer (e.g., 813), and the output is a row array of (flipped) binary digits (e.g, [1 0 1 1 0 1 0 0 1 1]), see **Table 7.7**.

   Hint: Use a while(number>0) ... end loop to divide the input number by 2 as many times as necessary. Each binary digit will be the remainder of division by two (the MATLAB function rem: b(j)=rem(number,2)), and the new input will be the smallest integer after the division (the MATLAB function floor: number=floor(number,2)). The condition to stop the while loop will be the next number equal to zero, see **Table 7.7**.

7. As in Problem 5, but add a second output argument that is the binary sum array converted to a decimal number.

8. **Double Bonus**: Write a function that takes two decimal integers as inputs, performs the binary addition and outputs the result in both binary and decimal form.

Table 7.7: Conversion of decimal 813 to a binary array through the iterated division by 2. The binary array is already flipped left-right, with the *least* significant bit as the first element.

Iteration	Decimal (floor)	Bit (rem)
1	813(start)	1
2	406	0
3	203	1
4	101	1
5	50	0
6	25	1
7	12	0
8	6	0
9	3	1
10	1(stop)	1

$$813 \doteq [1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1]$$

Table 7.8: Conversion of decimal 0.1 to a binary array through the iterated multiplication by 2. The binary array starts with the *most* significant bit as the first element.

Iteration	Decimal multiplication	Bit from integer part
0	0.1	0.(start)
1	0.2	0
2	0.4	0
3	0.8	0
4	1.6 → 0.6	1
5	1.2 → 0.2	1
6	0.4	0
7	0.8	0
8	1.6 → 0.6	1
9	1.2 → 0.2	1
10	etc.	etc.

$$0.1 \doteq 0.[0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \ldots]$$

9. **Double Bonus**: Conversion of decimal fractions to binary fractions. Conversion of decimal 0.1 to an infinite, periodic binary fraction is shown in **Table 7.8**. Write a MATLAB function to perform the conversion from an arbitrary decimal fraction as input to a binary fraction with a specified number of digits.

10. Show that with all bits of the double-precision significand f set to 1, $1.f = 2 - 2^{-52}$.

Chapter 8

Data Structures

8.1 What Are You Going To Learn?

In this chapter you will learn about more complicated data types in MATLAB, *structures* and *cell arrays*. We shall proceed as follows:

- Review of data types in MATLAB
- Building structure arrays
- Accessing data in structure arrays
- Finding the size of structure arrays
- Adding fields to structures
- Deleting fields from structures
- Applying functions and operators
- Writing functions to operate on structures
- Organizing data in structure arrays
- Nesting structures

The Chapter 8 folder on fttp:\\petroleum.berkeley.edu has the following files:

cellarray.m creates cell arrays.

doubledatatype.m creates an array of complex numbers.

readdoubledata.m reads a complex array from a text file.

input.dat is a text file with the complex array elements.

sparsearray.m creates a sparse array.

stringarray.m creates a string array.

structures.m creates structures and structure arrays.

In order to fully understand the material this chapter, please download the code fragments above, add your lines and answer the questions.

8.2 Why Is It Important?

MATLAB now provides you with ways of organizing data that rival capabilities of Java and C++. If you want to use the full power of MATLAB, you must be fluent in *data types* used by it.

8.3 Useful Definitions

Definition 4 A *class* is defined as a generalization of *objects* [13]: "a group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality, degree of competence, or condition". In the context of *object-oriented-programming* (OOP) a class is a set of objects that share a common structure and a common behavior. A single object is simply an instance of a class. □

Definition 5 There are many definitions of an object, e.g., [13]: "An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable". □

8.4 Data Types in MATLAB

The MATLAB language works with only one *object* type: the *MATLAB array.* All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects are stored as MATLAB arrays.

There are five fundamental *data types* or *classes* in MATLAB. Each of these data types resides of course in an array. An array can grow from 0-by-0 in size and to an n-dimensional array of any size computer memory will permit. One-dimensional arrays are loosely called vectors, and two-dimensional arrays are loosely called *matrices* [1]. The fundamental data types, or classes, are:

- Character (char): 'c', 'e77n'
- Numeric: double and sparse
- Cell: N{1,1}=[1,2,3]

[1] For more precise definitions, see Chapter 6

- Structure: struct phonebook ('number',1234)
- Function handle: function_handle = @myfunction;

MATLAB help will inform you that there are other classes, such as int8 (8-bit signed integer array), int16 (16-bit signed integer array), int32 (32-bit signed integer array), single (single-precision floating-point array), uint8 (8-bit unsigned integer array), uint16 (16-bit unsigned integer array), and uint32 (32-bit unsigned integer array). You can create objects that belong to each of these classes, but you cannot perform any operations on them without conversion to double precision. Thus, for the purpose of this course, we shall ignore all these additional classes.

When we define a particular array of one of these types, we create an *object* that belongs to a *class* from which it was derived. For example, array A = ['Joe', 'Doe', 'Mow'] is an object of class char:

$$\begin{aligned} K &= \text{isa(obj, 'class\_name')} \leftarrow \text{Syntax} \\ K &= \text{isa(A, 'char')} \leftarrow \text{Use} \\ >> &\quad 1 \end{aligned}$$

The char data type holds characters in Unicode[2] representation. A character string of length n is merely a 1-by-n array of characters. You can use char to hold an array of strings as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To hold an array of strings of unequal length, use a cell array.

Numeric data types include double precision arrays that are full, or sparse arrays that contain only a few nonzero elements. All MATLAB computations are done in *double-precision.*

A cell array provides a storage mechanism for dissimilar kinds of data. You can store arrays of different types and/or sizes within the cells of a cell array. For example, you can store a 1-by-50 char array, a 7-by-13 double array, and a 1-by-1 uint32 (unsigned 32-bit integer) in cells of the same cell array. You access data in a cell array using the same matrix indexing used on other MATLAB matrices and arrays, but you cannot perform any operations on them.

The MATLAB structure data type is similar to the cell array in that it also stores dissimilar kinds of data. But, in this case, it stores the data in named

[2] International character-encoding system designed to support the electronic interchange, processing, and display of the written texts of the diverse languages of the modern and classical world. The Unicode Worldwide Character Standard includes letters, digits, diacritics (marks such as the ´ in ó), punctuation marks, and technical symbols for all the world's principal written languages, using a uniform encoding scheme. Originally introduced in 1991, the most recent version contains almost 50,000 characters. Numerous encoding systems (incl. ASCII and EBCDIC) predate Unicode. With Unicode (unlike earlier systems), the unique number provided for each character remains the same on any system that supports Unicode.

fields rather than in cells. This enables you to attach a name to the groups of data stored within the structure. You access data in a structure using their field names. Most importantly, on each field, you can perform all operations that are legal for it. (For example, you can add, subtract, etc., all arithmetic fields.)

MATLAB data types are implemented as classes. You can also create MATLAB classes of your own. These user-defined classes inherit from the MATLAB structure class.

8.4.1 Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These matrices are of type double and have dimensions m-by-n, where m is the number of rows and n is the number of columns, e.g.,

$$D(i,j),\ i=1,\dots,m,\ j=1,\dots,n.$$

The matrix elements are stored as two vectors of double-precision numbers - one contains the real data and one contains the imaginary data. Now you should try the first exercise, doubledatatype.m to understand how MATLAB stores the double precision, real and complex numbers.

```
function doubledatatype()
%
% How to create an array of complex numbers
% T.W. Patzek, E77N, April 5, 2001
% Last modified TWP, 2002-02-06
%
clear all; clc
% Double array of complex numbers D:
ReD = [2,4,6;8,10,12;14,16,18];
ImD = [1,3,5;7, 9,11;13,15,17];
D=ReD+i*ImD;
D
fprintf(’%2g %2g %2g\n’,real(D)’);
fprintf(’\n’);
fprintf(’%2g %2g %2g\n’,imag(D)’);
fprintf(’\n’);
whos
% Questions for the students:
% 1. Input D as a single array of complex numbers.
% 2. How exactly are data stored in this array?
% 2a. Why did I put D’ in the fprintf statement?
% 3. By looking at output of whos, tell what is the
%    number of bits (not bytes) per element of D?
% 3a.  What about ReD and ImD?
% Input your code here, output the result:
% Put BRIEF comments below your code
```

```
%------------------------------------------
% Student code starts

% Student code ends
%------------------------------------------
```

Using a text input data file input.dat

```
 2  4  6
 8 10 12
14 16 18
 1  3  5
 7  9 11
13 15 17
```

read in the real and imaginary part of D in the previous example.

```
function readdubledata()
%
% How to read a complex array from a text file
% T.W. Patzek, E77N, April 5, 2001
% Last modified TWP, 2002-02-06
%
clear all; clc
%
% Read real and imaginary parts of D:
fp = fopen('input.dat', 'r');
[ReD]=fscanf(fp,'%f',[3,3]);
[ImD]=fscanf(fp,'%f',[3,3]);
fclose(fp);

D=ReD+i*ImD;
D
fprintf('\nReal part of D\n');
fprintf('%2g %2g %2g\n',real(D)');
fprintf('\nImaginary part of D\n');
fprintf('%2g %2g %2g\n',imag(D)');
fprintf('\n');
whos
% Questions for the students:
% 1. How are data read from input.dat into ReD and ImD?
% 2. Make D in this example look exactly as in Example 2
% Input your code here, output the result:
% Put BRIEF comments below your code
%------------------------------------------
% Student code starts

% Student code ends
%------------------------------------------
```

8.4.2 Sparse Matrices

Sparse matrices have a different storage convention than full matrices in MATLAB. Here is how a sparse matrix can be created from a full matrix.

With big full matrices, we do not have such a luxury and their sparse counterparts must be created directly.

8.4.3 MATLAB Strings

MATLAB strings are of type char and are stored the same way as unsigned 16-bit integers. Each character in the string is stored as 16-bit ASCII Unicode. Unlike C, MATLAB strings are *not null terminated*. Here is an example of inputting string arrays:

```
function stringarray()
% How to create a string array
% T.W. Patzek, E77N, April 5, 2001
% Last modified TWP, 2006-01-22
%
clear all; clc
%
s=[’I    ’;
   ’want ’;
   ’to   ’;
   ’learn’;
   ’all  ’;
   ’I    ’;
   ’can  ’;
   ’in   ’;
   ’E77N ’];
s
whos
% 1. How many bits per each character in s?
% 2. How to output just ’lear’ from s?
% 3. Can rows in s have different lengths?
s1=[];
s1 = strvcat(s1,’I’);
s1 = strvcat(s1,’want’);
s1 = strvcat(s1,’to’);
s1 = strvcat(s1,’learn’);
s1 = strvcat(s1,’even’);
s1 = strvcat(s1,’more’);
s1
whos
```

8.4.4 Cell Arrays

Cell arrays are a collection of MATLAB arrays where each array is referred to as a cell. This allows MATLAB arrays of different types to be stored together.

Each cell can be of any supported data type, even another cell array.

You can create cell arrays by:

- Using assignment statements
- Preallocating the array using the cell function, then assigning data to cells

Using Assignment Statements

You can build a cell array by assigning data to individual cells, one cell at a time. MATLAB automatically builds the array as you go along. There are two ways to assign data to cells:

- Cell indexing

  Enclose the cell subscripts in parentheses using standard array notation. Enclose the cell contents on the right side of the assignment statement in curly braces,“{}”. For example, create a 2-by-2 cell array A.

- Content indexing

  Enclose the cell subscripts in curly braces using standard array notation. Specify the cell contents on the right side of the assignment statement:

```
function cellarray()
% How to create cell arrays
% T.W. Patzek, E77N, April 5, 2001
% Last modified TWP, 2006-01-22
%
clear all; close all; clc
%
%-------------------------------------------------------------
% Cell indexing (each element of an array is a cell):
%-------------------------------------------------------------
A(1,1) = {[1 2 3; 4 5 6; 7 8 9]};
A(1,2) = {’Tad Patzek’};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/10:pi};
A
A{1,1}

% See if we can print out a cell array element:
try
    disp(’>>fprintf(’’%s\n’’,A(1,2));’)
    fprintf(’%s\n’,A(1,2));
catch
    % We could not, so we must clean up our mess:
    err  = lasterror;
    mess = err.message;
    fprintf(’%s\n’,mess);
    disp(’>>fprintf(’’%s\n’’,char(A(1,2)));’)
    fprintf(’%s\n’,char(A(1,2)));
end
%
%-----------------------------------------------------------------
% Content indexing (specify arbitrary elements of a cell array)
%-----------------------------------------------------------------
B{1,1} = [1 2 3; 4 5 6; 7 8 9];
B{1,2} = ’Tad Patzek’;
B{2,1} = 3+7i;
B{2,2} = -pi:pi/10:pi;
B
B(2,2)
celldisp(B)
cellplot(B)
%set(gcf,’PaperPosition’,[1 1 4 3])
%print -depsc cellplot.eps
whos
% Problems:
% Show the difference between B(2,2) and B{2,2}
% Declare an empty cell array
```

Remark 13 The notation“{}” denotes the empty cell array, just as “[]” denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments. Use square brackets to concatenate cell arrays, just as you do for numeric arrays. □

Pre-allocating Cell Arrays with the Cell Function

The cell function allows you to pre-allocate empty cell arrays of the specified size. For example, this statement creates an empty 2-by-3 cell array.

$$B = \text{cell}(2,3);$$

Use assignment statements to fill the cells of B.

$$B(1,3) = \{1:3\};$$

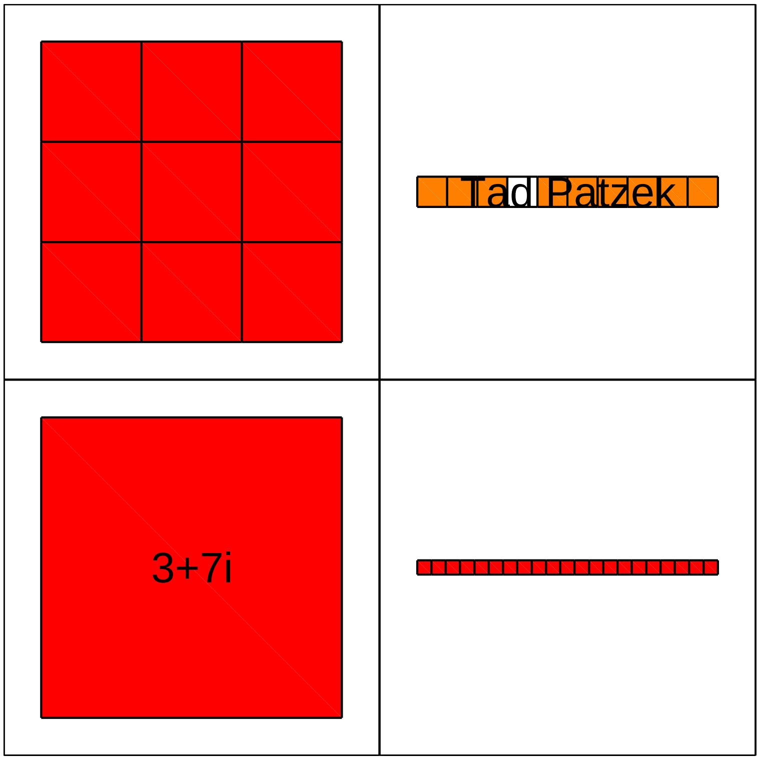


Figure 8.1: Result of cellplot(B).

8.4.5 Structures

A 1-by-1 structure is stored in the same manner as a 1-by-n cell array where n is the number of fields in the structure. Members of the data vector are called fields.

```
function structures()
% How to create structures and structure arrays
% T.W. Patzek, E77N, April 5, 2001
% Last modified TWP, 2002-02-06
%
clear all; clc
% Direct definition of all structure fields
mydata = struct(...
   'name', 'Tadeusz W. Patzek',...
   'office','425 Davis Hall',...
   'favoritenumbers',[0,1,1,3,5,8,13],...
   'phoneprefix',3,...
   'phone',5834);
```

```
% Add a field I forgot to add to the structure definition
mydata.height = 182;

fprintf('mydata.name=%s\n',mydata.name)
ll=length(mydata.favoritenumbers);
fprintf(...
    '\nNumber of elements in favoritenumbers field=%d\n',...
      ll);
fprintf('\nmydata.favoritenumbers:\n');
for(i=1:ll) fprintf('%d ',mydata.favoritenumbers(i));end
fprintf('\n\n');
% You can perform arithmetic operations on numeric fields:
mydata.favoritenumbers(1)+mydata.favoritenumbers(5)

%Structure array defined one field at a time:
friends = {};
friends(1).name = 'Allen Tom';
friends(1).phone = 1212;
friends(2).name = 'Berdichevsky Maria';
friends(2).phone = 3434;
friends(3).name = 'Rahmani Mustapha';
friends(3).phone = 5656;
friends(4).name = 'Carapia Jorge';
friends(4).phone = 7878;

fprintf('friends(%d).name=%s\n',3,friends(3).name);
ll=length(friends);
fprintf('\nfriends(:).phone:\n');
for(i=1:ll) fprintf('%d ',friends(i).phone);end
fprintf('\n');

% Get field information
f = getfield(friends,{3},'phone');
fprintf('\nf=%d\n\n',f);
% You can perform arithmetic operations on numeric fields:
fprintf('\nAdd phone numbers in elements 3 and 4=%d\n\n',...
friends(3).phone+friends(4).phone);
% whos creates a structure array with cell arrays as fields:
s=whos;
s(2).name
s(2).size
s(2).bytes
```

8.4.6 Objects

Objects are stored and accessed the same way as structures. In MATLAB, objects are named structures with registered methods.

8.4.7 Multidimensional Arrays

MATLAB arrays of any type can be multidimensional. A vector of integers is stored where each element is the size of the corresponding dimension. The storage of the data is the same as matrices.

8.4.8 Empty Arrays

MATLAB arrays of any type can be empty.

8.4.9 Data Storage

All MATLAB data is stored column-wise, which is how FORTRAN stores matrices. MATLAB uses this convention because it was originally written in Fortran. For example, given the matrix

$$\begin{aligned} a &= [\text{'house'; 'floor'; 'porch'}] \\ a &= \\ &\ \text{house} \\ &\ \text{floor} \\ &\ \text{porch} \end{aligned}$$

its dimensions are

$$\text{size(a)}$$
$$>> \text{ans} = 3\ 5$$

and its data are stored as (write `a` as a 3 row array and proceed by column)

h	f	p	o	l	o	u	o	r	s	o	c	e	r	h

8.5 Exercises

1. Go through the sample codes and answer the questions included in them.

Chapter 9

Fundamentals of Plotting

9.1 What Are You Going To Learn?

In this chhapter you will learn how to make nice plots in MATLAB. In particular, I will cover the following subjects:

- What is a MATLAB figure?
- How do you change figure properties?
- How do you label figures?
- How do you put legends into figures?
- How do you put text into figures?
- How do you include multiple plots in one figure?
- How do you plot surfaces?
- What is a parametric plot?

Your best source of information on the properties of figures and plotting tricks is MATLAB's online Help.

The Chapter 9 folder on fttp:\\petroleum.berkeley.edu has the following files:

CreateGreenYellowPlot.m demonstrates some of the enormous graphical capabilities of MATLAB. It creates a figure object (window) that covers most of computer screen, changes the figure background to green, turns off the figure number, gives the figure your own title, plots a blue line on a yellow background, changes the line thickness, plots legends, changes the legend font, etc.

CreateGreenYellowPlotLecture.m is the non-demo version of the function above.

fitplot.m is a script to perform several polynomial fits (of increasing order) to a data set. The data points and fitting models are then plotted using subplot. One figure with four sub-figures is created.

legends.m is a script to plot multiple legends.

ParametricPlot.m generates a three-dimensional parametric plot of a trajectory $[x(t), y(t), z(t)]$.

plotpwfun.m is a script file that plots the 2D surface pwfun.m using a naïve **if...end** construct.

pwfun.m computes the values of a 2D surface defined by patches, using the **if** statements (naïve implementation).

plotpwfun2.m plots the 2D function pwfun2.m using the vectorized **find**.

pwfun2.m computes the values of a 2D surface defined by patches, using the **find** function (sophisticated implementation).

PlotZCubed.m plots the real and imaginary parts of a complex function of the complex variable $w(z) = z^3 - 1$ as two 3D color-coded surfaces; one of the surfaces is semi-transparent. It also plots the roots of w as three vertical lines intersecting the surfaces.

9.2 Why Is It Important?

One of the main reasons for using MATLAB is the ease with which you can present your results graphically. MATLAB gives you an unparalleled flexibility in creating custom graphics that satisfy the most subtle artistic taste.

9.3 What Is MATLAB Figure?

Definition 6 MATLAB command figure creates a figure graphics object with a unique identifier, called handle. Each figure object creates its own window that is a *child* of the *root* window, i.e., the screen in which MATLAB displays graphical output. Each new figure object is created with the default properties *inherited* from the root.

Example 2 When you type:

```
h=figure;
```

MATLAB creates a new figure graphics object and refers to it through a handle h, which happens to be a unique number.

You get the current properties of a figure object identified by handle h by using the function:

```
  get(h)
```

You change the properties of `h` by using the function:

```
  set(h,'PropertyName', NewPropertyValue)
```

where `'PropertyName'` is any of the valid properties of the figure object, and `NewPropertyValue` is the new value to which this property will be set. □

Example 3 To create a figure window that is one quarter the size of your screen and is positioned close to the upper-left corner, use the root object's `ScreenSize` property to determine the size. `ScreenSize` is a four-element vector: [left, bottom, width, height]:

```
close all
% 0 is the handle of the root graphics object, i.e, the screen
scrsz = get(0,'ScreenSize');
%                   [left   bottom      width       height  ]
h=figure('Position',[25,  scrsz(4)/2, scrsz(3)/2, scrsz(4)/2]-20)
% On my screen, I need to shift the figure right by 25 pixels,
% and down by 20 pixels for the tool bar and border to show.
```

Note that the window may have to be moved away from the upper left corner to display correctly. □

You get the handle of the *current graphics figure* (gcf) by typing:

```
  figure;
  h=gcf;
```

To change the background color of your current figure from silver to green type:

```
  set(h, 'Color', [0.5, 1, 0]);
```

To change the figure name type:

```
  set(h, 'Name', 'My Green Figure');
```

To turn off the display of the figure number type:

```
  set(h, 'NumberTitle', 'off');
```

By now you have created a green *canvas* on which you may place whatever you want, e.g., a plot. So suppose that you want to plot $y = x^2$, for $x \in [1 : 20]$:

```
  x=1:1:20;
  y=x.^2;
  hp=plot(x,y)
  ax=gca;
  get(ax)
```

Now My Green Figure has a new child, a graphics object called axes. To get the properties of the *graphics current axes*, (gca), we have typed in the last two lines. Suppose that you want the plot area to have yellow background color. You achieve this by setting:

```
  set(ax, 'Color', [1,1,0]);
```

The axes child of the figure object has its own child called line. The line object was created by the axes created by the call to plot. The new line has a unique handle hp. Now you can change the properties of the line object as much as you please:

```
  set(hp, 'LineWidth', 2);
  set(hp, 'Color', 'r');
```

Here is the entire code, written properly as a function:

```
function CreateGreenYellowPlot(x,y)
% x,y vectors of x,y coordinates of the curve y=x^2 to be plotted
%
% TWP, E77N material, U.C. Berkeley, 2002-02-24
%
%
% 0 is the handle of the root graphics object, i.e, the screen
%           |
%           V
scrsz = get(0,'ScreenSize');

%                   [left   bottom      width       height  ]
h=figure('Position',[25,  scrsz(4)/2, scrsz(3)/2, scrsz(4)/2]-20);
% On my screen, I need to shift the figure right by 25 pixels,
% and down by 20 pixels for the tool bar and border to show.

% Change figure background to green, turn off the figure number
% give the figure your own title
set(h, 'Color', [0.5, 1, 0]);
set(h, 'Name', 'My Green Figure');
set(h, 'NumberTitle', 'off');

% Plot blue line on yellow background
hp = plot(x,y);
ax = gca;
get(ax)
set(ax, 'Color', [1,1,0]);

% Change line width to 2 and color to red
get(hp)
set(hp, 'LineWidth', 2);
set(hp, 'Color', 'r');

%Label axes
xlabel('x');
ylabel('y=x^2');

% Change font properties, replot labels
set(ax,'FontSize',14);
set(ax,'FontWeight','bold');
xlabel('x');
ylabel('y=x^2');
```

Let us summarize what we have learned thus far:

We have created three *graphics objects*. The first one was figure, a child of the root, **Figure 9.1**. The graphics object figure was endowed with its own child, axes. The axes child was created by a call to a high-level function plot, which also created a child, line, of the axes. Each of the graphics objects (figure, axes and line) was identified by a unique handle, and could be accessed through its own function set. The *accessor* functions set change the properties of the respective objects from the default values to what we wanted them to be.

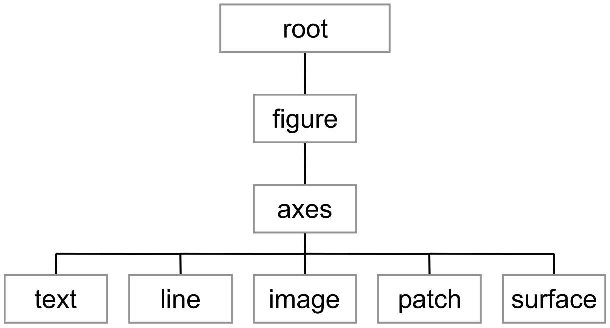


Figure 9.1: Hierarchy of MATLAB graphics objects.

9.4 Figure Labels

Axis labels belong to the axes graphical object. The labels are created by the xlabel and ylabel functions:

```
  xlabel('x');
  ylabel('y=x^2');
```

Properties of all text belonging to the axes object can be changed globally:

```
set(ax,'FontSize',14);
set(ax,'FontWeight','bold');
```

The labels must be replotted with the new text properties, because they inherit these properties from the axes.

9.5 Figure Legends

To plot legends for multiple curves, one may use strvcat, **Figure 9.2**:

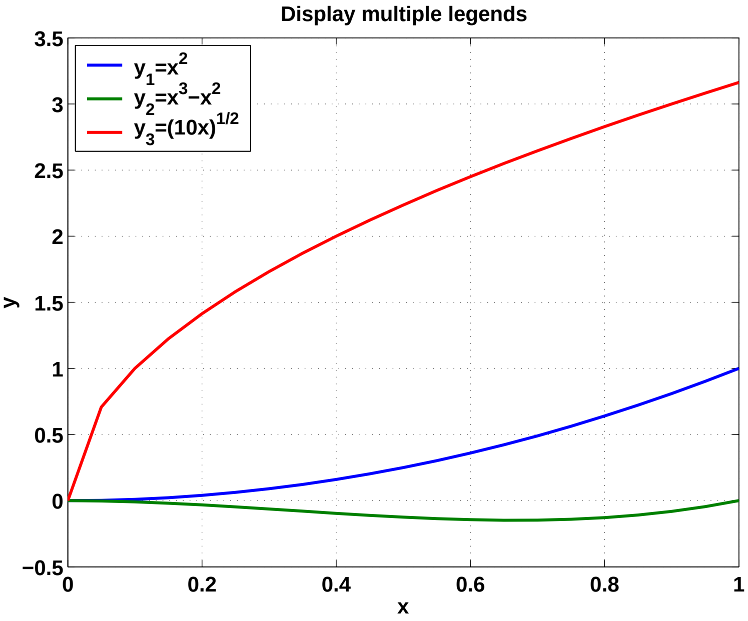


Figure 9.2: Displaying multiple legends.

```
% LEGENDS: a script to plot multiple legends
% TWP, E77N material, U.C. Berkeley, 2002-02-24
%
clear all
close all
x  = (0:.05:1)';
y=zeros(length(x),3);
y(:,1) = x.^2;
y(:,2) = x.^3-y(:,1);
y(:,3) = sqrt(10*x);
x= [x,x,x];
```

```
% Compose a string for multiple legends
s=[];
s1=sprintf('y_1=x^2');
s=strvcat(s,s1);
s1=sprintf('y_2=x^3-x^2');
s=strvcat(s,s1);
s1=sprintf('y_3=(10x)^{1/2}');
s=strvcat(s,s1);
figure
% Note that I am plotting 3 curves in 3 columns of x and y
plot(x,y,'LineWidth',2)
set(gca,'Fontsize',14,'FontWeight','b')
title('Display multiple legends')
xlabel('x')
ylabel('y')
% Display legends in the upper left corner
legend(s,2)
grid on
box on
%print -depsc legends.eps
```

9.6 Multiple Plots in One Figure

Here is an example that shows some of the features of basic plotting with MATLAB, **Figure 9.3**.

```
%
% Script to do several polynomial fits
% (of increasing order) to some dataset.
%
% Data points and fitting models are then
% plotted using subplot.
%
% Reference: W. J. Palm III,
% Introduction to Matlab 6 for Engineers,
% pp 277-278.
%
% Author: Ruben Juanes
% Last modified: 2/20/2001
clear all
close all

% data set
x = 1:9;
y = [5,6,10,20,28,33,34,36,42];
```

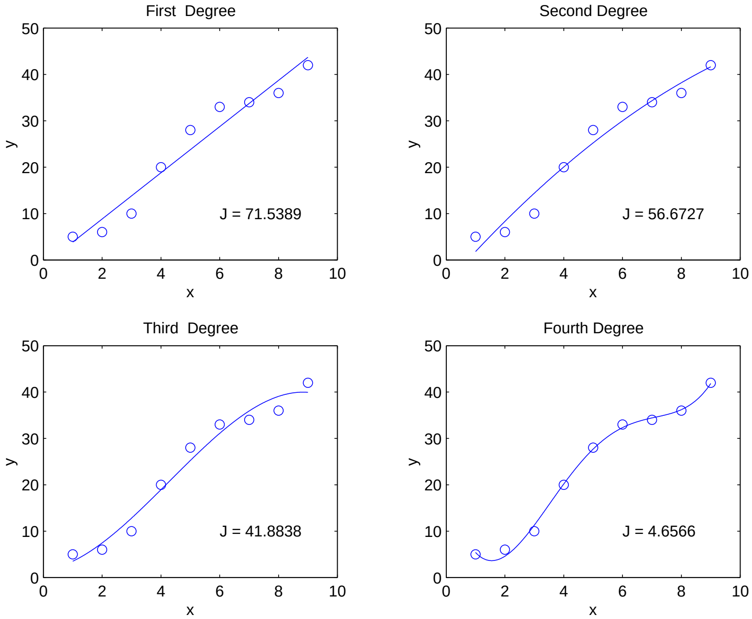


Figure 9.3: A two-by-two plot created with the subplot command.

```
% points for plotting and fitting models
xp = 1:0.01:9;
fitmod = ['First  Degree';...
          'Second Degree';...
          'Third  Degree';...
          'Fourth Degree'];

% loop over fitting models and plots
for k = 1:4
    coeff = polyfit(x,y,k);
    yp(k,:) = polyval(coeff,xp);
    J(k) = sum((polyval(coeff,x)-y).^2);
    subplot(2,2,k)
    plot(x,y,'o')
    hold on
    plot(xp,yp(k,:),'-')
    axis([0 10 0 50])
    title(fitmod(k,:))
    xlabel('x'),ylabel('y')
    text(6,10,['J = ',num2str(J(k))])
end
```

9.7 Surface Plots

Example 4 Consider the following two-dimensional function, which is shown in **Figure 9.4** for $(x, y) \in [-1, 1] \times [-1, 1]$:

$$f(x,y) = \begin{cases} \sin(\pi x)\sin(\pi y) & \text{if } x > 0, y > 0, \\ 0 & \text{if } x < 0, y > 0, \\ -|xy| & \text{if } y < 0. \end{cases} \qquad \square$$

The following MATLAB function evaluates $f(x, y)$ using **if** statements:

```
function [f] = pwfun(x,y)
%
if y < 0
    f = -abs(x*y);
else
    if x > 0
        f = sin(pi*x)*sin(pi*y);
    else
        f = 0;
    end
end
```

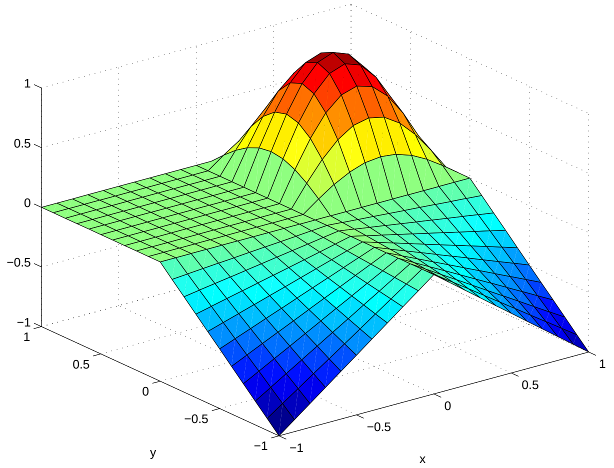


Figure 9.4: MATLAB plot of the function $f(x, y)$ defined in pwfun.m.

Example 5 Recall the function in Figure 9.4. Here is the naïve way to generate it (using the function `pwfun.m` and **for** loops):

```
x = -1:.1:1; y = x;
%
```

```
% loop to compute function
for ix = 1:length(x)
    for iy = 1:length(y)
        xx = x(ix);
        yy = y(iy);
        f(ix,iy) = pwfun(xx,yy);
    end
end
%
% plot
figure; surf(x,y,f')
```

Use `Help surf` to find out why `f'` is needed. □

Here is the vectorized code using array operations, which runs significantly faster. (Can you understand it?)

```
x = -1:.1:1; y = x;
%
% compute function
f = pwfun2(x,y);
%
% plot
figure; surf(x,y,f')
% ------------------------------------------
function [f] = pwfun2(x,y)
%
f = - abs(x'*(y.*(y<0))) ...
    + sin(pi*x.*(x>0))'*sin(pi*y.*(y>0));
```

9.8 Parametric Plots

Let us plot a three-dimensional logarithmic spiral:

$$\begin{aligned} t &\in [0, 20\pi] \\ x &= \ln(t)\cos(t) \\ y &= \ln(t)\sin(t) \\ z &= t^3 \end{aligned}$$

Function ParametricPlot.m does the job, and the result is shown in **Figure 9.5**.

```
function ParametricPlot()
% Generate a 3D parametric plot
% TWP , E77N material, U.C. Berkeley, 2002-02-24
%
close all
```

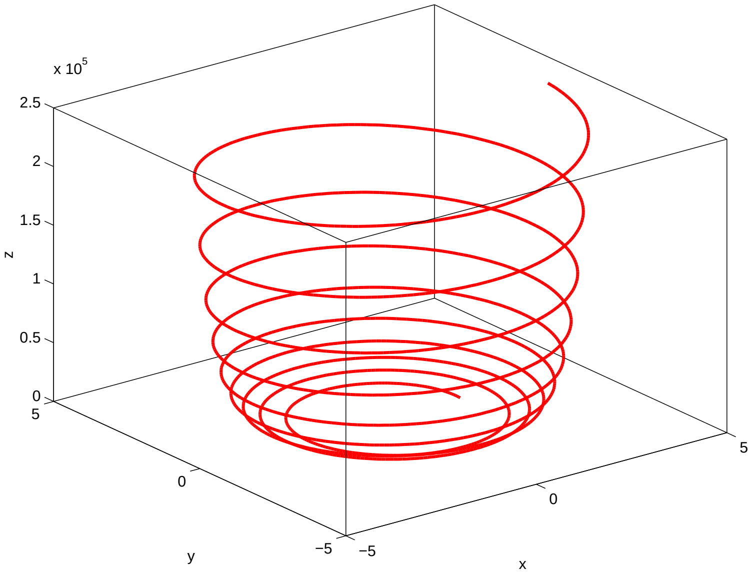


Figure 9.5: A three-dimensional spiral generated as a parametric curve.

```
clear all
clc
% Parameter vector
t=(1:0.005:10)*2*pi;
x=log(t).*cos(t);
y=log(t).*sin(t);
z=t.^3;
figure
hp=plot3(x,y,z);
set(hp, 'Linewidth',2);
set(hp, 'Color','r');
xlabel('x');
ylabel('y');
zlabel('z');
box on
rotate3d on
%print -depsc parametric.eps
```

Chapter 10

Mathematical Induction

10.1 What Are You Going To Learn?

You will learn how to use mathematical induction, an important method of science and philosophy specialized to mathematics and computer science.

10.2 Why Is It Important?

Mathematical induction the oldest way of making sure that a formula you just worked out, which you say is true as far as you can tell, is always true. An inductive proof assures that a formula you tested for several values of its arguments will *always* work for *all* valid values of these arguments.

Mathematical induction is one of the most important tools in proving algebraic identities and properties of algorithms (to be discussed in Lecture 11). Recursion, which will also be discussed in Lecture 11, is closely related to mathematical induction.

10.3 Important Definitions

Definition 7 Deduction is an act of deriving a conclusion by reasoning; inference in which conclusion follows necessarily from the premises [32]. □

Definition 8 Induction is an instance of reasoning from a part to a whole, from particulars to generals, or from the individual to the universal [32]. □

Definition 9 Iteration is the action of repeating the same set of operations or reapplying the same method. □

Definition 10 Recursion (from Latin to run back, to return) is (1) the action of a function calling itself, or (2) the definition of an object which refers to the object itself. □

Definition 11 A recursive definition of an expression proceeds by first specifying a *base case* of the items it applies to and then specifying the remaining items it applies to in terms of a relation (*recursive formula*) which any such item bears to an item to which the expression already applies. Thus the term "ancestor" may be defined recursively as follows: (1) both of a person's parents are ancestors of that person; (2) any parent of an ancestor of a person is also an ancestor of that person; (3) nothing else is an ancestor of a person [30]. □

Definition 12 Trees are the most important nonlinear structures that arise in computer algorithms. Let us define a tree as a finite set T of one or more *nodes* such that

1. There is one specifically designated node called the *root* of the tree, $\mathrm{root}(T)$, and

2. The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, T_2, \ldots, T_m$, and each of these sets in turn is a *tree*. The trees $T_1, T_2, \ldots, T_m$ are called the *subtrees* of the root.

The definition just given is recursive: We define a tree in terms of trees. There is no problem with logical circularity, because the trees with only one node must consist of only the root, and trees with $n > 1$ nodes are defined in terms of trees with fewer than n nodes. □

10.4 Essentials of Mathematical Induction

In a narrow sense induction is a method of generalization to all cases from the observation of particular cases. In a broad sense induction is a method for reasoning from some observed fact to a different fact not involved in the former. Relying on ARISTOTLE's testimony we can say that SOCRATES introduced induction (*epagōgē*) into Greek philosophy. In the Indian tradition, induction (*vyāptigraha*) played a significant role in GOTAMA's *Nyāyasutra*, the oldest systematic logical work in Sanskrit [5].

In ordinary induction[1], we examine a certain number of particular cases and then generalize. Here is how you could carry out a real-life induction process about the properties of hamburgers in your town. Suppose that you decided to inspect burgers in 100 coffee shops, by tasting one burger in each one of the shops. Based on the *inspection* of a *sample* of all coffee shops in your city, you would then draw conclusions about all other burger shops:

1. Coffee shop burger #1 was greasy.

2. Coffee shop burger #2 was greasy.

3. Coffee shop burger #3 was greasy.

[1] Based on the Web page of PETER SUBER, who is a professor of philosophy and chair of the Philosophy Department at Earlham College.

4. ...

100. Coffee shop burger #100 was greasy.

101. *Therefore*, all coffee shop burgers are greasy. (Or: The *next* coffee shop burger will be greasy.)

Inductions differ from valid deductions in one important way: in the induction, the conclusion contains information that was *not* contained in the premises. This is the source of the uncertainty of inductions. Conversely, deductions are certain because they tautologically restate their premises. Inductions are strengthened as confirming instances pile up. However, they can *never* bring certainty unless every possible case is actually examined, in which case they become deductions. Despite our experience with 100 greasy coffee shop burgers, the next one might be lean and bone dry.

"Mathematical induction" is unfortunately named, for it is unambiguously a form of deduction carried out in reverse. However, it has certain similarities to induction, which very likely inspired its name. It is like induction in that it generalizes to a whole class from a smaller sample. In fact, the sample is usually a sample of *one*, and the class is usually *infinite*.

Remark 14 Mathematical induction[2] is deductive; it works because the sample and a rule about the unexamined cases together give us information about every member of the class.

Conversely, the conclusion of a mathematical induction does not contain more information than was latent in the premises. Mathematical inductions therefore conclude with deductive certainty. □

10.4.1 Structure of Proof by Induction

You are probably quite sure that every even number is divisible by 2. But you have never examined every even number, and nobody else has either. So how do you know that some very big even number won't violate this rule? Why isn't the situation analogous to the coffee shop burgers? In both cases, aren't we moving from known cases to unknown cases?

The reason the even numbers are decisively different from coffee shop burgers lies in the logic of mathematical induction. We can prove that the smallest even number (2) is divisible by 2. This is our very small sample. And we can prove that the next even number after every number divisible by 2 will also be divisible by 2. This is our rule about the unexamined cases. That is enough to imply that the successor of 2, namely 4, will be divisible by 2, and its successor, 6, and its successor 8... and so on *ad infinitum.* (Obviously, if $2k$ is divisible by 2, then $2(k+1) = 2k+2$ is also divisible by 2.) This is how a small sample and a

[2] The first European on record to apply mathematical induction to rigorous proofs was Francesco Maurolico, in 1575. Pierre de Fermat made further improvement in the early 17th century. Mathematical induction also appears in the later writings of Blaise Pascal (1653). The phrase "mathematical induction" was apparently coined by A. De Morgan in the early 19th century.

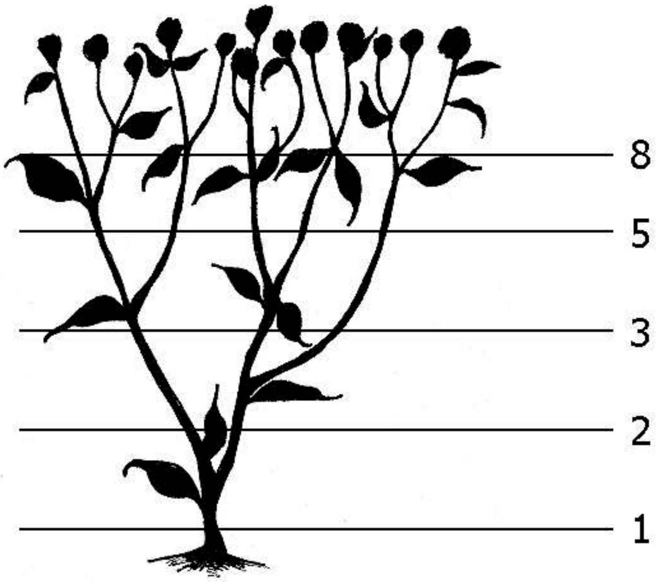


Figure 10.1: The number of stems at each horizontal level of development of this plant follows the FIBONACCI sequence. (Sketch by L. Patzek from a sneezewort photo.)

rule about unexamined cases can give us information about every case. This is how our knowledge of an infinite set of unexamined cases can be as certain as the conclusion of a valid deduction, *quite unlike* the conclusion of an ordinary induction.

The large-scale structure of a proof by mathematical induction is simple:

1. P(1): The theorem is true of the sample. (This requires a separate proof.)
2. $\mathrm{P}(k) \rightarrow \mathrm{P}(k+1)$: A rule tells us that if the theorem is true of the sample, then it is true of the unexamined members of a certain class. (This rule requires a separate proof.)
3. Therefore, the theorem is true of all the members of a certain class.

Here's how the inference looks when described in the special terminology of mathematical induction:

1. **Basis**. Prove that the theorem holds for the minimal case, i.e., for $i = 1$.
2. **Induction Step**. Prove that the property of complying with the theorem is "hereditary" and extends to all the successors of the minimal case.
   (a) If the rule of heredity tells us that the theorem is true of "descendant" cases if it is true of the "ancestor" cases, then the claim that it is true of the ancestor cases is called the *induction hypothesis*.

3. **Conclusion**. Together, Steps 1 and 2 imply that the theorem holds for all possible cases, i.e., the minimal case and all its successors. If you didn't use the *true* minimal case in Step 1, you have proved only that the theorem holds for your minimal case and its successors, not for all possible cases.

The *induction step* is the part that causes the most difficulties. It can take two forms, which correspond to two forms of mathematical induction:

- **Weak**: Prove that if the theorem holds for all cases at some arbitrary point n, then it holds for all cases at point $n + 1$.

- **Strong**: Prove that if the theorem holds for all cases up to some arbitrary point n, then it holds for all cases at point $n + 1$.

10.4.2 Example: A sum of Fibonacci Numbers

Figure 10.2: Fibonacci was born *circa* 1175 A.D. and died *circa* 1250 A.D.

The sequence

$$F(0) = 0,\ F(1) = 1,\ F(2) = 1,\ F(3) = 2,\ F(4) = 3,\ F(5) = 5,\ F(6) = 8, \ldots$$

in which each next number is the sum of the preceding two, plays an important role in this course, governs the behavior of plants (**Figure 10.1**), bees and stock markets, decides who is pretty and who is not, and which music is pleasant to listen to[3]. This famous sequence was published in 1202 by FIBONACCI.

The numbers in the sequence are denoted by $F(n)$, and formally defined as:

$$\begin{aligned} F(n+2) &= F(n+1) + F(n), \quad n \geq 0 \\ F(0) &= 0, \quad F(1) = 1 \end{aligned} \tag{10.1}$$

If we look at sums of the FIBONACCI numbers, we see:

$$\begin{aligned} F(0) &= 0 \\ F(0) + F(1) &= 0 + 1 = 1 \\ F(0) + F(1) + F(2) &= 0 + 1 + 1 = 2 \\ F(0) + F(1) + F(2) + F(3) &= 0 + 1 + 1 + 2 = 4 \\ F(0) + F(1) + F(2) + F(3) + F(4) &= 0 + 1 + 1 + 2 + 3 = 7 \\ &... \end{aligned} \tag{10.2}$$

etc., generating a new sequence $0, 1, 2, 4, 7, 12, \ldots$. What are these numbers? That is, what is $F(0) + F(1) + \cdots + F(n)$?

It shouldn't take long to hypothesize that this sum may be $F(n+2) - 1$. Let's test this proposition, i.e.,

$$\mathrm{P}(n) : F(0) + F(1) + ... + F(n) = F(n+2) - 1 \quad \text{for } n \geq 0. \tag{10.3}$$

Clearly P(1) is a true. Suppose we have checked the truth of the statement for $n = 0, 2, ..., k$ - this is our induction hypothesis.

What about $\mathrm{P}(k+1)$? Using the induction hypothesis to replace the first k terms in the summation, we obtain

$$\underbrace{F(0) + F(1) + \cdots + F(k)}_{\textbf{Hypothesis}} + F(k+1) = \underbrace{[F(k+2) - 1]}_{\textbf{Hypothesis}} + F(k+1), \tag{10.4}$$

but, from the definition of the FIBONACCI numbers,

$$[F(k+2) - 1] + F(k+1) = F(k+3) - 1. \tag{10.5}$$

Therefore

$$F(0) + F(1) + ... + F(k) + F(k+1) = F(k+3) - 1 \tag{10.6}$$

Thus, since P(1) is true and the assumption that $\mathrm{P}(k)$ is true implies the truth of $\mathrm{P}(k+1)$, we have that the truth of P(1) implies P(2) is true, in turn the truth of P(2) implies P(3) is true, in turn the truth of P(3) implies ..., i.e., $\mathrm{P}(n)$ is true for all n.

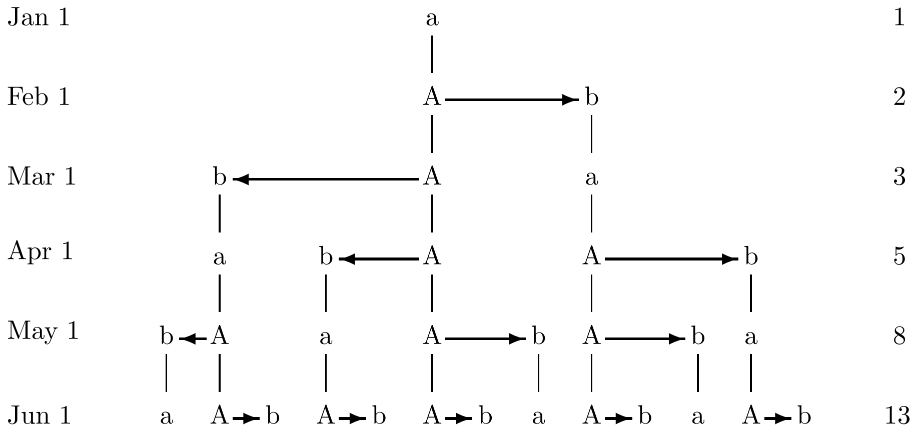


Figure 10.3: The ancestral tree of the rabbit family follows the FIBONACCI sequence. KEY: a = pair of young adult rabbits, A= pair of breeding adult rabbits, b = pair of baby rabbits. (Reproduced from Figure 5-1 in [14].)

10.5 Fibonacci and His Numbers

LEONARDO PISANO[4], better known under a literary name of FIBONACCI[5] was the greatest mathematician of Medieval Europe.

In 1202, he published a book, *Liber Abaci*[6], which changed European history. The book acquainted Europeans with the Indian-Arabic ciphers $0, 1, 2, \ldots$. It also contained the following problem of ever-lasting importance:

One pair of rabbits is born at time 0. After one month, this pair is mature and after a month, it gives birth to new pair of rabbits and continues to do so, i.e., every month a new pair of rabbits is born to the original pair. Moreover, each new pair of rabbits matures after one month and begins producing pairs of offspring every month after that *ad infinitum*. We assume that the rabbits live forever. What is the number of pairs after n months? See **Figure 10.3** for the answer. The exploding rabbit population problem was perhaps the first *population balance* envisioned eight hundred years ago by FIBONACCI.

10.6 Exercises

1. Prove by induction that $n^2 = 1 + 3 + \cdots + (2n - 1)$.

2. Prove by induction that $\dfrac{n(n+1)}{2} = 1 + 2 + \cdots + n$.

[3] For more information on the FIBONACCI numbers please look up the beautiful books by TRUDI H. GARLAND [14], and H. E. HUNTLEY [18].

[4] He did not have the last name, so people called him PISANO (of Pisa) because he was born in Pisa, where he settled again after extensive travels to Egypt, Syria and Greece.

[5] FIBONACCI = *Filius Bonaccii*, son of Bonaccio.

[6] Latin: *Book of the Abacus*.

3. Prove by induction that $\dfrac{n(n+1)(2n+1)}{6} = 1^2 + 2^2 + \cdots + n^2$.

4. Formulate and prove by induction a rule for the sums 1^2, $2^2 - 1^2$, $3^2 - 2^2 + 1^2$, $4^2 - 3^2 + 2^2 - 1^2$, $5^2 - 4^2 + 3^2 - 2^2 + 1^2$, etc.

5. Prove by induction that $F(n+m) = F(m)F(n+1) + F(m-1)F(n)$, for any positive integer m.

Chapter 11

Recursion, Part I

11.1 What Are You Going To Learn?

You will learn about recursion, and a way of describing recursive calls through trees. In Chapter 11 folder on fttp:\\petroleum.berkeley.edu, please look up the following MATLAB files:

BruteForceGCD.m is a naïve Greatest Common Divisor (GCD) algorithm, which calls `tryDivisor`.

tryDivisor.m implements recursive brute force approach to GCD.

DijkstraGCD.m is the smartest recursive GCD algorithm, due to Dijkstra.

EuclidGCD.m is the classical Euclid recursive GCD algorithm.

FactorialDemo.m demonstrates the different speeds of recursive and iterative calculation of $n!$. Calls the recursive function `nf` and the iterative function `IterativeFactorial` to perform the calculations.

nf.m is a recursive implementation of the $n!$ algorithm.

IterativeFactorial.m is an iterative implementation of the $n!$ algorithm. It compares a naïve implementation (`flag=1`) with a vectorized one (`flag=2`).

fib.m is a naïve recursive algorithm to calculate the FIBONACCI numbers. To calculate the n^{th} FIBONACCI number it makes 2^{n-1} calls.

FibonacciDemo.m demonstrates the Second Order Recurrence, `modfib`, written with a helper function, `helpFib`. The function `helpFib` "memorizes" the previous FIBONACCI numbers just as you would using a piece of paper. Because of the helper function, the number of calls is n, not 2^{n-1} as in the naïve `fib` algorithm.

modfib.m is the modified FIBONACCI algorithm that uses a helper function to memorize the prior two FIBONACCI numbers: $n = 0, 1, 2, \ldots$.

helpFib.m is the helper function for `modfib`. It memorizes the last two FIBONACCI numbers.

11.2 Why Is It Important?

Recursion is probably the most powerful technique of generating simple, efficient computer algorithms. It is also a way of thinking about problems in science and engineering, which is as old as the science itself.

Recursive functions are often simple and their correctness is easily seen. In practice, the speed of a computer implementation of a sloppily constructed recursion may suffer from the excessive number of cascading calls to the same function[1], but this is merely a technicality.

11.3 Useful Background

The concept of an algorithm is fundamental to all of computer science, and we should begin by carefully analyzing this concept. In Chapter One of his monograph [20], KNUTH discusses the origin and history of the concept of algorithm. Here we include a brief summary of KNUTH's discussion.

The word "algorithm" comes from the Middle Age term "algorism," the process of doing arithmetic using Arabic numerals. The term algorism itself came from the name of a famous Persian mathematician Abū ʻAbd Allāh Muhammad ibn Mūsā al-Khwārizmi - literally, "Father of Abdullah, Mohammed, son of Moses, native of Khwārizm[2]."

By 1950, the word algorithm was most frequently associated with EUCLID's algorithm, a process of finding the greatest common divisor of two numbers that appears in EUCLID's *Elements*[3]. As we will use it in the remainder of this chapter, it is instructive to present EUCLID's algorithm here:

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n ($m > n$), find their greatest common divisor, i.e., the largest positive integer that evenly divides both m and n.

E1. [Find reminder.] Divide m by n and let r be the reminder ($0 \leq r < n$).

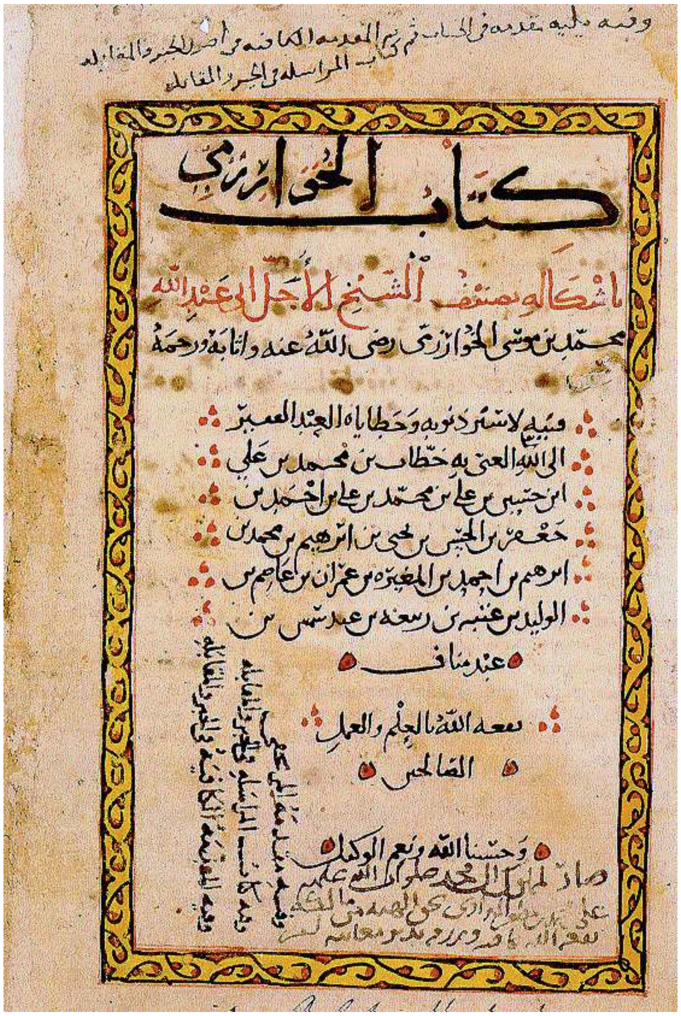
E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Reduce.] Set $m \leftarrow n$ (read this "set m to n"), $n \leftarrow r$ (n to r), and go back to step E1.

[1] You will get a warning message of stack overflow, or you will crash your computer.

[2] Today's Aral Sea in Central Asia

[3] Euclid of Alexandria (*circa* 330-275 B.C.) published this algorithm in Volume VII of his *Elements*, the most successful text ever written in the history of mathematics, see Chapter 29.



كتاب المراسله في الجبر والمقابله

كتاب الخوارزمي

اشكاله وتصنيف الشيخ الاجل ابي عبد الله

محمد بن موسى الخوارزمي رضي الله عنه واثابه ورحمه

وقفه لا ستر ذنوبه وخطاياه العبد الفقير

الى الله الغني به خطاب بن محمد بن علي

ابن حسين بن علي بن محمد بن علي بن احمد بن

جعفر بن الحسن بن يحيى بن ابرهيم بن محمد بن

ابرهيم بن احمد بن المغيره بن عمران بن عاصم بن

الوليد بن عتبه بن ربيعه بن عبد شمس بن

عبد مناف

نفعه الله بالعلم والعمل

الصالحين

وحسبنا الله ونعم الوكيل

صلى الله على محمد صلوات الله عليه

Figure 11.1: Al-Khwarizmi's book *Kitab al-Jabr wa al-Mukhabala* (*A Book of Algebra and Comparisons*), written in the first quarter of the ninth century, is the oldest book on algebra. In it al-Khwarizmi tried to provide a theory for the solution of all types of linear and quadratic equations. (Reproduced from the figure on page 186 in [10].)

Table 11.1: Steps in implementing a mathematical definition.

Algorithm	Procedure	Process
An idea of how to perform a mathematical activity	Step-by-step description of the activity (Many different procedures may describe the same algorithm)	Activity itself

11.4 Introduction

If we want to solve a mathematical problem, we need first to define what we want, i.e., we need a *mathematical definition.* For simplicity, let us start from the celebrated definition of n factorial:

$$n! \equiv \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases} \tag{11.1}$$

> The definition above only tells us what we want, but *not* how to get it. To implement a definition in practice we need the three other steps described in **Table 11.1**. The central step is to have an idea of how to perform the necessary calculations. This idea is encapsulated as a mathematical algorithm. Then we need to transcribe the idea into a series of steps necessary to reach our goal; we need to write a procedure. Finally, to get a specific answer, we need to execute the procedure on a computer, or carry out a process.

Example 6 Build a recursive procedure `nf` to calculate n factorial. In MATLAB this procedure will be a single function nf.m. **Figure 11.2** illustrates how this function works.

Because of the problem simplicity, an idea of how to obtain n factorial is at the same time a procedure to obtain it. We simply devise an algorithm straight from the mathematical definition and write it down in as a MATLAB function. In words, the algorithm can be described as:

Algorithm F (*Recursive factorial algorithm*). Given a positive integer n ($n \geq 1$), find n factorial, the product of all integers from 1 up to and including n. Define zero factorial as one.
F1. [Reduce.] Remember n (put it on a stack). Set $n \leftarrow n - 1$.

F2. [Go down.] Call factorial of n.
F3. [Is $n = 0$?] If $n > 0$, go to step F1. If n is 0, go to step F4.
F4. [Go backward and combine.] Take the product of all numbers stored in the stack vector, $N = [1, 2, \ldots, n]$. Terminate the algorithm.

```
function p = nf(n)
if (n<1)
   p = 1;
else
   p = n*nf(n-1);
end
```

Will this work? Let's try using the substitution model.

```
nf(0)
>> 1
nf(3)
3 * nf(2)
3 * 2 * nf(1)
3 * 2 * 1 * nf(0)
3 * 2 * 1 * 1
>> 6
```

This looks good; we have reduced the factorial problem to a sequence of smaller instances of the same problem. The idea of reducing a problem to itself is known as recursion.

Let's pick apart the code:

```
function p = nf(n)
% A RECURSIVE METHOD is a method that CALLS ITSELF
if (n<1)
   p = 1;
else
   p = n  *  nf(n-1);
%         ^ |-----------|
%         |       ^
%         |       |____Recursive call to itself
%         |
%         |___Combining step (used to combine other
%              calculations with the results of a
%              recursive call
end
```

Here are some hints for writing recursive procedures:

- Always identify the base case and associated result first.

- Make sure the recursive call is for a smaller problem (one "closer" to the base case)

Another way to think about the execution of a recursive procedure is with the "actors" model (or dataflow model). In this simple model, we draw a box for each procedure invocation and show the flow of data into and out of each box, Figure 11.2.

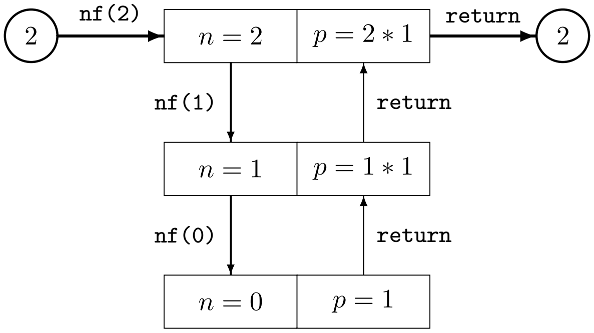


Figure 11.2: Flowchart of calculations in the recursive n factorial function `nf`.

Example 7 FIBONACCI numbers

Consider now the following definition of the FIBONACCI numbers:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases} \tag{11.2}$$

□

The FIBONACCI series is $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Let's write a recursive procedure for FIBONACCI numbers directly from the definition:

```
function s = fib(n)
% A naive recursive algorithm to calculate
% the Fibonacci numbers. Note n = 0,1,2,...
if (n<=1)
   s = n;
else
    % Two recursion calls
   s = fib(n-1) + fib(n-2);
end
fprintf('n=%4d, s=%4d\n',n,s);
```

This procedure combines results from two different recursive calls. Such an approach is sometimes known as "deep" recursion, or in other cases "divide and conquer."

Let's consider all the recursive calls for `fib(5)`, **Figure 11.3**. The function calls described in Figure 11.3 return the correct answer, but it takes a long time, since there are many of them.

As you can see a lot of work is repeated. For example, call [11] to `fib(3)` is a repeat of call [3], calls [8] and [12] to `fib(2)` repeat call [3], etc. In general, when n increases by 1, we roughly double the work and make about 2^{n-1} calls!

Let's think of another algorithm that is less wasteful. When we compute the Fibonacci series on paper, this is what do we do: 0 1 1 2 3 5 8 13 21 34 .... We look at the previous two numbers and add them to get the next number. We *do not* recompute the previous two numbers, we just write them down (or remember them).

Let's use this algorithm. We shall start the implementation by writing a helper procedure whose input parameters are n, a number k, and the values of $F(k)$ and $F(k-1)$. Think of k as the place we have come so far in writing out the series. When k reaches n, we are done.

Here is the code:

```
function s = modfib(n)
% The modified Fibonacci algorithm that uses
% a helper function to memorize the prior two
% Fibonacci numbers: n=0,1,2,...
%
if (n <=1 )
    s = n;
else
% Start recursion   k=0, F(1), F(0)
    s =      helpFib(n-1, 0,   1,    0);
end

function retval = helpFib(n, k, fibk, fibk1)
```

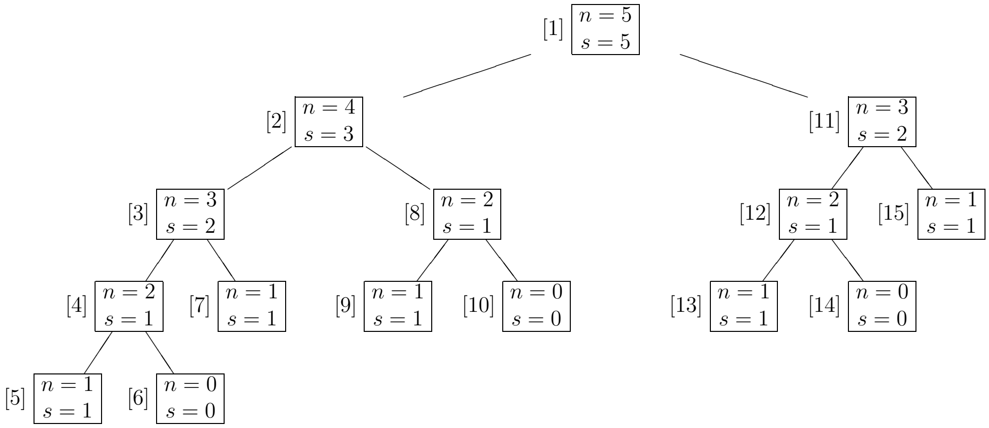


Figure 11.3: Flowchart of 15 recursive calls made by `fib(5)` function.

```
% The helper function for modfib memorizes
% the last two Fibonacci numbers
if (n == k)
   retval = fibk;
% Preorder printout
   fprintf('%2d %2d %4d %4d\n',n,k,fibk,fibk1);
else
%Preorder printout
   fprintf('%2d %2d %4d %4d\n',n,k,fibk,fibk1);
% Recursion            Next k   F(k)+F(k-1)  F(k-1)
   retval = helpFib(n,  k+1,    fibk+fibk1,   fibk);
end
%Postorder printout
%fprintf('%2d %2d %4d %4d\n',n,k,fibk,fibk1);
```

And this is how the code works:

```
>>modfib(6)
helpFib(6, 1, 1, 0)
helpFib(6, 2, 1, 1)
helpFib(6, 3, 2, 1)
helpFib(6, 4, 3, 2)
helpFib(6, 5, 5, 3)
helpFib(6, 6, 8, 5)
>>8
```

The new recursive procedure needs only n iterations to compute `fib(n)`, much better than 2^{n-1}. The lesson here is that being clever about an algorithm can yield significant savings.

11.5 More Complicated Recursion: Greatest Common Divisor (GCD)

Definition 13 Let us assume that we deal only with integers and $m \geq n > 0$. The greatest common divisor (GCD) of m and n is defined as the largest integer that divides both m and n with no remainder. □

11.5.1 GCD Algorithm 1: Brute Force

The idea is to try all integers from n down until finding one that divides m and n evenly. First, define `TryDivisor` that takes in m, n, and a guess. If the guess works, then it returns the guess. Otherwise, it tries a smaller guess:

```
function gcd = TryDivisor(m, n, g)
% Recursive brute force approach to GCD
% Try out all integers until one works...
if ((rem(m,g) == 0) & (rem(n,g) == 0))
    gcd = g;
else
    gcd = TryDivisor(m, n, g - 1);
end
```

Now we can reduce GCD to TryDivisor:

```
function retval = BruteForceGCD(m, n)
% Naive Greatest Common Divisor Algorithm calls tryDivisor
% Use n as our first guess
retval = tryDivisor(m, n, n);
```

An example using the substitution model: This algorithm obviously works, but for large numbers, the calculations could take a while. Let us consider another algorithm.

11.5.2 GCD Algorithm 2: Euclid's Algorithm

Definition 14 In this section "/" denotes *integer division.* For example, take $m = 22$ and $n = 4$:

$$22/4 \equiv \text{ integer part of } \frac{22}{4} = 5, \tag{11.3}$$

and the reminder is 2. In MATLAB, `floor(22/4) = 5` and `rem(22,4) = 2` □

Euclid's GCD algorithm dates from c. 300 B.C., and it is based on the following fact:

$$\text{for } m \geq n > 0, \quad \gcd(m,n) = \left\{ \begin{array}{c} n \text{ if } n \text{ divides } m \text{ with no remainder} \\ \gcd\left(n, \text{ reminder of } m/n\right) \end{array} \right. \tag{11.4}$$

Why is this true? We can rewrite m as follows:

$$\begin{aligned} m &= n\,(m/n) + \text{ reminder}\,(m/n)\,, \text{ e.g.,} \\ 22 &= 4 * 5 + 2 \end{aligned} \tag{11.5}$$

Now any integer d (2) that divides both m (22) and n (4), must divide the first term on the right hand side with no remainder, since it is the product of n and an integer. Therefore, d must also divide the second term since d divides m and m is the sum of the two terms:

$$\begin{aligned} m/d &= (n/d)\,(m/n) + \text{ reminder}\,(m/n)\,/d, \text{ e.g.,} \\ 22/2 &= (4/2) * (5) + (2)/2 \end{aligned} \tag{11.6}$$

Since any divisor common to m and n must divide the remainder of m/n, we know that, in particular, the GCD does, since it is a common divisor. It just happens to be the greatest such divisor.

So by taking the `gcd(n, remainder of m/n)`, we can "close in quickly" on the GCD of m and n. (This is extremely clever - you would not be expected to come up with something like this in an algorithm question for the E77N midterm examination.)
Now we can write:

```
function retval = EuclidGCD(m, n)
% Classical Euclid recursive GCD algorithm
if (rem(m, n) == 0)
    retval = n;
else
    retval = EuclidgGCD(n, rem(m, n) );
end
```

An example using the substitution model:

```
>>EuclidGCD(468, 24)
EuclidGCD(24, 12)
>>12

>>EuclidGCD(135, 19)
EuclidGCD(19, 2)
EuclidGCD(2, 1)
>>1
```

Euclid's GCD algorithm is very fast, but division (taking remainders) is a more time-consuming operation than simple addition and subtraction. Can we devise a GCD algorithm that doesn't use division?

11.5.3 GCD Algorithm 3: Dijkstra's Algorithm

Dijkstra is a Dutch mathematician and computer scientist who had the following idea:

If $m > n$, GCD(m,n) is the same as GCD(m-n,n).

Why? If m/d and n/d both leave no remainder, then $(m-n)/d$ leaves no remainder. (Again, this is *very*, *very* clever. Most graduate students probably could not come up with this if they have not already seen it.)

Dijkstra's idea leads to the following algorithm:

$$\text{for } m \geq n > 0, \ \gcd(m,n) = \begin{cases} m & \text{if } m = n \\ \gcd(m-n, n) & \text{if } m > n \\ \gcd(n-m, m) & \text{if } n > m \end{cases} \tag{11.7}$$

Here is the MATLAB implementation:

```
function retval = DijkstraGCD(m, n)
% The smartest recursive GCD algorithm, due to Dijkstra
if(m == n)
    retval = m;
elseif (m > n)
    retval = gcd(m-n, n);
else
    retval = gcd(m, n-m);
end
```

An example using the substitution model:

```
>>DijkstraGCD(468, 24)
DijkstraGCD(444, 24)
DijkstraGCD(420, 24)
...
DijkstraGCD(36, 24)
DijkstraGCD(12, 24)     (Now n is bigger)
DijkstraGCD(12, 12)     (Same)
>>12
```

Dijkstra's algorithm does accomplish the calculation with no division.

11.6 Summary and More Solved Examples

Recursion is closely related to mathematical induction. As we have learned in Lecture 10, one has to define a base case and recursive formula. For example, in the identity $n^2 = 1 + 3 + \cdots + (2n-1) =: S(n)$, we have

(a) Base case: $S(1) = 1$.

(b) Recursive formula: $S(n) = S(n-1) + (2n-1),\ n \geq 2$.

A recursive function is a function which calls itself, by means of some *recursive formula*, until the *base case* is encountered. Once this happens, the function returns (recursively) to the calling procedure. The program creates a new internal variable in every function call and puts on the MATLAB *stack*. In MATLAB, up to 500 recursive calls can be made without *stack overflow*. Here is how it works for the example above. The code is:

```
function S = n2(n)
% Finds the square of an integer n
% Recursive implementation
if (n == 1)
    S = 1;
else
    S = n2(n-1) + 2*n-1;
end
```

Figure 11.4 below shows a flowchart of how the calculations are actually done in the computer.

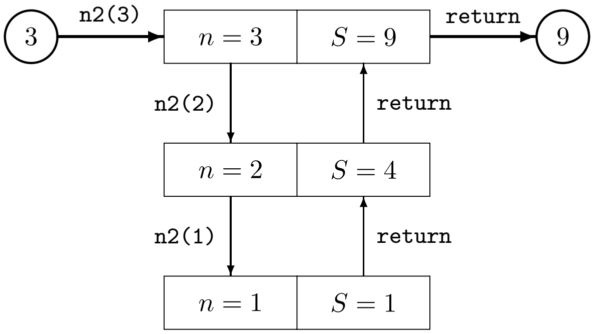


Figure 11.4: Flowchart of calculations in a recursive function

In what follows we present more examples. We include the recursive and the iterative version, so that you can compare them.

Example 8 Again, the identity $n^2 = S(n) := 1 + 3 + \cdots + (2n - 1)$. □

```
function s = nsqr_rec(n)
% Square of an integer n
% Recursive implementation
if (n == 1)
    s = 1;
else
    s = nsqr_rec(n-1) + 2*n-1;
end
```

```
function s = nsqr_itr(n)
% Square of an integer n
% Iterative implementation
s = 0; for k = 1:n
    s = s + 2*k-1;
end
```

Example 9 Sum of an arithmetic series

$$S(n) := \sum_{k=1}^{n} [x + (k-1)a] = S(n-1) + (x + (k-1)a)$$ □

```
function s = arith_rec(x,a,n)
% Sum of an arithmetic series
% Recursive implementation
if n == 1
    s = x;
else
    s = arith_rec(x,a,n-1) + ...
          x+(n-1)*a;
end
```

```
function s = arith_itr(x,a,n)
% Sum of an arithmetic series
% Iterative implementation
s = 0; for k = 1:n
    s = s + x+(k-1)*a;
end
```

Example 10 Factorial of an integer $n! = P(n) := \prod_{k=1}^{n} k$ □

```
function p = fact_rec(n)
% Factorial of an integer n
% Recursive implementation
if (n == 0)
    p = 1;
else
    p = n*fact_rec(n-1);
end
```

```
function p = fact_itr(n)
% Factorial of an integer n
% Iterative implementation
p = 1; if (n > 0)
    for k = 2:n
        p = p*k;
    end
end
```

Example 11 Calculation of the greatest common divisor of two integers using Euclid's algorithm. □

```
function gcd = euclid_rec(m,n)
% Greatest common divisor
% Recursive implementation
r = rem(m,n); if (r == 0)
    gcd = n;
else
    m = n;
    n = r;
    gcd = euclid_rec(m,n);
end
```

```
function gcd = euclid_itr(m,n)
% Greatest common divisor
% Iterative implementation
r = rem(m,n); while r ~= 0
    m = n;
    n = r;
    r = rem(m,n);
end gcd = n;
```

11.7 Exercises

1. Write a recursive function to multiply m times n.
2. Write a recursive function to calculate $\sqrt{n}$.
3. Write an iterative function `fib_itr(n)` to generate the FIBONACCI sequence with n terms, and compare its performance with the naïve recursive implementation `fib.m`, and the sophisticated implementation `modfib.m`. Use sufficiently large values of $n \leq 500$.

Chapter 12

Trees

12.1 What Are You Going To Learn?

You will learn about oriented ordinary trees and forests, and binary trees which are altogether different.

12.2 Why Is It Important?

Trees, forests and binary trees form three fundamental information structures in computer science[1]. They arise from many sources and are, therefore, important in computer algorithms. Here we will follow closely the monograph by Knuth [20].

12.3 Useful Definitions

Tree structure means "branching" relationship between nodes. If you recall Lecture 10, the most appropriate definition of a tree is *recursive*: a finite set of one or more nodes such that

(a) there is a special node called the *root* of the tree, and

(b) the remaining nodes are partitioned into disjoint sets, each of which is in turn a tree.

It follows from the definition that every node of a tree is the root of some subtree contained in the whole tree. More specific definitions follow:

Degree of a node: number of subtrees of that node.

Leaf (or terminal node): a node of degree zero.

[1] The other types of information structure are lists and graphs.

Branch node: a node that is not a terminal node.

Children of a node: the roots of the subtrees of that node.

Parent: the root of the children subtrees.

Level: the level of root(T) is zero; the level of any other node is one higher than that node's level with respect to the subtree of root(T) containing it.

Forest: a set (usually an ordered set) of zero or more disjoint trees. For example, nodes of a tree form a forest when the root is excluded.

The very nature of computer representation defines an implicit ordering for a tree, so in most cases ordered trees are of interest to us.

Trees can be drawn in many ways. There are four principle alternatives in how a tree can be drawn depending on the position of the root (draw them as an exercise). How you draw a tree, is significant. Our natural inclination is to draw the root at the bottom as in a living tree, see Figure 10.1 in Chapter 10. But after years of testing, KNUTH [20] has found out that people are used to drawing algorithmic trees down, not up. Consequently, we will adopt here the *root-at-the-top convention*, **Figure 12.1**.

12.3.1 Binary trees

In a *binary tree* each node has at most two subtrees; and when only one subtree is present, we distinguish between the left and right subtree. Therefore, a binary tree is *not* a special case of a tree. It is a different concept altogether. For example, the binary trees in **Figure 12.2** *are* different, although as generic trees these two diagrams would represent the same tree structure. Also, a binary tree can be *empty*; an ordinary tree cannot.

12.3.2 Traversing binary trees

The notion of *traversing* or "walking through" a tree is essential. This is a method of examining the nodes of a tree systematically so that each node is visited exactly once. Three principal ways may be used to traverse a binary tree, which are defined recursively as follows:

Preorder traversal

1. Visit the *root*
2. Traverse the *left* subtree
3. Traverse the *right* subtree

Inorder traversal

1. Traverse the *left* subtree
2. Visit the *root*

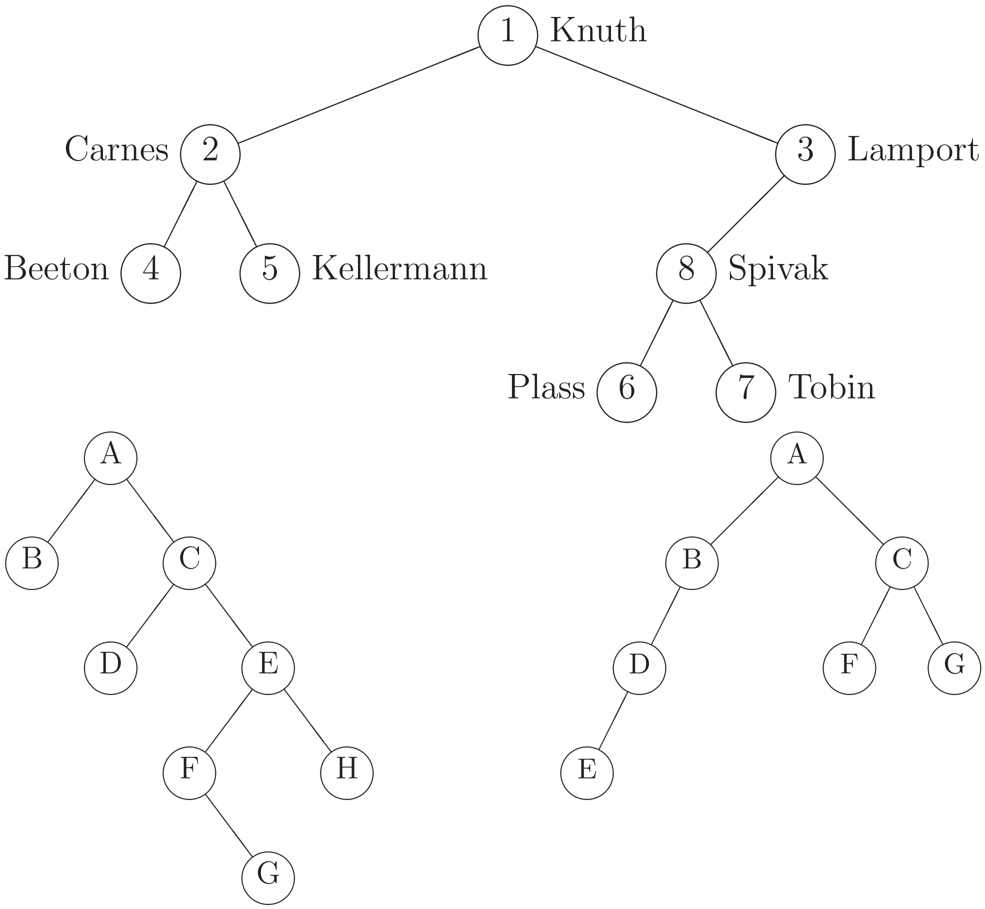


Figure 12.1: This is an example of a tree that includes labels (top), and two trees that don't. The *level* of the tree *root*, node 1 and A, is *zero*. Node 1 has two *subtrees*, $\{2, 4, 5\}$ and $\{3, 8, 6, 7\}$. The level of nodes 2 and 3, or B and C is *one*. The level of nodes $4, 5, 8$, or D, E, or D, F, G is *two*, etc. Nodes $4, 5, 6, 7$ are *terminal* or *leafs*. Nodes 4 and 5 are *children* of node 2. Node 8 is a *parent* of subtrees 6 and 7 and its *degree* is two.

3. Traverse the *right* subtree

Postorder traversal

1. Traverse the *left* subtree
2. Traverse the *right* subtree
3. Visit the *root*

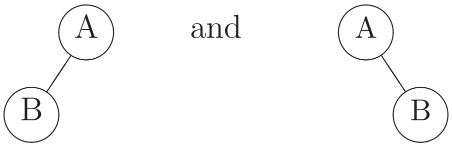


Figure 12.2: These two binary trees are different. From left to right, the root has an empty right subtree in one case and the left one in the other. As ordinary trees, these two binary trees would be equivalent.

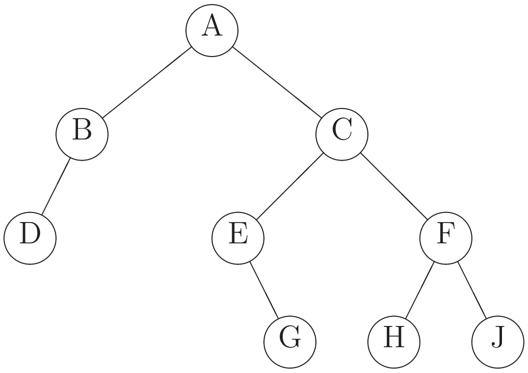


Figure 12.3: Binary tree used in Example 12. Preorder: $A\ B\ D\ C\ E\ G\ F\ H\ J$, Inorder: $D\ B\ A\ E\ G\ C\ H\ F\ J$, and Postorder: $D\ B\ G\ E\ H\ J\ F\ C\ A$.

Example 12 Consider the binary tree in **Figure 12.3**. We can write the list of nodes using each of the methods for traversing a tree:

$$\begin{aligned} \text{Preorder:}&\quad A\ B\ D\ C\ E\ G\ F\ H\ J \\ \text{Inorder:}&\quad D\ B\ A\ E\ G\ C\ H\ F\ J \\ \text{Postorder:}&\quad D\ B\ G\ E\ H\ J\ F\ C\ A \end{aligned}$$ □

Example 13 Any algebraic formula defines an implicit tree structure [20]. **Figure 12.4** shows a binary tree corresponding to the arithmetic expression $a - b(c/d + e/f)$. □

12.4 Traversing Forests of Trees

The ways of traversing the binary trees can be recast [20], pages 334-336, in terms of forests and, therefore, trees. There is *no* simple analog of the *inorder* traversal sequence, since there is no obvious place to insert the root among its descendants. The other two types of traversal are carried over as follows:

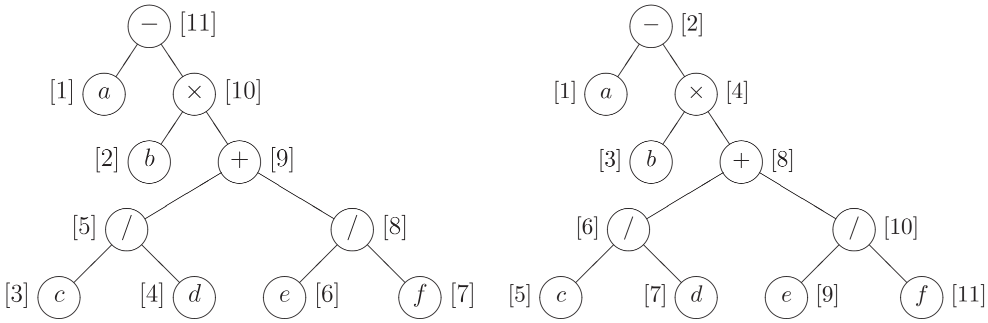


Figure 12.4: Binary tree representation of the formula in Example 13. **Left**, *postorder* traversal as in an HP calculator: a, `Enter`, b, `Enter`, c, `Enter`, d, $\div$, e, `Enter`, f, $\div$, $\times$, $-$. **Right**, *inorder* traversal as in a TI calculator: a, $-$, b, $\times$, (c, $\div$, d, $+$, $e \div f$).

Preorder traversal

1. Visit the *root* of the first tree
2. Traverse the *subtrees* of the first tree
3. Traverse the remaining *trees*

Postorder traversal

1. Traverse the *subtrees* of the first tree
2. Visit the *root* of the first tree
3. Traverse the remaining *trees*

The two ways of forest traversal are shown in **Figure 12.5**.

12.5 Recursion Trees

Recursion trees are useful in understanding more complicated recurrences. For example the action of the recursive n factorial function in previous lecture was to create a binary tree on the stack, **Figure 12.6**, which was traversed in *postorder*.

Recall, for example, the sequence of Fibonacci numbers, introduced in Lecture 7:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ldots,$$

in which each number is the sum of the preceding two. This sequence can be formally defined by the recursive formula that starts from $n = 0$:

$$F(0) = 0; \quad F(1) = 1; \quad F(n) = F(n-2) + F(n-1),\ n \geq 2.$$

Also, recall a (very inefficient) recursive function to evaluate Fibonacci numbers:

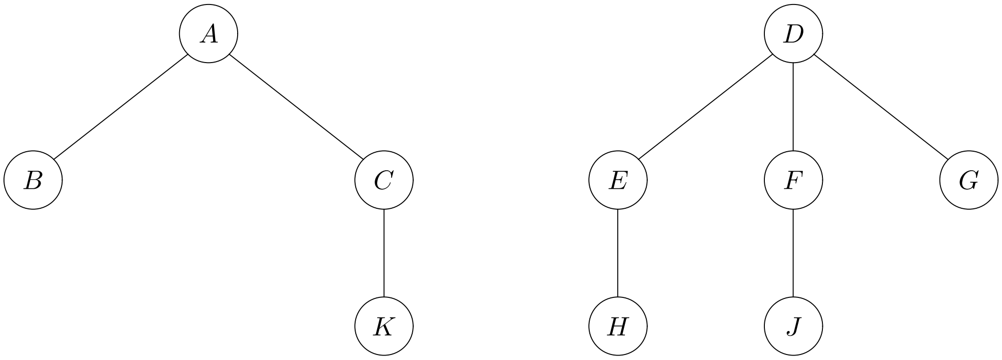


Figure 12.5: Forest traversal. Preorder: $A\ B\ C\ K$; $D\ E\ H\ F\ J\ G$. Postorder: $B\ K\ C\ A$; $H\ E\ J\ F\ G\ D$.

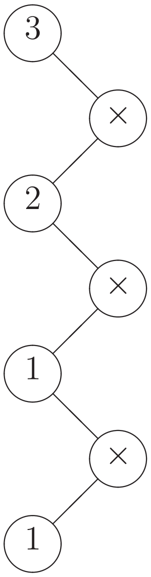


Figure 12.6: Flowchart of the four recursive calls made by `nf(3)` function.

```
function s = fib(n)
if (n < =1)
    s = n;
else
    s = fib(n-2) + fib(n-1);
```

```
end
```

This is what happens when fib(5) is called:

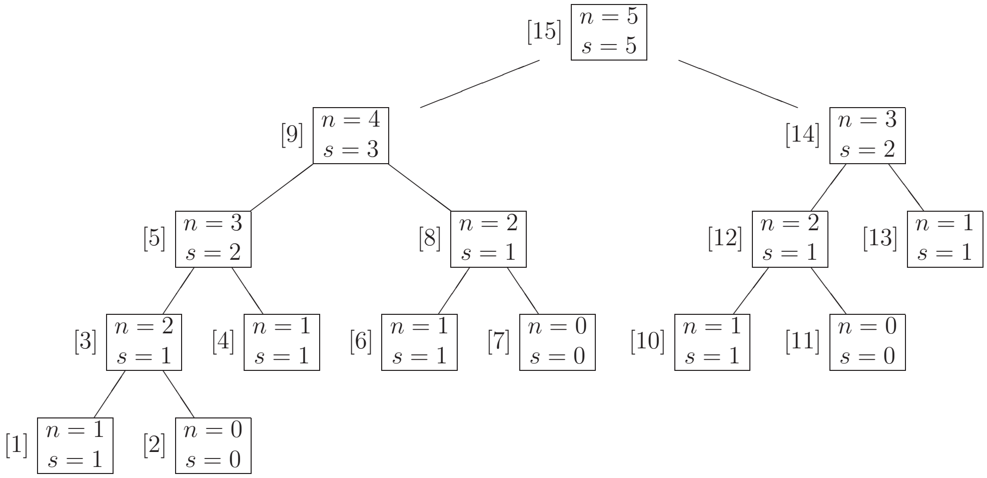


Figure 12.7: Flowchart of 15 recursive calls made by `fib(5)` function.

A recursion tree is helpful in explaining the flow of the recursive function calls. Each node in the recursion tree represents one call to the function. The root of the tree is the first call. The children of the root are the recursive calls that the root makes, and so on. In **Figure 12.7** we show the binary tree representation of the recursive implementation of the Fibonacci numbers for $n = 5$. At the center of each node we write the current value of n and $s = F(n)$. At the left of each node we write the index of the corresponding function call. A total number of 15 calls are generated. Note that the sequence of function calls is precisely the *postorder traversal* of the tree.

Chapter 13

Finding Roots of Nonlinear Functions

13.1 What Are You Going To Learn?

You will learn about two most popular iterative methods used to find zeros (roots) of nonlinear functions: the bisection and NEWTON's method. You will also learn about the approximation errors and the rate of convergence.

In Chapter 13 folder on fttp:\\petroleum.berkeley.edu, please look up the following MATLAB files:

domains.m plots the naïve domains of attraction of all three roots of $f(z) = z^3 - 1$, where z is the complex variable.

truedomains.m finds the complex roots of $f(z) = z^3 - 1$ for a very fine grid of initial guesses on an (x, y)-square between -1 and 1 (a unit square on the complex plane), thus generating the beautiful Julia fractal for the true domains of influence of each root.

rootbisection.m finds a previously bracketed root of a function using the bisection method; based on the *Numerical Recipes in C*.

rootnewton.m using a close enough guess for a root, find the root using Newton's method; based on the *Numerical Recipes in C*.

rootsafe.m in an interval $[x_1, x_2]$ containing a root, find the root using a combination of bisection and Newton's method; based on the *Numerical Recipes in C*.

showbisection.m demonstrates how the bisection method works.

shownewton.m demonstrates how Newton's method works.

funcd.m is a cubic polynomial used by the `showbisection` and `shownewton` demos.

13.2 Why Is It Important?

Many interesting problems in engineering and science are nonlinear. Almost all solutions to these problems involve finding roots of nonlinear functions. In many practical problems, we end up with a large set of *nonlinear algebraic equations*, whose roots we must find. Often these equations are solved using variants of NEWTON's method.

13.3 Introduction

The basic problem is to solve (numerically) a nonlinear equation

$$f(x) = 0.$$

The solution or solutions x are called the *roots* of function f, and thus the name *root finding*. In this course we shall not deal with the much more complicated problem of solving *systems* of nonlinear equations.

You should take the following steps when trying to solve a nonlinear equation:

Algorithm R (*Root-finding algorithm* for a function of one variable $f(x)$).
R1. [Explore.] Plot the function $f(x)$ with different $x-$ and $y-$ranges.
R2. [Bracket the root.] Identify an interval $[a, b]$ which brackets the desired root.
R3. [Guess the root.] Provide an initial guess x_0 of the root.
R4. [Refine estimate.] Refine your initial guess iteratively using a numerical method appropriate[1] for $f(x)$ until a convergence criterion is met.

Here we shall study two numerical methods only, namely, bisection (Section 13.4) and NEWTON (Section 13.5).

13.3.1 A Sample Nonlinear Function

For simplicity, we shall consider the following polynomial function:

$$w = f(z) = z^3 - 1 \tag{13.1}$$

This function can be also written as

$$\begin{aligned} f(z) &= z^3 - 1 = (z-1)(z^2 + z + 1) \\ &= (z-1)\left(z - \frac{-1 + i\sqrt{3}}{2}\right)\left(z - \frac{-1 - i\sqrt{3}}{2}\right) \end{aligned} \tag{13.2}$$

[1] The function $f(x)$ should be at least continuous, but different numerical methods require different degrees of smoothness of $f(x)$.

Thus, as expected, the function $w = z^3 - 1$ has exactly three roots:

$$\begin{aligned} z_1^* &= 1 + i0 \\ z_2^* &= -\frac{1}{2} + i\frac{\sqrt{3}}{2} \\ z_3^* &= -\frac{1}{2} - i\frac{\sqrt{3}}{2} \end{aligned} \tag{13.3}$$

These roots are shown in **Figure 13.1** as points on the complex plane $z = x+iy$. The lines passing through the origin and each of the three roots bisect the three colored regions in Figure 13.1. Each region is the *basin of attraction*[2], of the root along its bisector. As we shall learn in Section 13.5, Newton's method will converge to a given root if the starting point is well inside the root's region of influence. If the starting point is near a boundary between two regions, the story is much more complicated. For simplicity, here we shall find numerically only the real root, $z_1^* = 1 + 0i = 1$, by using the bisection and Newton's method.

13.3.2 Function Handles

An iterative root finding method is independent of a suitable function whose root we want to find. Therefore, we should design the root-finding functions so that they can be called with any suitable function $f(z)$.

In MATLAB this is achieved by using a pointer to a particular $f(z)$, say `f1(z)`, and then calling, e.g., the bisection method function with that pointer:

```
fhandle = @f1;
% Interval bracketing the root and approximation tolerance
a=0; b=2; tol=1e-04;
root = rootbisection(fhandle,a,b,tol);
```

Inside the bisection function, we use a special MATLAB function that calculates the values of a function through a pointer to it. This function is

```
feval(fhandle,argumentvalues).
```

13.4 Bisection method

Given an interval $[a, b]$ containing the root, the bisection method cannot fail. The idea is very simple:

Algorithm B (*Bisection algorithm* for a function of one variable $f(x)$).

[2] Arthur Cayley recognized that if we already know the roots of a function, Newton's method suggests another problem: which initial guesses iterate to which roots [4], i.e., what are the basins of attraction of the roots. He determined the basins of attraction for the quadratic equation, $z^2 - 1$, and claimed to have found the basins of attraction for the cubic equation $z^3 - 1$.

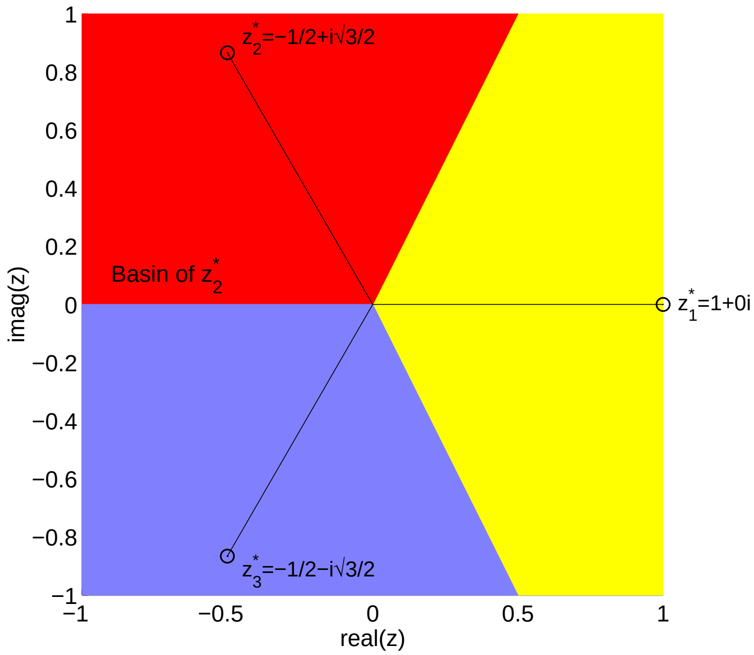


Figure 13.1: The roots of $f = z^3 - 1$, and their rough basins of attraction for NEWTON's method. This picture illustrates the state of knowledge near the end of the 19th century.

B1. [Is the root bracketed?] Choose an interval $[a, b]$. Check that $f(a)$ and $f(b)$ have different signs, so that somewhere in between the value of the function will be zero.
B2. [Evaluate.] Evaluate the function at the interval's midpoint and examine its sign.
B3. [Refine estimate.] Use the midpoint to replace the interval's limit where the function sign is the same as in Step **B2**.
B4. [Is the estimate good enough?] If the size of the interval is smaller than the tolerance, then stop. Otherwise go back to Step **B2**.

Appendix 13.6 contains a MATLAB implementation of the bisection subroutine in the *Numerical Recipes*.

The bisection method exhibits *linear convergence*, i.e., if e_n is the size of the interval at iteration n, then

$$e_{n+1} = \frac{1}{2}e_n.$$

Example 14 Convergence of the Bisection Method. We solve the equation $f(x) = x^3 - 1 = 0$ (**Figure 13.2**) with the initial interval $[-1, 2]$ and a tolerance of 10^{-4} for convergence. **Figure 13.3** represents the sequence of intervals and **Figure 13.4** the evolution of the interval width (error).

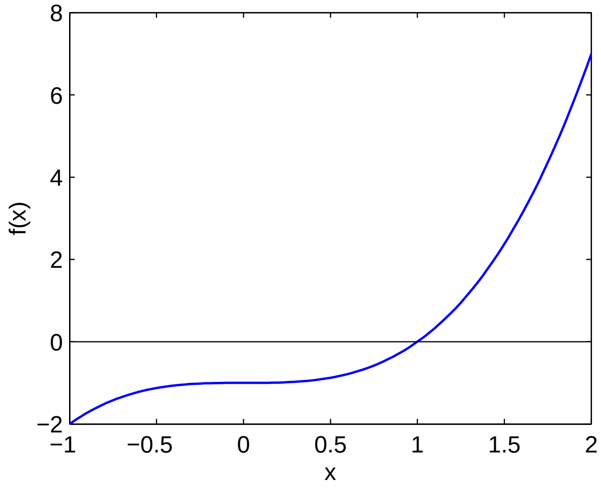


Figure 13.2: Function $f(x) = x^3 - 1$

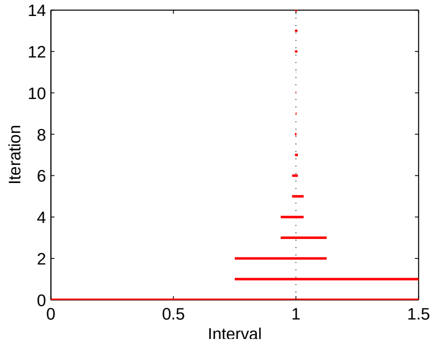


Figure 13.3: Sequence of intervals

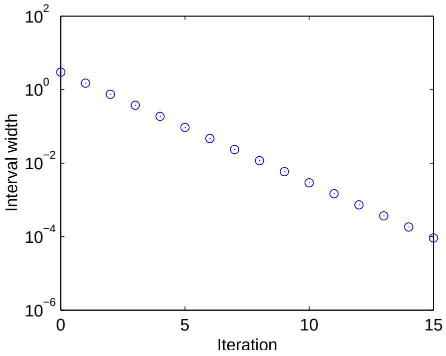


Figure 13.4: Evolution of error

13.5 Newton's method

This is probably the most celebrated method for one-dimensional root finding[3]. It requires the evaluation of *both* the function $f(x)$ and its derivative $f'(x)$, at arbitrary points x. There are two possible interpretations:

[3] It appeared in NEWTON's *Method of Fluxions*, written in Latin in about 1671.

Geometrical. NEWTON's method consists of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see **Figure 13.5**).

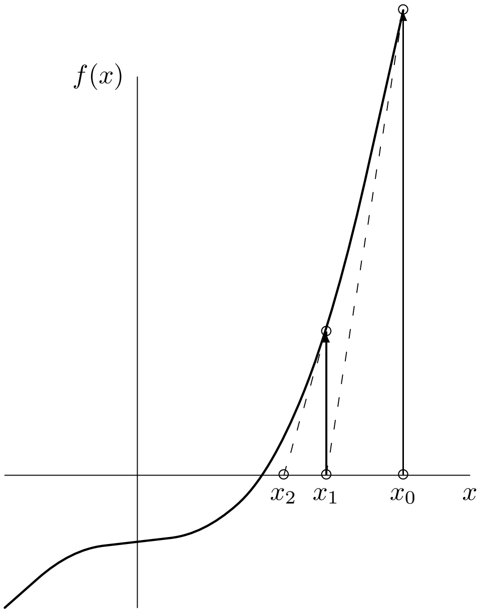


Figure 13.5: Geometrical interpretation of NEWTON's method

Algebraic. The method derives from TAYLOR series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \mathcal{O}(\delta^2),$$

where $\mathcal{O}(\delta^2)$ are small terms of second order. If we are sufficiently close to the solution, so that higher order terms are unimportant, i.e., $f(x+\delta) \approx 0$:

$$\delta = -\frac{f(x)}{f'(x)}.$$

Thus, we can describe NEWTON's method concisely in the following steps (a MATLAB implementation is given in Appendix 13.7):

Algorithm N (*Newton's algorithm* for a function of one variable $f(x)$).
N1. [Guess the root.] Set $i \leftarrow 0$. Take an initial guess x_i *sufficiently* close to the root.
N2. [Refine.] Given an approximation x_i to the root, take

$$x_{i+1} = x_i - \delta_i, \qquad \delta_i = \frac{f(x_i)}{f'(x_i)}.$$

N3. [Is the estimate good enough?] If the correction δ_i is smaller than the tolerance, then stop. Otherwise set $i \leftarrow i+1$ and go back to Step **N2**.

Definition 15 Error: Let x_i approximate a real number x (here the root). The most useful measures of the accuracy of x_i are its absolute error

$$e_{\text{abs}}(x_i) = |x_i - x|$$

and its relative error

$$e_{\text{rel}}(x_i) = \frac{|x_i - x|}{|x|}, \quad x \neq 0$$

In scientific computation, where answers to problems may vary by many orders of magnitude, the relative error is of most interest, because it is scale-independent. □

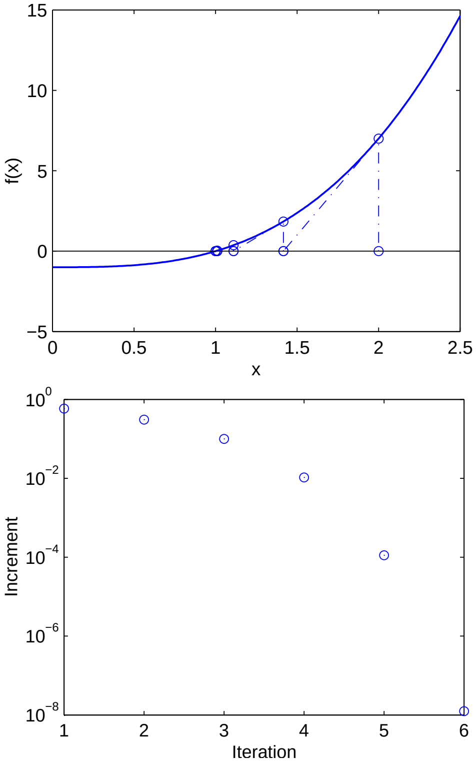


Figure 13.6: NEWTON's method with $x_0 = 2$. **Left**: Sequence of approximations. **Right**: Magnitude of correction.

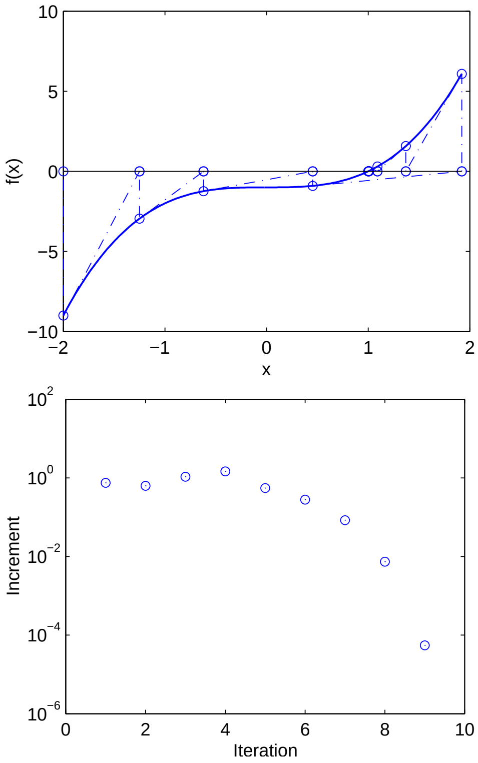


Figure 13.7: Newton's method with $x_0 = -2$. **Left**: Sequence of approximations. **Right**: Magnitude of correction.

The power of Newton's method lies in its rate of convergence. By Newton's formula, we have

$$
\begin{aligned}
(x_{i+1} - x) &= (x_i - x) - \frac{f(x_i)}{f'(x_i)} \\
e_{i+1} &= e_i - \frac{f(x_i)}{f'(x_i)}.
\end{aligned}
$$

Using Taylor series expansion about the root x (recall that $f(x) = 0$),

$$
\begin{aligned}
f(x_i) &= f(x + e_i) = f(x) + e_i f'(x) + e_i^2 \frac{f''(x)}{2} + \cdots , \\
f'(x_i) &= f'(x + e_i) = f'(x) + e_i f''(x) + \cdots ,
\end{aligned}
$$

with $e_i \ll 1$ it follows that

$$
\frac{f(x_i)}{f'(x_i)} = e_i + e_i^2 \frac{f''(x)}{2f'(x)} + \cdots .
$$

Substituting this last expression into the equation for e_{i+1} we get

$$
\begin{aligned}
e_{i+1} &\approx -e_i^2 \frac{f''(x)}{2f'(x)} \\
|e_{i+1}| &\approx \left| e_i^2 \frac{f''(x)}{2f'(x)} \right| = \text{Const} e_i^2.
\end{aligned}
$$

This equation says that NEWTON's method converges quadratically, i.e., near a root the number of significant digits approximately *doubles* with each step. However, one should be aware of the limitations of NEWTON's method:

1. Far from the root, where the higher order terms *are* important, NEWTON's formula can give inaccurate, meaningless corrections δ_i.

2. Nothing ensures that the iterative scheme will converge, except when we are sufficiently close to the root.

Hence, it is usually best to combine the bisection and NEWTON's method (see a Matlab implementation in Appendix 13.8).

Example 15 Convergence of Newton's Method. We solve the equation $f(x) = x^3 - 1 = 0$ (Figure 13.2) with two different initial guesses: $x_0 = 2$ (**Figure 13.6**) and $x_0 = -2$ (**Figure 13.7**).

Now suppose that we cover the square $[-1, 1] \times [-1, 1]$ with a fine grid of initial guesses for the complex roots of $z^3 - 1$. We then use NEWTON's method to obtain the root corresponding to each initial guess. The result, shown in **Figure 13.8**, is quite amazing. The basins of attraction of each root are infinitely nested into each other and form a *fractal Julia set.* The MATLAB code is listed in Appendix 13.9.

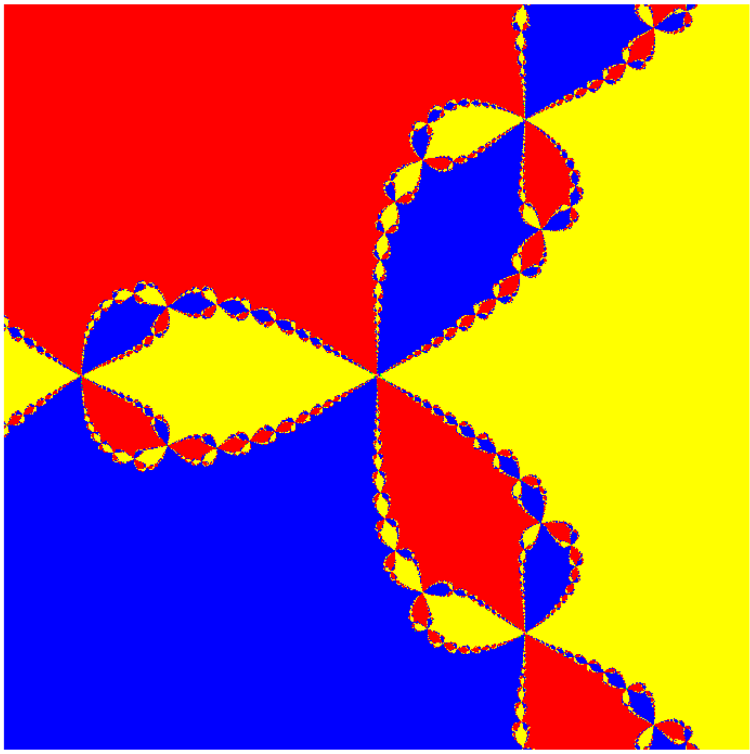


Figure 13.8: In Newton's method, the basins of attraction of the roots of $f(z) = z^3 - 1$ have very complicated fractal boundaries: any circle enclosing points of two colors must also enclose points of the third color.

13.6 MATLAB Code for the Bisection Method

```
function [xbis,niter] = rootbisection(fhandle,x1,x2,xtol)
%-------------------------------------------------------
% ROOTBISECTION: Root finding using bisection
% Input: fhandle = handle to a user-defined function
%        [x1,x2] = interval containing the root
%        xtol    = accuracy of the root (xbis)
% Output: xbis   = approximate root
%         niter  = number of iterations
%
% E77N Class Material, Numerical Recipes, 2002-03-07
%
% Note: the function handle is something like @myfun
% where myfun is a function defined as
% function [f] = myfun(x), ...
%-------------------------------------------------------
%
% Maximum number of iterations
MAXITER = 40;
%
% initialize
xbis = []; niter = 0;
if x1>x2
   x=x1; x1=x2; x2=x; % swap x1 and x2
end
fmid = feval(fhandle,x2);
f    = feval(fhandle,x1);
if (f*fmid > 0)
    s=sprintf('Root not bracketed in [x1=%g,x2=%g]',x1,x2);
    error(s)
end
%
% Orient the search so that f>0 lies at x+dx
if (f < 0)
    xbis = x1;
    dx   = x2-x1;
else
    xbis = x2;
    dx   = x1-x2;
end
%
% Bisection loop
for j = 1:MAXITER
    dx   = 0.5*dx;
% New xmid if f>0, xbis does not change
    xmid = xbis+dx;
    fmid = feval(fhandle,xmid);
    if (fmid <= 0)
        xbis = xmid; % new xbis if f<=0
    end
    if (abs(dx) < xtol | fmid == 0)
        niter = j;
        return
    end
end
%
disp('too many iterations in rootbisection')
return
```

13.7 MATLAB Code for Newton's Method

```
function [xnewt,niter] = rootnewton(fhandle,x1,x2,xtol)
%-----------------------------------------------------
% ROOTNEWTON: root finding using Newton's method
% Input:  fhandle = handle to a user-defined function
%         [x1,x2] = interval containing the root
%         xtol    = accuracy of the root (xnewt)
% Output: xnewt   = approximate root
%         niter   = number of iterations
%
% E77N Class Material, Numerical Recipes, 2002-03-07
%
% Note: the function handle is something like @myfun
% where myfun is a function defined as
% function [f,df] = myfun(x)
%-----------------------------------------------------
%
% Maximum number of iterations
MAXITER = 20;
%
% initialize
xnewt = 0.5*(x1+x2);
niter = 0;
%
% Newton-Raphson loop
for j = 1:MAXITER
    [f,df] = feval(fhandle,xnewt);
    dx = f/df;
    xnewt = xnewt - dx;
    if ((x1-xnewt)*(xnewt-x2) < 0)
        disp('rootnewton jumped out of brackets')
        niter = j;
        return
    end
    if (abs(dx) < xtol)
        niter = j;
        return
    end
end
%
disp('too many iterations in rootnewton')
return
```

13.8 MATLAB Code for Newton with Bisection

```
function [xsafe,niter] = rootsafe(fhandle,x1,x2,xtol)
%-----------------------------------------------------
% ROOTSAFE: root finding using a combination of
% bisection and Newton's method
% Input:  fhandle = handle to a user-defined function
%         [x1,x2] = interval containing the root
%         xtol    = accuracy of the root (xbis)
% Output: xsafe   = approximate root
%         niter   = number of iterations
%
% E77N Class Material, Numerical Recipes, 2002-03-07
%
% Note: the function handle is something like @myfun
% where myfun is a function defined as
% function [f,df] = myfun(x)
%-----------------------------------------------------
%
% Maximum number of iterations
MAXITER = 100;
%
% initialize
xsafe =[]; niter=1;
[fl,df] = feval(fhandle,x1);
[fh,df] = feval(fhandle,x2);
if ((fl>0 & fh>0) | (fl<0 & fh<0))
    disp('root is not bracketed in [x1,x2]')
    niter = 1;
    return
end
%
% Orient search so that f(x1)<0
if (fl == 0)
    xsafe = x1;
    niter = 1;
    return
elseif (fh == 0)
    xsafe = x2;
    niter = 1;
    return
elseif (fl < 0)
    xl = x1;
    xh = x2;
else
    xl = x2;
    xh = x1;
end
%
% Initialize guess for root, the "stepsize before last"
% and the last step
xsafe = 0.5*(x1+x2);
dxold = abs(x2-x1);
dx = dxold;
[f,df] = feval(fhandle,xsafe);
%
% Loop
for j = 1:MAXITER
    % Bisect if Newton out of range
    % or not decreasing fast enough
    if (((xsafe-xh)*df-f)*((xsafe-xl)*df-f) > 0 ...
        | abs(2.*f) > abs(dxold*df) )
        dxold = dx;
        dx = 0.5*(xh-xl);
        xsafe = xl+xh;
        if (xl == xsafe)  % change in root is negligible
            niter = j;
            return
        end
    else  % Newton step acceptable. Take it
        dxold = dx;
        dx = f/df;
        temp = xsafe;
        xsafe = xsafe - dx;
        if (temp == xsafe)
            niter = j;
            return
        end
    end
    if (abs(dx) < xtol)  % convergence criterion
        niter = j;
        return
    end
    % the one new function evaluation per iteration
    [f,df] = feval(fhandle,xsafe);
    if (f < 0)  % maintain the bracket on the root
        xl = xsafe;
    else
        xh = xsafe;
    end
end
%
disp('too many iterations in rootsafe')
return
```

13.9 MATLAB Code for Newton Method's Domains of Influence

```
function truedomains(m)
%
% Finds complex roots of f=z^3-1=0 for a fine grid of
% initial guesses, thus generating the beautiful Julia
% fractal for the domains of influence of each root
%
% E77N Class Material, T. W. Patzek, 2002-03-07
%
close all
niter = 20;    % Max number of Newton iterations
if nargin == 0
    m     = 1024; % Number of initial guesses is m*m
end
b     = 1;       % Limits of the initial guess grid
x     = linspace(-b,+b,m);
y     = linspace(-b,+b,m);
% Complex initial guesses
[X,Y] = meshgrid(x,y);
Z     = X +i*Y;
% Free memory
clear X Y
% Value of a reference root
z2      = -0.5+i*0.5*sqrt(3);
% Newton's method with f and f' substituted in
for k=1:niter;
   Z=2/3*Z + 1/3*1./(eps+Z.^2);
end
% Magnitude of difference
W = abs(Z-z2);
% Phase of each root
A = angle(Z);

SS = get(0, 'ScreenSize');
% Initialize a new figure
figure('Visible','on',...
       'Units','Pixels',...
       'Resize','on',...
       'Name','Newton''s method: Domains of influence of each root',...
       'Numbertitle','off',...
       'Position',[1,10,SS(3),SS(4)-85],...
       'BackingStore', 'off');
colors = prism(3*m); % The number of colors is ~3*m
colormap(colors);
hold on
% Color pixels in pseudocolor whose index increases with
% the magnitude of W-A
```

```
pcolor(W-A);
shading flat;
axis('square','equal','off');
hold off
%print -depsc E77N-Figure10.7.eps
```

Chapter 14

Iteration: Fractal Geometry of Mandelbrot's Set

14.1 What Are You Going To Learn?

You are going to learn more about iteration and its convergence properties. The quadratic function example I will use reveals the existence of fixed points, n-cycles, $n = 1, 2, \ldots$ and chaos, and will lead to the Mandelbrot set. I will follow closely some of the material presented on the Web by Prof. Robert L. Devaney[1] and in the book *Chaos Under Control* by David Peak and Michael Frame [26].

MandelbrotFractal.m generates the Mandelbrot fractal on an $m \times m$ grid, based on the code by Alberto Strumia.

MandelbrotMap.m calculates n terms in the Mandelbrot iteration, given a constant c.

MandelbrotOrbits.m plots the history of the Mandelbrot iteration $z = z^2 + c$, as a "time series," i.e., as $z(n)$ vs. n. Calculates the iteration amplitude histograms. You can use different constants c, some listed in the program, to obtain very different behaviors of the time series.

JuliaFractal.m plots colorful or b&w Julia fractals, some initial points are listed.

14.2 Why Is It Important?

The simple numerical experiments I will present here are a part of the worldwide interest in the *complex dynamic systems*. These experiments deal with chaos

[1] Department of Mathematics, Boston University, Boston, MA 02215, USA, http://math.bu.edu/DYSYS/FRACGEOM/FRACGEOM.html.

and order, and with their competition and coexistence. They show that the transition from one to the other has most magnificent geometric properties. As you will see soon, the emergence of chaos is a common feature of the simplest iterative schemes.

14.3 Iteration and Mandlebrot's Set

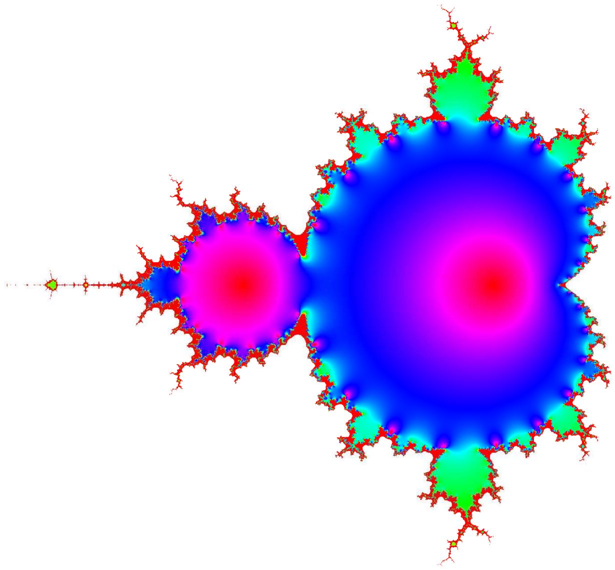


Figure 14.1: The MANDELBROT set obtained with the MATLAB function in Section 14.7.

One of the most intricate and beautiful images in all of mathematics is the MANDELBROT set, **Figure 14.1**, discovered by a French mathematician BENOIT MANDELBROT in 1980. At that time, MANDELBROT worked on new visualization techniques at the IBM Thomas J. Watson Research Center. The MANDELBROT set is considered the most complex object mathematics has ever encountered. It is a rather peculiar fractal in that it combines the properties of self-similarity with the properties of infinite change. Most people within the mathematics community, and many people outside of the discipline, have seen this image and have marvelled at its geometric intricacy. Unfortunately, only a few of these people are acquainted with the equally beautiful mathematics and numerics that lurk behind this image.

The MANDELBROT set is generated by iteration. Recall that to *iterate* means to repeat a process many times. In mathematics and computer science, this process is most often the applied to a mathematical function. For the MANDELBROT set, the function involved is the simplest nonlinear function imaginable, namely:

$$f(z) = z^2 + c, \tag{14.1}$$

where c is a constant. As we go along, we will specify exactly what values c takes. To iterate $z^2 + c$, we begin with a *seed* or the starting point for the iteration. This is a (real or complex) number, which we denote by z_0. Applying the function $z^2 + c$ to z_0 yields the new number

$$z_1 = f(z_0) = z_0^2 + c \tag{14.2}$$

Now, we iterate, using the result of the previous computation as the input for the next. That is

$$\begin{aligned} z_2 &= f(z_1) = z_1^2 + c \\ z_3 &= f(z_2) = z_2^2 + c \\ &\dots \\ z_{n-1} &= f(z_{n-2}) = z_{n-2}^2 + c \\ z_n &= f(z_{n-1}) = z_{n-1}^2 + c \end{aligned} \tag{14.3}$$

and so forth.

Definition 16 Orbits. The list of numbers $z_0, z_1, z_n, \dots$ generated under iteration of $z^2 + c$ is called the *orbit* of z_0. □

The principal questions in iterations are:

1. What is the fate of typical orbits?

2. Do they converge or diverge?

3. Do they cycle or behave erratically?

The MANDELBROT set is a geometric version of the answers to these questions. Let's begin with a few examples. Suppose we start with the constant $c = 1$. Then, if we choose the seed $z_0 = 0$, the orbit is

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0^2 + 1 = 1 \\ z_2 &= 2 \\ z_3 &= 5 \\ z_4 &= 26 \\ z_n &\to \infty \end{aligned} \tag{14.4}$$

and we see that this orbit tends to infinity. As another example, for $c = 0$, the orbit of the seed 0 is quite different: this orbit remains fixed for all iterations:

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0 \\ z_2 &= 0 \\ &\dots \\ z_n &= 0 \end{aligned} \tag{14.5}$$

If we now choose $c = -1$, something else happens. For the seed 0, the orbit is

$$\begin{aligned} z_0 &= 0 \\ z_1 &= -1 \\ z_2 &= 0 \\ &\dots \\ z_{2n-1} &= -1 \\ z_{2n} &= 0 \end{aligned} \tag{14.6}$$

Here we see that the orbit bounces back and forth between 0 and -1, a cycle of period 2.

Before proceeding, let us make a seemingly obvious and uninspiring observation. Under iteration of $z^2 + c$, either the orbit of 0 goes to infinity, or it does not. When the orbit does not go to infinity, it may behave in a variety of ways. It may be fixed or cyclic or chaotic, but the fundamental observation is that there is a dichotomy: sometimes the orbit goes to infinity, other times it does not.

The MANDELBROT set is a picture of precisely this dichotomy in the special case where 0 is used as the seed. Thus, the MANDELBROT set is a record of the fate of the orbit of $z_0 = 0$ under iteration of $z^2 + c$.

How then is the MANDELBROT set a two-dimensional or planar picture? The answer is, instead of considering real values of c, we also allow c to be a complex number. For example, the orbit of 0 under $z^2 + i$ is given by

$$\begin{aligned} z_0 &= 0 \\ z_1 &= i \\ z_2 &= i^2 + i = -1 + i \\ z_3 &= (-1 + i)^2 + i = 1 - 2i - 1 + i = -i \\ z_4 &= -1 + i \\ z_5 &= -i \\ &\dots \end{aligned} \tag{14.7}$$

and we see that this orbit eventually cycles with period 2. If we change c to $2i$, then the orbit behaves very differently

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 2i \\ z_2 &= -4 + 2i \\ z_3 &= 12 - 14i \\ z_4 &= 52 - 334i \\ z_5 &= \text{BIG (meaning far from the origin)} \\ z_6 &= \text{BIGGER} \end{aligned} \tag{14.8}$$

and we see that this orbit tends to infinity in the complex plane (the numbers comprising the orbit move further and further from the origin). Again, we make the fundamental observation either orbit of 0 under $z^2 + c$ tends to infinity, or it does not.

14.4 The Mandelbrot Set

The MANDELBROT set puts some geometry into the fundamental observation above. The precise definition is:

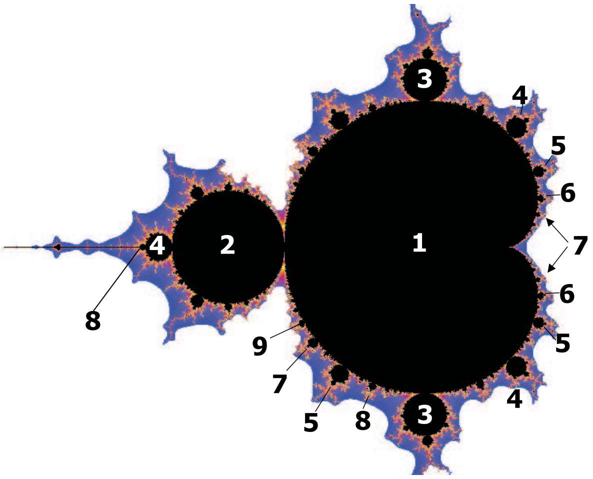


Figure 14.2: Each arrow labelled "n" points to an n-cycle region in the MANDELBROT set.

Definition 17 The Mandelbrot set M consists of all of those complex values of c for which the corresponding orbits of $z_0 = 0$ do not escape to infinity under the iteration $z^2 + c$. □

The Mandelbrot set is made up of infinitely many discs and cardioids (most are slightly distorted from true discs and cardioids), but the arrangement and sizes of these pieces exhibit many patterns.

Definition 18 Given a fixed (complex) value of c, the filled-in Julia set J consists of all of those (complex) z_0-values for which the corresponding orbit under $z^2 + c$ does not escape to infinity. The Julia set is the edge or boundary of the filled-in Julia set. □

Referring to[2] **Figure 14.2**, we see that attached to the left of the big cardioid (the fixed point, or 1-cycle, component) is a 2-cycle component, attached to the

[2] Figure 14.2 follows Figure 7.13 in [26].

left of the latter is a 4-cycle component, attached to the left of the latter is an 8-cycle component, and so on. This is called the period-doubling cascade. In the 1970s this cascade was studied (for a process equivalent to looking at the Mandelbrot set for real numbers) by MITCHELL FEIGENBAUM [11], and PIERRE COULETTE and CHARLES TRESSER [6]. Certainly, the $2n$-cycle components get smaller as n increases.

Call c_1 the point of attachment of the 2-cycle and 1-cycle components. Call c_2 the point of attachment of the 4-cycle and 2-cycle components. Call c_3 the point of attachment of the 8-cycle and 4-cycle components. In general, call c_n the point of attachment of the 2^n-cycle and 2^{n-1}-cycle components in the period-doubling cascade. Careful numerical experiments [26] give these values:

$$\begin{aligned}
c_1 &= -0.75\\
c_2 &= -1.25\\
c_3 &= -1.3680989394\ldots\\
c_4 &= -1.3940461566\ldots\\
c_5 &= -1.3996312389\ldots\\
c_6 &= -1.4008287424\ldots\\
c_7 &= -1.4010852713\ldots\\
c_8 &= -1.401140214699\ldots\\
c_9 &= -1.401151982029\ldots\\
c_{10} &= -1.401154502237\ldots
\end{aligned} \tag{14.9}$$

To understand the fate of orbits, it is most often easiest to proceed geometrically. Accordingly, a "time" series ($f(z)$ vs. n) plot of the orbit often gives more information about the fate of the orbits. In the plots below, we have displayed the time series for z^2+c where $c = -0.6$, -0.75, -1.3, -1.38, and -1.9. In each case, we have computed the orbit of 0. Note that the fate of the orbit changes with c. For $c = -0.6 < c_1$ there is no oscillation and the orbit tends to a fixed point, **Figure 14.3**. For $c = -0.75$, we see that the orbit approaches a 2-cycle, **Figure 14.4**, as required by Eq. $(14.9)_1$. In the histogram, you can see that z oscillates between two values (hence the name "2-cycle"). For $c = -1.3 > c_2$, the orbit tends to a 4-cycle, **Figure 14.5**. For $c = -1.38 > c_3$, we see an 8-cycle, **Figure 14.6**. When $c = -1.9 > c_{10}$, there is no apparent pattern for the orbit; mathematicians use the word *chaos* for this phenomenon, **Figure 14.7**.

From our previous calculations, it follows that $c = -0.6, -1.1, -1.3, -1.38$, -1.9, and i all lie in the MANDELBROT set, whereas $c = 1$ and $c = 2i$ do not. At this point, a natural question is: Why would anyone care about the fate of the orbit of 0 under $z^2 + c$? Why not the orbit of $z_0 = i$? Or $z_0 = 2 + 3i$, or any other complex seeds, for that matter?

There is a very good reason for inquiring about the fate of the orbit of 0; somehow the orbit of 0 tells us a tremendous amount about the fate of *all* other orbits under $z^2 + c$. Before turning to this idea, note that the very definition of the MANDELBROT set gives us an algorithm for computing it. We simply consider a square in the complex plane (usually centered at the origin with sides of length 4). We overlay a grid of equally spaced points in this square. Each of these points is to be considered a complex c-value. Then, for each such c, we ask the computer to check whether the corresponding orbit of 0 goes to infinity (escapes) or does not go to infinity (remains bounded). In the former case, we leave the corresponding c-value (pixel) white. In the latter case, we paint the c-value black (or in a pseudocolor). Thus, the color points in **Figure 14.1** represent the MANDELBROT set.

Two points need to be made. Figure 14.1 is only an approximation of the MANDELBROT set. Indeed, it is not possible to determine whether certain c-values lie in the MANDELBROT set. We can only iterate a finite number of times to determine if a point lies in M. Certain c-values close to the boundary of M have orbits that escape only after a very large number of iterations. A second question is: How do we know that the orbit of 0 under $z^2 + c$ really does escape

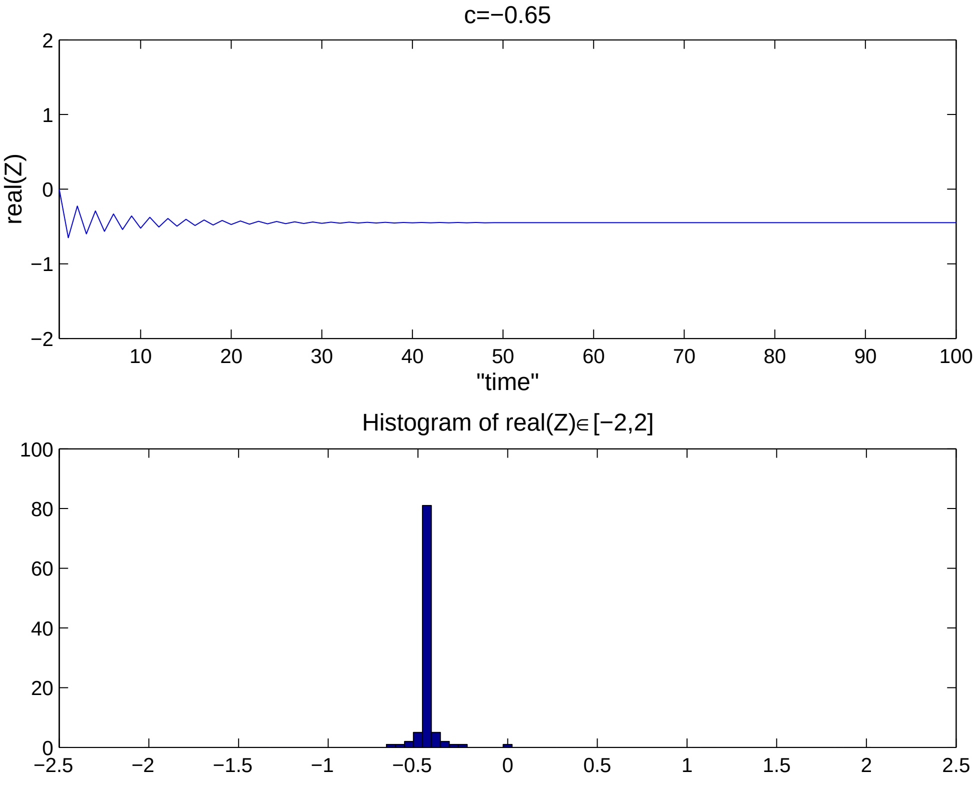


Figure 14.3: “Time” series plot of the real part of orbit of $z_0 = 0$ and the corresponding histogram for $c = -0.65$. The orbit settles on a fixed point at $z = 0.4$.

to infinity? Fortunately, there is an easy criterion, which helps:

Definition 19 The Escape Criterion: Suppose $|c|$ is less than or equal to 2. If the orbit of 0 under $z^2 + c$ ever lands outside of the circle of radius 2 centered at the origin, then this orbit definitely tends to infinity. For a proof see **Appendix 14.5**. □

It may seem that this criterion is not too valuable, as it only works when $|c|$ is less than or equal to 2. However, it is known that the entire Mandelbrot set lies inside this disk, so these are the only c-values we need consider anyway.

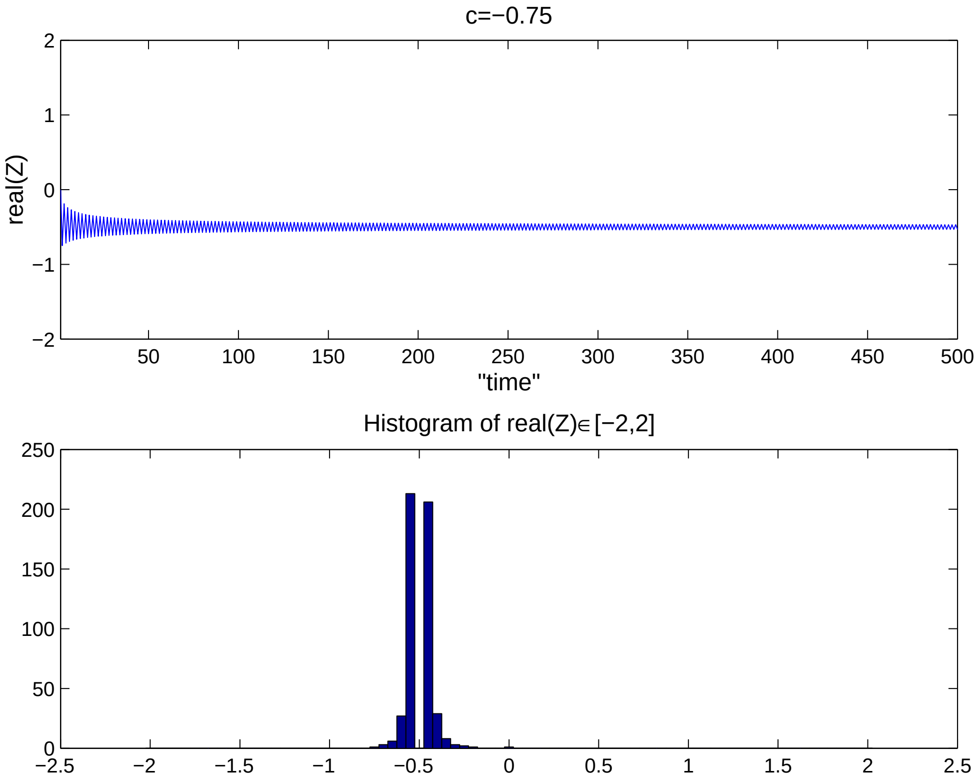


Figure 14.4: "Time" series plot of the real part of orbit of $z_0 = 0$ and the corresponding histogram for $c = -0.75$. The orbit eventually settles to an oscillation between -0.55 and -0.45, or a "2-cycle".

14.5 Why $z > 2$ Runs Away to Infinity?

For a complex number $z_n = x_n + iy_n$, the absolute value is $|z_n| = \sqrt{x_n^2 + y_n^2}$, the distance from z_n to the origin.

Recall that the sequence $z_0, z_1, \ldots$ is generated by iteration $z_{n+1} = z_n^2 + c$. We want to show that if some z_n satisfies $|z_n| > \max(2, |c|)$, then the sequence z_n, z_{n+1}, $\ldots$ runs away to infinity.

Because $|z_n| > 2$, we can write $|z_n| = 2 + e$, for some $e > 0$. From the triangle inequality:

$$|z_n^2| = |z_n^2 + c - c| \leq |z_n^2 + c| + |c| \tag{14.10}$$

Because $|z_n| > |c|$

$$\begin{aligned} |z_n^2 + c| \geq |z_n^2| - |c| = |z_n|^2 - |c| > |z_n|^2 - |z_n| \\ = (|z_n| - 1)|z_n| = (1 + e)|z_n| \end{aligned} \tag{14.11}$$

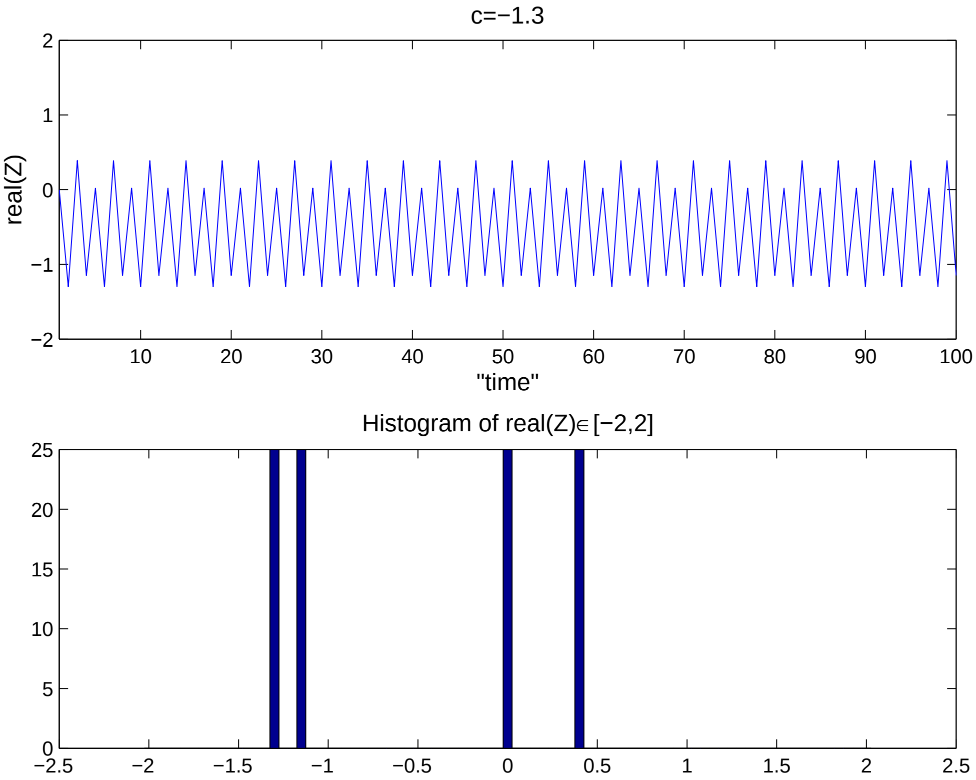


Figure 14.5: "Time" series plot of the real part of orbit of $z_0 = 0$ and the corresponding histogram for $c = -1.3$. The orbit eventually settles to a modulated oscillation between roughly -1.3 and 0.4, and -1.1 and 0 or a "4-cycle".

That is,

$$|z_{n+1}| > (1+e)|z_n|. \tag{14.12}$$

Iterating,

$$|z_{n+k}| > (1+e)^k|z_n|. \tag{14.13}$$

To complete the proof that $|z_n| > 2$ implies the sequence runs away to infinity, observe that if $|c| > 2$, then

$$\begin{aligned} z_0 &= 0 \\ z_1 &= c \\ z_2 &= c^2 + c = c(c+1). \end{aligned} \tag{14.14}$$

Therefore, $|z^2| = |c||c+1| > |c|$ (noting $|c+1| > 1$ because $|c| > 2$).

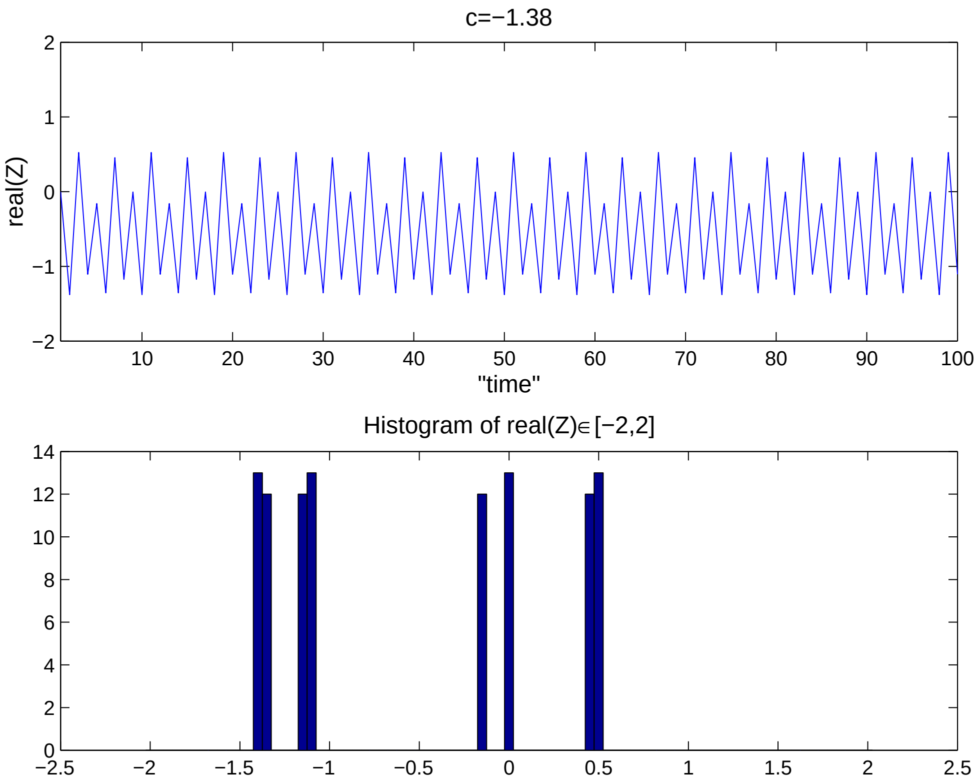


Figure 14.6: "Time" series plot of the real part of orbit of $z_0 = 0$ and the corresponding histogram for $c = -1.38$. The orbit eventually settles to a modulated oscillation with 4 different amplitudes or an "8-cycle".

14.6 MATLAB Code for the MandelBrot Iteration

```
function MandelbrotOrbits(c,n)
%---------------------------------------------------------------------------
%
% MANDELBROTORBITS plots results of Mandelbrot iteration
% as time series, z(n) vs (n).  Calculates amplitude histograms
%
% INPUTS:
%               c = iteration constant,
%               n = number of points in time series
%
% Here are the points of attachment of the k-cycles to
% (k-1)-cycles.  Use them as your constants c to obtain some
% interesting results.
%
```

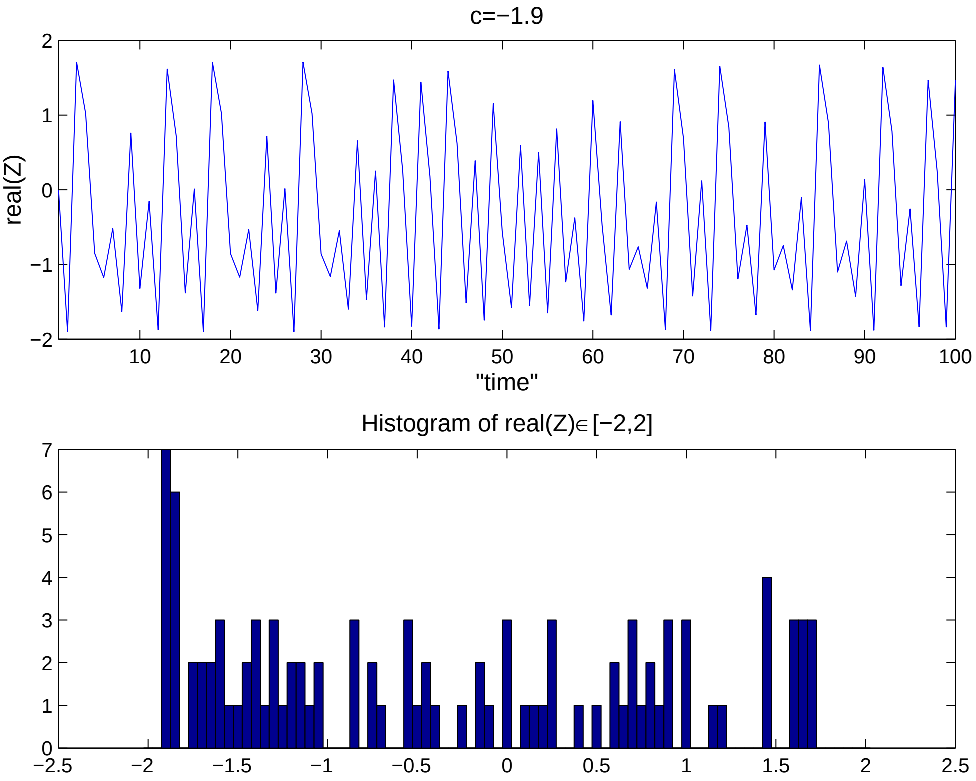


Figure 14.7: "Time" series plot of the real part of orbit of $z_0 = 0$ and the corresponding histogram for $c = -1.9$. The orbit becomes chaotic.

```
% E77N Course Material, TWP, 2002-03-11
% Last modified, TWP, 2002-10-07
%--------------------------------------------------------------------------
%  c1  = -0.75
%  c2  = -1.25
%  c3  = -1.3680989394
%  c4  = -1.3940461566
%  c5  = -1.3996312389
%  c6  = -1.4008287424
%  c7  = -1.4010852713
%  c8  = 1.401140214699
%  c9  = -1.401151982029
%  c10 = -1.401154502237
%--------------------------------------------------------------------------

if nargin<2
    n=100;
end

PrintFlag    = 0;
PlotFlag     = 0;
FigureNumber = 1;
Ns           = 1:1:n;
% Calculate the time series
Zs = MandelbrotMap(c,n);

% Get the current screen size and set the figure size
% so that it does not cover the bottom screen bar
SS = get(0, 'ScreenSize');

% Initialize a new figure
figure(FigureNumber)
set(gcf,'Visible','on',...
       'Units','Pixels',...
       'Resize','on',...
       'Name','Mandelbrot Orbits',...
       'Numbertitle','off',...
       'Position',[1,1,SS(3),SS(4)-50],...
       'BackingStore', 'off');


% Specify plot area
%                                     xmin   ymin   dx    dy
axes('Units','Normalized','Position',[0.28  0.15  0.55  0.80])
set(gca, 'FontSize', 14)
if imag(c)==0
   s=sprintf('c=%.14f',c);
else
   s=sprintf('c=%g+i%g',real(c),imag(c));
end
```

```
index=1;
NumberOfSubplots=3;
if (imag(c)==0)
    NumberOfSubplots=2;
end
subplot(NumberOfSubplots,1,index)
index=index+1;
plot(Ns,real(Zs))
axis([1,n,-2,2]);
title(s, 'fontsize',12)
ylabel('real(Z)', 'fontsize',12)
xlabel('"time"', 'fontsize',12)

if (imag(c)~=0)
    subplot(NumberOfSubplots,1,index)
    index=index+1;
    plot(Ns,imag(Zs))
    axis([1,n,-2,2]);
    ylabel('imag(Z)','fontsize',12)
end
x=-2:min(50/n,0.05):2;
subplot(NumberOfSubplots,1,index)
index=index+1;
hist(real(Zs),x)
title('Histogram of real(Z)\in[-2,2]', 'fontsize',12)

if PrintFlag
    fn = sprintf('Figure11.%s.eps',s);
    print( gcf, '-depsc2', fn );
end
% Plot a Poincare return map (ommitted by default)
if (PlotFlag)
    figure(2)
    offset=1;
    z2 = real(Zs(1+offset:end));
    plot(real(Zs(1:end-offset)), z2)
    s= sprintf('real[Z(n-%d)]',offset);
    xlabel(s)
    ylabel('real[Z(n)]', 'fontsize',12)
    title('Return map of real(Z)', 'fontsize',12)
end

if PrintFlag
    fn = sprintf('Figure2-%g.eps',c);
    print( gcf, '-depsc2', fn );
end

function Zs = MandelbrotMap(c,n)
% Mandelbrot iteration, n-times
% E77N Course Material, TWP, 2002-03-11
```

```
%
z0   = complex(0,0);
Zs   = complex(zeros(n,1), zeros(n,1));
Zs(1)= z0;
%
for i = 2:n
    Zs(i)=Zs(i-1)*Zs(i-1)+c;
end
% Q: Can you vectorize this function?
```

14.7 MATLAB Code for the MandelBrot Set

```
function MandelbrotFractal(m, noiter)
%------------------------------------------------
% Adapted by TWP from Alberto Strumia
% University of Bolognia
% http://eulero.ing.unibo.it/~strumia/
%
% Input arguments: m      = number of points in x
%                                       and y directions.
%                          noiter = number of iterations
%
% An m-by-m grid of constants C
% will be used to find the orbits of zero
% I have used m=2000 to generate the pictures
% but it may be too much for your computer
%
% The iteration seed is an m-by-m zero array
% The Mandelbrot set is the locus of finite
% orbits of zero.
%
% $Version 1.1, TWP, 2002.02.07
%
%------------------------------------------------
%
if nargin<1
   m =500;
   noiter=25;
elseif nargin <2
   noiter=25;
end

ColorFlag=1;%<<<Set to 1 if color is desired

%--------------------------------------------------------
% Span x=[-2.1, 0.5], y=[-1.3,1.3] with a grid of
% m-by-m points
```

```
x=linspace(-2.1,0.5,m);
y=linspace(-1.3,1.3,m);

%----------------------------------------------------------
% [X,Y] = meshgrid(x,y) transforms the domain
% specified by vectors x and y into arrays X and Y,
% which can be used to evaluate functions of two variables
% and three-dimensional mesh/surface plots.
% Rows of the output array X are copies of vector x.
% Columns of the output array Y are copies of vector y.

[X,Y] = meshgrid(x,y);

% Array of complex values of C
C = X + i*Y;

% Free precious memory from X and Y
clear X Y

%----------------------------------------------------------
% A variant of the classical escape method is applied here:
% instead of evaluating the number of cycles required
% in order that the modulus of the partial sum exceeds
% a fixed bound, we evaluate the value of that modulus
% for a fixed number of cycles.
%
% Instead Z=zeros(m), we’ll save one iteration by setting

Z = C; % Because Zo = 0, Z1=C!

%----------------------------------------------------------
% MATLAB shines here in performing matrix operations
for k=1:noiter
    Z = Z.*Z + C;
end
% I do not need C anymore
clear C
close all
figure(1)
if (ColorFlag)
   %
   % Attenuate the huge absolute values of abs(Z)
   % for the diverging orbits
   % This avoids an expensive vector if statement.
   % For complex Z, abs(Z)= sqrt(X^2+Y^2)
   %
   % I do not need the old Z anymore
   Z = exp(-abs(Z));
   % Set a color map of your choice: colormap prism(256);
   % or colormap copper(256); etc.
   colormap hsv(1024);

   % A pseudocolor plot is a rectangular array of cells
   % with colors determined by the array Z.
   % MATLAB creates a pseudocolor plot by using each set
   % of four adjacent points in C to define a surface patch
   % (i.e., cell).

   pcolor(Z);

   % No color interpolation
   shading flat;
%----------------------------------------------------------
% Print to an eps file.  The bitmap from Copy Figure
% may take too much memory and crash MATLAB
% This prints a huge eps file to the current directory
   %fn = sprintf('Figure1Color.eps');
else
   W = ones(m);
   W = W + ( abs(Z) < 2 ); % black or white?
   % Because images are displayed down from top of screen,
   % not that it matters here
   W = flipud(W);

   % Ones for white exterior; zeros for black interior
   bwmap = [ 1 1 1; 0 0 0 ];
   image(W), colormap(bwmap);
   %fn = sprintf('Figure1B&W.eps');
end

% Set both axes to form a square and turn them off
axis('square','equal','off');
```

Chapter 15

Recursion, Part II∗

15.1 What Are You Going To Learn?

A sloppy recursion algorithm will prevent you from using it in practice because of stack overflow. As we saw in Lecture 8, a naïve FIBONACCI recursion resulted in roughly 2^{n-1} calls to calculate the nth FIBONACCI number. We overcame our naïveté by introducing a helper function that "memorized" the previous FIBONACCI numbers, and limited the number of recursive calls to n. In this chapter, we establish the connection between memory and more complicated recursion, and develop new smart, first- and second-order linear recursion and iteration algorithms. The one-step feedback machine with one and two variables is introduced to illustrate the concept of single-step and two-step iteration and recursion. The feedback machines are described in detail in a remarkable book by PEITGEN, JÜRGENS and SAUPE [27].

In Chapter 15 folder on fttp:\\petroleum.berkeley.edu, look up the following MATLAB files:

IterSqrt.m An iterative calculation of the square root of x with an initial guess x_0. Returns the root approximation, x_n, and the number of iterations, n, necessary to obtain the root with machine precision.

RecSqrt.m A recursive calculation of the square root of x with an initial guess x_0. Returns the root with machine precision.

IterFib.m An iterative calculation of the FIBONACCI numbers based on a two-variable, single-step feedback machine.

SORDemo.m Demo for Second Order Recurrence, `modSOR`, written with a helper function, `helpSOR`, which "memorizes" two previous values.

modSOR.m This SOR function calculates recursively the current response by using a helper function to store two previous responses.

helpSOR.m The helper function for `modSOR` calculates recursively the current response by swapping the second and third argument, so that the current response becomes the previous one in the next call.

15.2 Why is it important?

Recursion and iteration are two different ways of looking at feedback machines, very useful mathematical tools developed to solve complex problems in mathematics and engineering. Simple iteration and recursion correspond to a one-step feedback machine operating with a single number input and outputting a single number. Complicated iteration and recursion correspond to one-step feedback machines that operate on vectors of input numbers and output vectors of numbers. Your understanding of the connection between iteration and recursion will help you to become the cool computer-literate engineers and scientists.

15.3 One-Step Feedback Machine: First Order Recursion

One-step machines are characterized by an iteration formula $x_{n+1} = f(x_n)$, where $f(x)$ can be any function of x, **Figure 15.1**. It requires one number as input and returns one number - the result of the formula as output. The formula can be controlled by a fixed parameter (as in the Julia sets), e.g., $f(x_n) = x_n^2 + c$, i.e., with control parameter c. The numbers are indexed to keep track of the time (cycle) in which they were obtained. Recall the "time" series and the onset of chaos in the discussion of the Mandelbrot set.

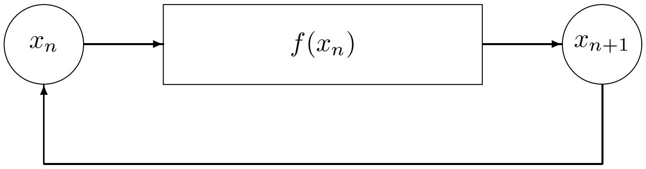


Figure 15.1: One-step feedback machine is a very useful tool developed for the solution of complex mathematical problems. One-step feedback machines have a long tradition in mathematics.

Example 16 Ancient Square Root Computation. This example of a one-step feedback machine goes back to the Sumerian mathematicians some 4000 year ago. It is a beautiful illustration of the strength and continuity of mathematics.

Pick a real number $a > 0$. Compute a sequence $x_1, x_2, x_3, \ldots$ such that the limit is $\sqrt{a}$, i.e., x_n approaches $\sqrt{a}$ as we increase n. We begin with an arbitrary guess $x_0 > 0$ and proceed with

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{15.1}$$

Let us approximate, for example, $\sqrt{2} \approx 1.41421356237310$. Let us start with $x_0 = 2$.

$$\begin{aligned}
x_1 &= \frac{1}{2}\left(x_0 + \frac{a}{x_0}\right) = \frac{1}{2}\left(2 + \frac{2}{2}\right) = 1.5 \\
x_2 &= \frac{1}{2}\left(x_1 + \frac{a}{x_1}\right) = \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.41666666666667 \\
x_3 &= \frac{1}{2}\left(x_2 + \frac{a}{x_2}\right) = \frac{1}{2}\left(1.416... + \frac{2}{1.416...}\right) = 1.41421568627451 \\
&\ldots
\end{aligned} \tag{15.2}$$

With our starting point, this algorithm converges to machine precision in five iterations. Here is the iterative algorithm:

```
function [xn,n]=IterSqrt(x,x0)
% Iterative square root of x with an initial guess x0
% Returns the root approximation, xn, and the number of
% iterations, n, to obtain the root with machine precision
% E77N Class Material, T.W. Patzek,  2002-03-10
%
error = eps; %eps = machine precision
n     = 0;
xn    = x0;
while error > eps
    xnp1  = 0.5*(xn+x/xn)
    error = abs(xnp1-xn);
    xn    = xnp1;
    n     = n+1;
end
n = n-1;
```

Here is the recursive version of the algorithm:

```
function root = RecSqrt(x,x0)
% Recursive square root of x with an initial guess x0.
% Returns the root with machine precision
% E77N Class Material, T.W. Patzek, 2002-03-10
%
root = 0.5*(x0+x/x0);
if abs(root-x0) > eps
   root = RecSqrt(x,root);
end
```

Which one do you prefer? In this simple case, both algorithms are similar in length and complexity, although the recursive algorithm is always more compact.

These two algorithms for the computation of the square root are an example of a more general method for the solution of nonlinear equations, today called NEWTON's method. Indeed, put $f(x) = x^2 - a = 0$, $f'(x) = 2x$, and

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}x_n + \frac{a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \qquad (15.3)$$

Thus, NEWTON's method has roots (no pun intended) that are 4000 years old!□

15.4 One-Step Feedback Machine with Two Variables: Second Order Linear Recursion

Suppose now that the feedback machine can remember the recent past; for example, the last iteration.

15.4.1 Feedback Machines with Memory

Superficially it seems that the two-step processes cannot be fit into the feedback machine in Figure 15.1. Indeed the output, x_{n+1}, depends not only on the last step, x_n, but also on the previous step, x_{n-1}. Therefore, our feedback machine must now be endowed with *memory* of its recent past.

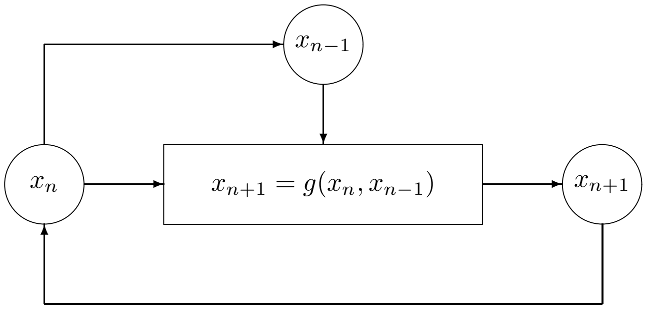


Figure 15.2: Feedback machine with memory. Here x_n is the input unit, and x_{n-1} is the memory unit. These two units must be initialized with x_1 and x_0, respectively. At each time level the memory unit is updated, $x_{n-1} \leftarrow x_n$, before the input unit, $x_n \leftarrow x_{n+1}$. This feedback machine is equivalent to the one-step feedback machine with two variables, shown in Figure 15.3.

We now extend the concept of a simple feedback machine by equipping the processing unit with internal memory. Here the iteration of the two-step method can be implemented as follows. To start the machine we need two values, x_0 and x_1, **Figure 15.2**, and then we follow with iteration:

Preparation: Initialize the memory unit with x_0 and the input unit with x_1.

Iteration: Evaluate $x_{n+1} = g(x_n, x_{n-1})$, where x_n is in the input unit and x_{n-1} is in the memory unit. Then update the memory unit with x_n and the input unit with x_{n+1}.

15.4.2 One-Step Machines with Two Variables

The feedback machine with memory seems to be a generalization of the one-step feedback machine. This is not the case at all. If we analyze carefully the feedback machine in Figure 15.2, we see that it is equivalent to a one-step machine in which a pair of input variables (x_n, x_{n-1}) generates a pair of output variables (x_{n+1}, x_n) . Formally, we introduce a new variable $y_n = x_{n-1}$, and extend the formula $x_{n+1} = g(x_n, x_{n-1})$ to the equivalent pair:

$$\begin{aligned} x_{n+1} &= g(x_n, y_n) \\ y_{n+1} &= x_n \end{aligned} \tag{15.4}$$

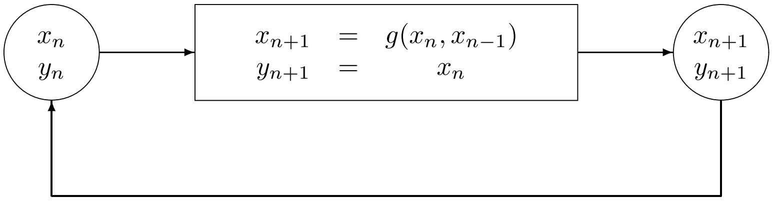


Figure 15.3: One-step feedback machine with two variables.

We conclude that one-step feedback processes can be extended to any number of iteration steps, if their variables are represented by vectors, **Figure 15.3**.

The now familiar FIBONACCI numbers are such a sequence if we put

$$g(x_n, x_{n-1}) = x_n + x_{n-1} \tag{15.5}$$

Let us now analyze carefully the problem of rabbit population dynamics in Lecture 10 and recast it in terms of a feedback machine algorithm. In our population of rabbits, we must distinguish between adult and young rabbits, **Figure 15.4**. A just-born pair is young, and turns adult after one time step

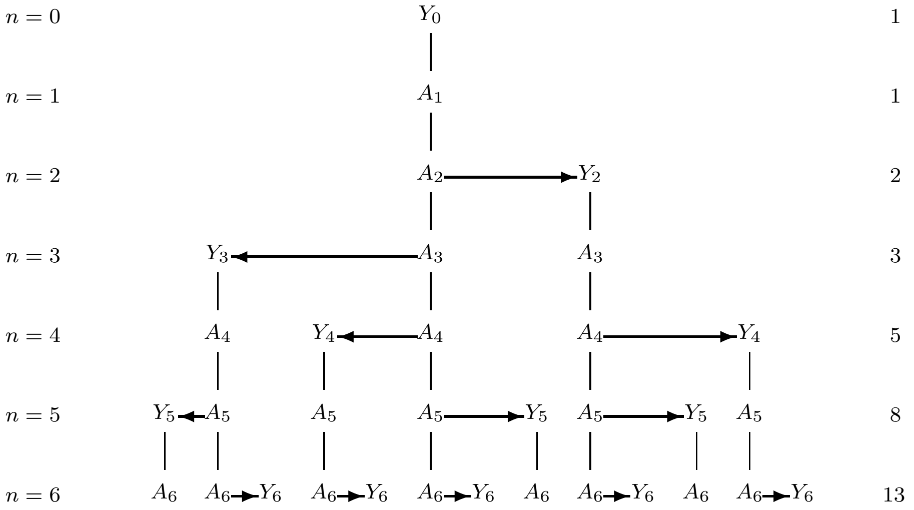


Figure 15.4: The ancestral tree of the FIBONACCI rabbit family. KEY: Y_n = baby rabbits at time level n, A_n = adult rabbits at time level n. At each time level, the adult population consists of young adults and breeding adults. Note that the current notation, which refers to time levels, is different from that in Lecture 7.

(one month). Moreover, an adult pair gives birth to a young pair after one time step. Let Y_n and A_n be the populations of young and adult pairs after n months, respectively. Initially at time $n = 0$, there is one young pair and no adults ($Y_0 = 1$ and $A_0 = 0$). After one month, the young pair turns into an adult one ($Y_1 = 0$ and $A_1 = 1$). After two months, the adult pair gives birth to one young pair ($Y_2 = 1$, $A_2 = 1$), then again after one month. In addition, the young pair turns into an adult one ($Y_3 = 1$, $A_3 = 2$).

The general rule is that the number of newborn pairs, Y_{n+1}, equals the previous adult pair, A_n. The adult population grows by the number of young pairs, Y_n, from the previous month. Thus, the following two equations completely describe the dynamics of rabbit population:

$$\begin{aligned} Y_{n+1} &= A_n \\ A_{n+1} &= A_n + Y_n \end{aligned} \tag{15.6}$$

As initial values, we take $Y_0 = 1$ and $A_0 = 0$. From the Eq. $(15.6)_1$, it follows that $Y_n = A_{n-1}$. Inserting this into the second equation, we obtain

$$\begin{aligned} A_{n+1} &= A_n + A_{n-1} \\ A_0 &= 0, \quad A_1 = 1 \end{aligned} \tag{15.7}$$

This is the single equation for the total adult rabbit population (see Eq. $(15.6)_2$) at time n. Using this equation the number of pairs in successive generations is easily computed:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233, \ldots \tag{15.8}$$

The FIBONACCI numbers are a special case of a general recurrence called second-order linear recurrence or SOR. The FIBONACCI sequence is roughly exponential. Let's confirm that by plotting `modfib(n)` vs. n with a linear x-axis and a logarithmic y-axis. The MATLAB plotting function is called `semilogy`, **Figure 15.5**.

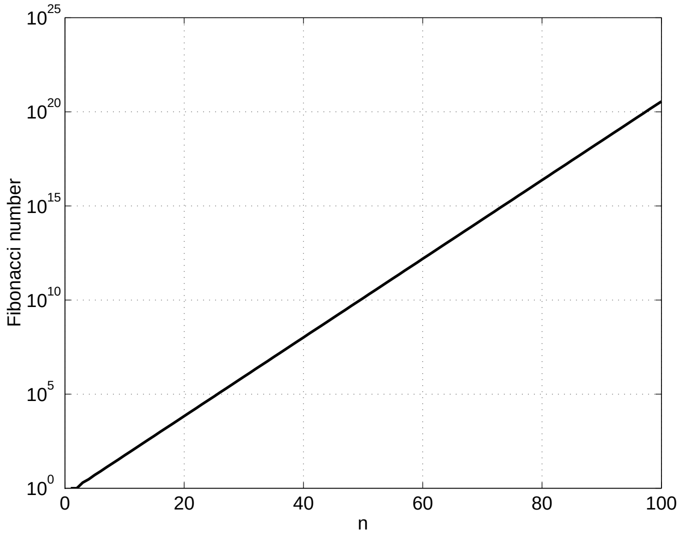


Figure 15.5: The FIBONACCI numbers grow exponentially as this semi-logarithmic plot demonstrates.

The exponent of the line slope in Figure 15.5 is

$$1.618 \approx \frac{1+\sqrt{5}}{2}, \tag{15.9}$$

which is better known as the "golden mean" or *proportio divina*[1]. Therefore, the FIBONACCI numbers grow as

$$A_n \sim \exp\left(\frac{1+\sqrt{5}}{2} n\right) \tag{15.10}$$

[1] In architecture and art this number is obtained by dividing a given length of $1+x$ into two parts, the larger, x, and the smaller, 1, and requesting that the whole to the larger part is in the same proportion as the larger part to the smaller part: $(1+x)/x = x/1$. The golden ratio is the larger root of the resulting quadratic equation, $x^2 - x - 1 = 0$.

With the FIBONACCI numbers, we already know that

$$\begin{aligned} x_{n+1} &= x_n + y_n \\ y_{n+1} &= x_n \\ x_0 = 1, &\quad y_0 = 0 \end{aligned} \tag{15.11}$$

So here is the iterative FIBONACCI algorithm:

```
function xnp1=IterFib(n)
% Iterative calculation of Fibonacci numbers
% based on a two-variable single-step feedback machine
% E77N Class Material, T.W. Patzek, 2002-03-10
%
xn = 1; yn = 0; xnp1 = 0;
if (n==2) xnp1 = 1; end
for i=3:n
    xnp1 = xn + yn;
    ynp1 = xn;
    xn   = xnp1;
    yn   = ynp1;
end
```

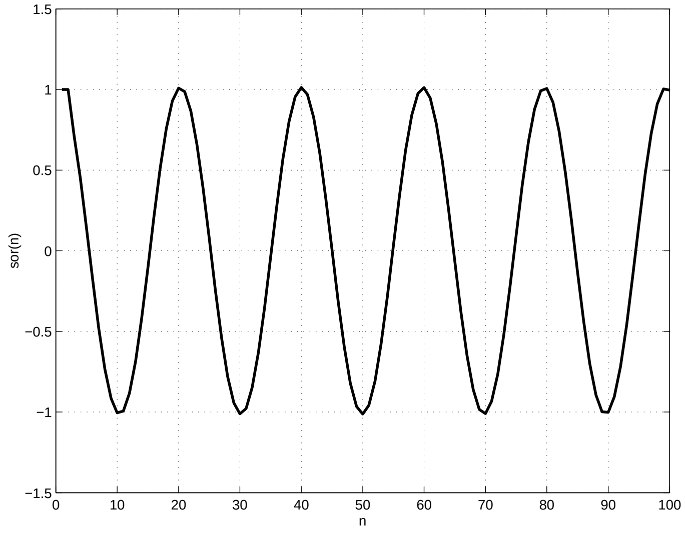


Figure 15.6: Second order linear recurrence with freq = 0.05 and gain = 1, results in the plot of $\cos(2\pi\nu_1 n)$, where $\nu_1 = 1/20$.

This algorithm is about five times faster than the modified recursive algorithm of LECTURE 11, `modFib`.

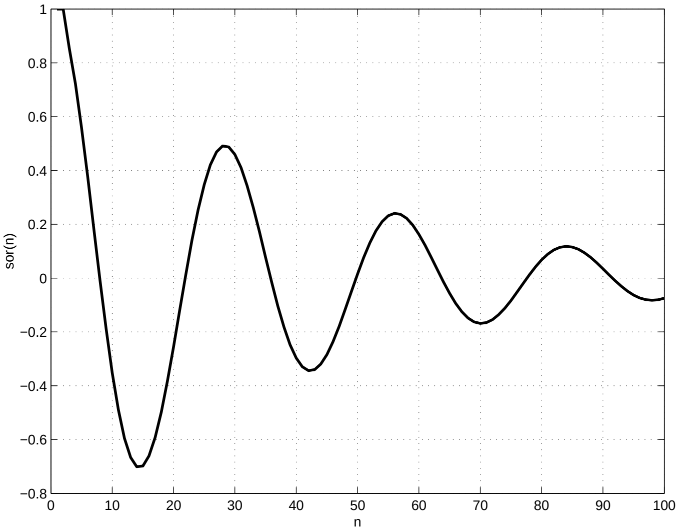


Figure 15.7: Second order linear recurrence with freq = 0.05 and gain = 0.95, results in the plot of exponentially damped $\cos(2\pi\nu_2 n)$, where $\nu_2 \neq \nu_1$.

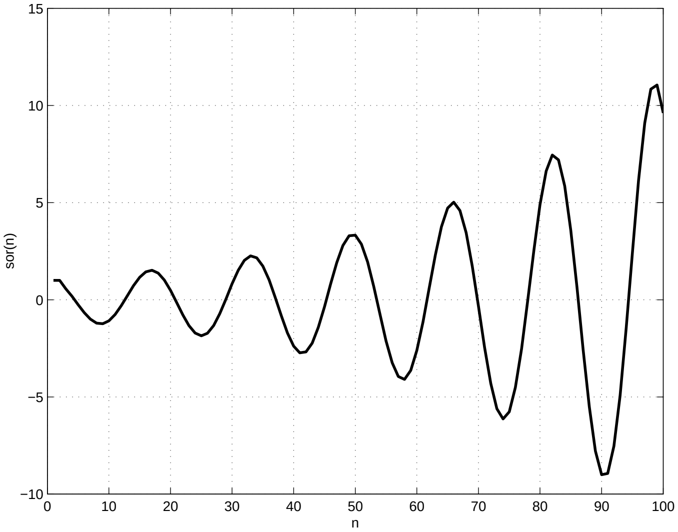


Figure 15.8: Second order linear recurrence with freq = 0.05 and gain = 1.05, results in the plot of exponentially growing $\cos(2\pi\nu_3 n)$, where $\nu_3 \neq \nu_2$.

Second-order recurrences give sequences that are sums of two exponentials (if they are different, the smaller once is swamped by the bigger one, as in the case of Fibonacci numbers). If we change the recurrence a little bit, we get sums of two complex exponentials, which is very interesting from an engineering point of view. Let's call the function to do this `modSOR` for modified Second-Order Recurrence. The complex amplitude and frequency are governed by two parameters: `freq` and `gain`.

To accommodate the complex exponentials, we alter the modified FIBONACCI algorithm in Chapter 11. Our new smart algorithm requires only n calls, and it can be used directly to plot results with up to 495 points (remember the stack limit is 500 words).

```
function retval = modSOR(n,freq,gain)
%---------------------------------------------------------------
% Input arguments:
%       n    = current argument
%       freq = frequency of response
%       gain = gain of amplitude of response
%
%       The SOR function calculates recursively the current
%       response by using a helper function to store the
%       previous responses
%
% Written by T. W. Patzek, March 27, 2001
% UC Berkeley, 437 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834 or 510-486-5322,
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/10/2002, TWP
% $Revision: 2.0 $  $Date: 2002.03.10 16:03:00 $
%
%---------------------------------------------------------------
if (n<3)
    retval = 1;
else
% Note that the third and fourth arguments here are 1 and 1, not
% 1 and 0 as for the Fibonacci numbers in modfib.m
    retval = helpSOR(n, 1, 1, 1, freq, gain);
end

function retval = helpSOR(n, k, sork, sorkm1, freq, gain)
%----------------------------------------------------------------
% Input arguments:
%       n      = current argument
%       k      = variable argument 1<=k<=n
%       sork   = response for the last k
%       sorkm1 = memory of response for k-1
%       freq   = frequency of response
%       gain   = gain of amplitude of response
%
%       The helper function calculates recursively the current
%       response by swapping the second and third argument,
%       so that the current response becomes the previous one
%       in the next call with k=k+1
%
% Written by T. W. Patzek, March 27, 2001
% UC Berkeley, 437 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834 or 510-486-5322,
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/10/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.10 16:03:00 $
%-----------------------------------------------------------------
if (n == k)
   retval = sork;
   return
else
   retval = ...
    helpSOR(n,k+1,2*(1-freq)*sork-gain*sorkm1,sork,freq,gain);
%                 |-------------------------|   ^
%                             |                 | Current value
%                             |_____New value   | moves to memory
%                                   of response   as sorkm1
end
```

The results are plotted in **Figure 15.6 - Figure 15.8**.

15.5 Exercises

1. Write the iterative version of the recursive SOR algorithm with memory.

Chapter 16

Quadratic Iterator: Don't Trust Your Computer*

16.1 What Are You Going To Learn?

In this chapter we will remind you about the quadratic iterator, i.e.,, a single-step feedback machine that iterates a quadratic function. You will see that such an iterator represents broad classes of problems, including the law of logistic population growth. You will also see that a simple quadratic iterator leads to computations which expose the inherent limitations of computers. Depending on the properties of the *iterated* function $f(x)$, the process of *direct iteration*: $x_{n+1} = f(x_n) = ax_n(1 - x_n)$, may converge to a fixed point, a cycle, or not converge at all. The material of this lecture follows closely [27].

In Chapter 16.1 folder on fttp:\\petroleum.berkeley.edu, look up the following MATLAB files:

IterationDemo.m A demo that calls `GraphicalIteration`.

GraphicalIteration.m performs graphical iteration with a quadratic iterator.

qiterator.m is a general quadratic iteration function, given the initial value of x and the constant c.

16.2 Why Is It Important?

Modern computers allow scientists to perform computations which are so complex and extensive that humans no longer can digest fully their enormity. In such massive computations it is often true that a comprehensive error propagation analysis is impossible, and this realization has lead to a very dangerous trend. Many engineers and scientists are foolish enough to have absolute confidence in the correctness of their computations. Such an unjustified confidence may lead to disastrous results, and loss of property and life.

16.3 Quadratic Iterators

The quadratic iterator, **Figure 16.1**, belongs to the family of one-step feedback machines

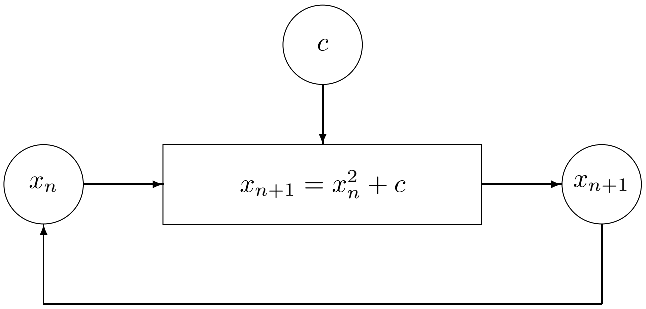


Figure 16.1: The quadratic iterator with a control parameter c.

$$x_{n+1} = f(x_n),\ n = 0, 1, \dots$$

and uses the simplest quadratic function for f:

$$f(x_n) = x_n^2 + c \tag{16.1}$$

Here x_i and c are just numbers, but their roles are different. To iterate the expression above for a fixed control value c means this: start with an x, evaluate the expression, use the result as new x, and so on. Here is an example:

Preparation: Choose a value for c, say $c = -1$. Then choose an initial value for x, say $x = 0.1$.

Iteration: Evaluate the expression for x, obtaining $0.01 - 1 = -0.99$. Now use the new value $x = 0.99$ to evaluate the same expression again, obtaining $x = 0.0199$, and so on, **Figure 16.2**.

The table below summarizes the results of the first 10 iterations:

Iteration	x	$x^2 + c$
1	0.100000000000000	-0.990000000000000
2	-0.990000000000000	-0.019900000000000
3	-0.019900000000000	-0.999603990000000
4	-0.999603990000000	-0.000791863176080 ←
5	-0.000791863176080	-0.999999372952710
6	-0.999999372952710	-0.000001254094186
7	-0.000001254094186	-0.999999999998427
8	-0.999999999998427	-0.000000000003145
8	-0.000000000003145	-1.000000000000000
10	-1.000000000000000	0.000000000000000

After a mere 4 iterations we are running into a serious problem. In each iteration, the squaring operation essentially doubles the number of digits necessary to represent the answer. This makes it *impossible* to achieve correct results after more than 4 iterations, when using double precision in MATLAB. Of course if we used quadruple precision (128 bits), we would delay the loss of accuracy for another four iterations, and so on. The computers are *finite* (For more detailed warnings refer to Chapter 5.)

Here is the MATLAB code:

```
function qiterator(x,c)
%
% E77N, TWP, 29.04.2001, A general quadratic iterator machine
% with the starting point x and the constant c.
%
if nargin==0
    x=0.1;
    c=-1;
elseif nargin<2
    c=-1;
end
if (imag(x)~=0 | imag(c) ~=0)
    error('Imaginary parts of input arguments will be ignored\n');
end
clc
n=60;
ys=zeros(n,1);
clc;
for i=1:n
    ys(i)=x;
    xold=x;
    x=x^2+c;
    fprintf('%17.15f %17.15f\n',xold,x);
end
% Plot the iterator response
close all
```

```
figure(1)
xs=1:1:n;
plot(xs,ys);
xlabel('Time')
ylabel('Iterator response')
s = sprintf('c=%g, x_0=%g',c, ys(1));
title(s)
set(gcf,'PaperPosition',[1 1 4.5 3.2])
fn=sprintf('qiterator%g%g.eps',ys(1),c)
%print('-depsc2', fn)
```

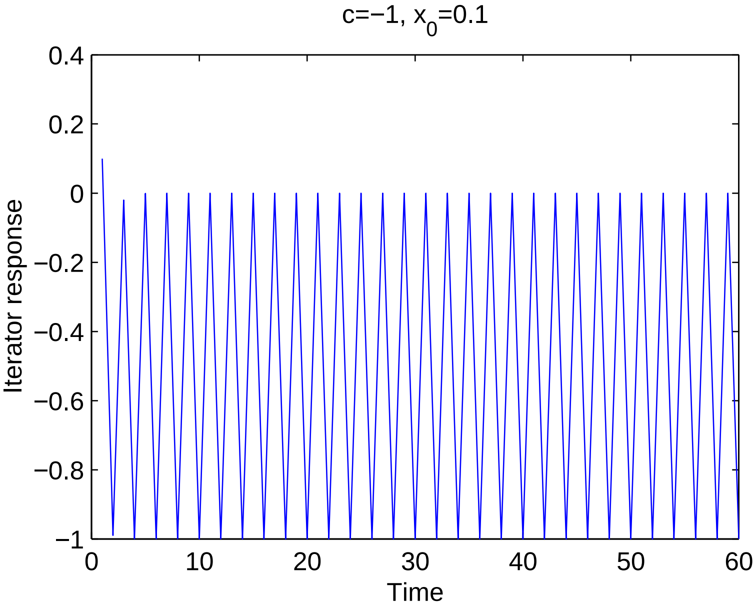


Figure 16.2: The "time series" of a quadratic iterator response.

16.3.1 Error Propagation Matters!

At this point, if we just shrug our arms and proceed, we will abandon the unbelievable battles for accuracy in measurement and computation fought by the great heros of science and engineering, Tycho Brahe, Johannes Kepler, Galileo Galilei, Sir Isaak Newton, Leonhard Euler, Carl Friedrich Gauss, just to name a few.

To point out just how limited computations are, let us quote from James Glick's *Chaos, Making a New Science*:

> "The modern weather models work with a grid of points on the order of sixty miles apart, and even so, some starting data has to be

> guessed, since ground stations and satellites cannot see everywhere. But suppose the earth could be covered with sensors spaced one foot apart, rising at one-foot intervals all the way to the top of the atmosphere. Suppose that every sensor gives perfectly accurate readings of temperature, pressure, humidity, and any other quantity a meteorologist would want. Precisely at noon an infinitely powerful computer takes all the data and calculates what will happen at each point at 12:01, then 12:02, then 12:03, ... The computer will still be unable to predict whether Princeton, New Jersey, will have sun or rain on a day one month away. At noon the spaces between the sensors will hide fluctuations that the computer will not know about, tiny deviations from the average. By 12:01, those fluctuations will have already created small error one foot away. Soon the errors will be multiplied to the ten-foot scale, and so on up to the size of the globe."

16.3.2 More on the Quadratic Iteration

We will now iterate the quadratic expression of the form

$$p_{n+1} = p_n + rp_n(1 - p_n),\ n = 0, 1, \ldots$$

This equation is equivalent to our prototype quadratic iterator $x \to x^2 + c$. The quadratic expression above has a very interesting interpretation in the history of biology. It was at the core of the first realistic[1] population dynamics model, which goes back to the Belgian mathematician Pierre Françoise Verhulst [31], **Figure 16.3**.

A population dynamics model allows us to predict the future growth of a biological population. As with Fibonacci's rabbits, time is measured in increments $n = 1, 2, \ldots$ (seconds, hours, years, generations, whatever is appropriate). The size of the population at time n is measured by the actual number of individuals, P_n. The size of a population may depend on many factors, such as food, climate, predators, age structure, fertility, mortality, etc. Let us denote the maximum population size supported by the environment as N, and the fractional population size as $p_n = P_n/N$.

After Verhulst, we shall assume that the *relative* growth rate of the population is proportional to the fraction of its size that has not been yet exhausted by the current population, and the proportionality constant is r:

$$\frac{p_{n+1} - p_n}{p_n} = r(1 - p_n),\ n = 0, 1, \ldots$$

Therefore

$$p_{n+1} = p_n + rp_n(1 - p_n),\ n = 0, 1, \ldots \tag{16.2}$$

Equation 16.2 is the famous *logistic* growth law of Verhulst.

[1] Please recall Fibonacci's rabbits.

Figure 16.3: PIERRE FRANÇOIS VERHULST, 1804-1849.

16.3.3 Equivalence of Logistic Equation and Quadratic Iterator

It can be verified by *induction* that the logistic equation (16.2) and the quadratic iterator equation are equivalent with the following *change of scale*:

$$\begin{aligned} c &= \frac{1-r^2}{4} \\ x_n &= \frac{1+r}{2} - rp_n \end{aligned} \tag{16.3}$$

This change of scale is similar to the change of temperature scale in which a temperature p in degrees Fahrenheit corresponds to

$$x = \frac{5}{9}(p - 32)$$

in degrees Celsius.

Remark 15 The quadratic iterator and the logistic population balance are equivalent with the change of scale (16.3_1) - (16.3_2).

Therefore, to represent all quadratic iterators, we can study just the simplest one, $x \leftarrow x^2 + c$, or its cousin, $x \leftarrow ax(1-x)$.

16.3.4 Is Loss of Precision Alone Responsible For the Sometimes Wild Behavior of the Iterator?

As we have already seen, if there is an error in the feedback process, this error is amplified by the squaring process. Therefore we are tempted to blame error propagation alone for the chaotic behavior of the iterator. It is the case, but in a much more subtle way that we might think. (In the previous lecture, this subtlety lead to the MANDELBROT set and the JULIA sets.) It then follows that squaring alone does not explain anything!

If we choose $x_0 = 1.97$ and $c = -2$, the iterator behaves chaotically, **Figure 16.4**.

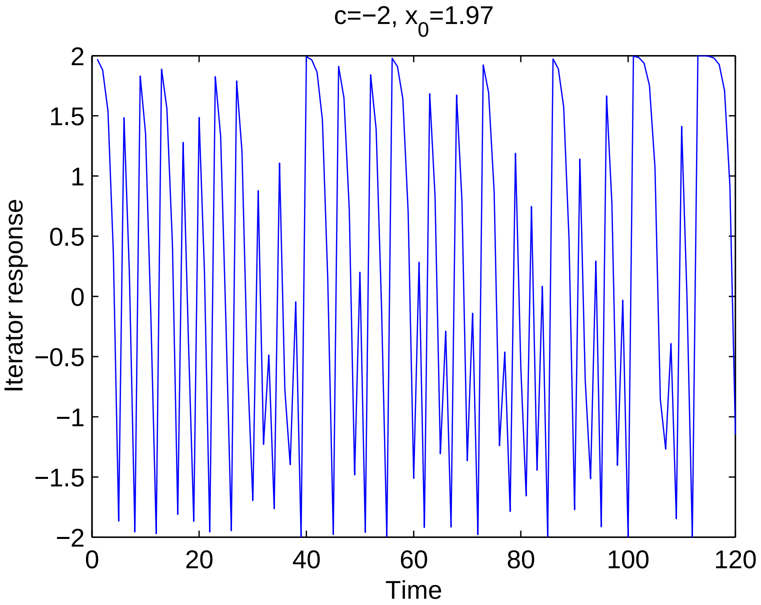


Figure 16.4: The "time series" of a chaotic response of the quadratic iterator.

But in Figure 16.2, we saw that the same iterator behave perfectly reasonably, albeit with a different constant c. We get the same perfect behavior if start the iterator with $x_0 = 1$. Then the iterator converges immediately to $-1, -1, \ldots$. If we start with $x_0 = 2$, the iterator will settle on $2, 2, \ldots$. In other words, there exist values of x_0, for which the same iterator (with the same c) behaves perfectly tamely. We could demonstrate that this orderly behavior of the iterator is an exception, i.e., for almost any $x \in [-2, 2]$ the iterator behaves chaotically. We can start with a very good approximation of $x_0 = 2$, e.g., $x_0 = 1.99999999$, and the result is shown in **Figure 16.5**.

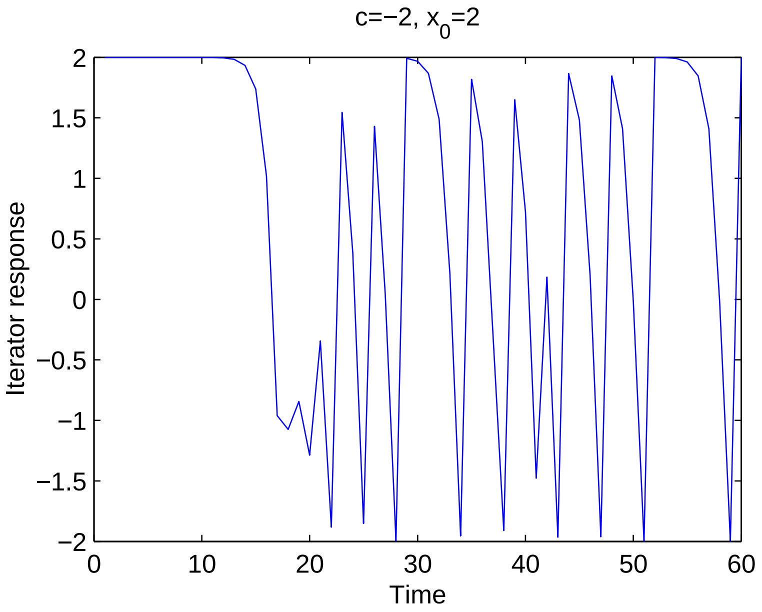


Figure 16.5: The "time series" of a chaotic response of the quadratic iterator for $x_0 = 1.99999999$ (rounded in the title to 2).

This mess shows that the behavior of the iterator is not so straightforward. Our difficulties are only amplified if we set $c = -1$ in the iterator. Then $x_0 = 0.1$ or $x_0 = 0.25$ or $x_0 = 0.754$ yield the same answer, shown if Figure 16.2. The feedback process is now perfectly stable.

Remark 16 The quadratic iterator may be either stable or wildly unstable depending on its constant c and the starting point x_0. The chaotic behavior of this iterator has little to do with round-off error caused by the repetitive squaring, but depends essentially the shape of the parabola, i.e., the on the constant c. □

16.3.5 Graphical representation of the feedback process

Here we will use a quadratic iterator on the interval $x \in [0, 1]$

$$x_{n+1} = ax_n(1 - x_n) \tag{16.4}$$

with the initial guesses on the same interval. The graph of the function $y = ax(1-x)$ is a parabola with $y = 0$ at $x = 0$ and $y = 0$ at $x = 1$, independent of the choice of a. The vertex of the parabola is at $(1/2, a/4)$. Of course the quadratic iterator (16.4) is equivalent to our basic iterator $x_{n+1} = x_n^2 + c$ under a change of scale. It is also equivalent to the logistic curve iterator under the following change of scale:

$$\begin{aligned} a &= r+1 \\ x_n &= \frac{r}{r+1} p_n \end{aligned} \tag{16.5}$$

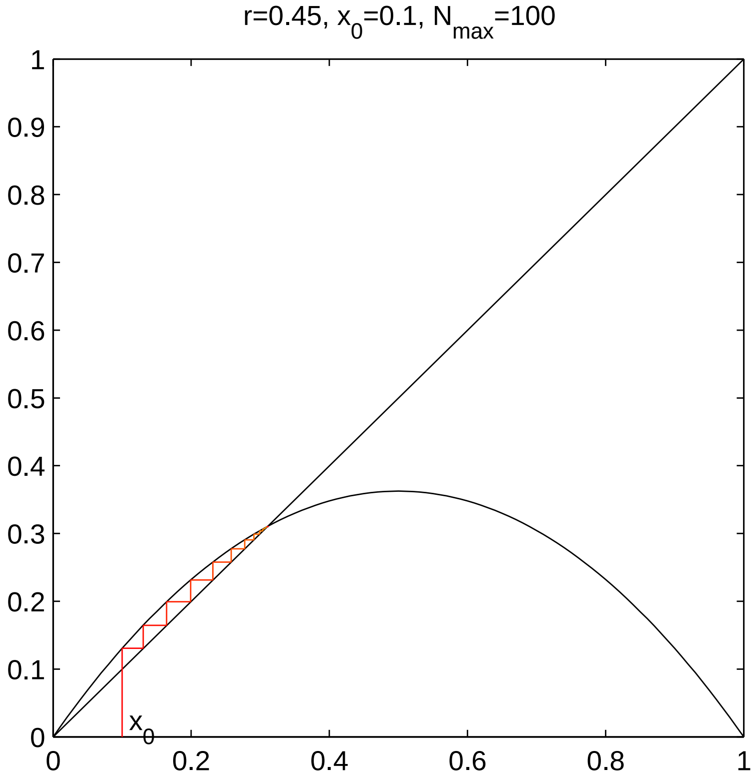


Figure 16.6: The "time series" of a fixed-point response of the quadratic iterator.

There is a very simple way of constructing the graphical representation of the iteration process. First we plot the diagonal and the parabola. Then we go vertically from the initial guess $(x_0, 0)$ to the parabola (x_0, x_1), then horizontally to the diagonal (x_1, x_1), then vertically to the parabola (x_1, x_2), and so on. Here is the MATLAB code:

```
function GraphicalIteration(r,x0,nmax)
```

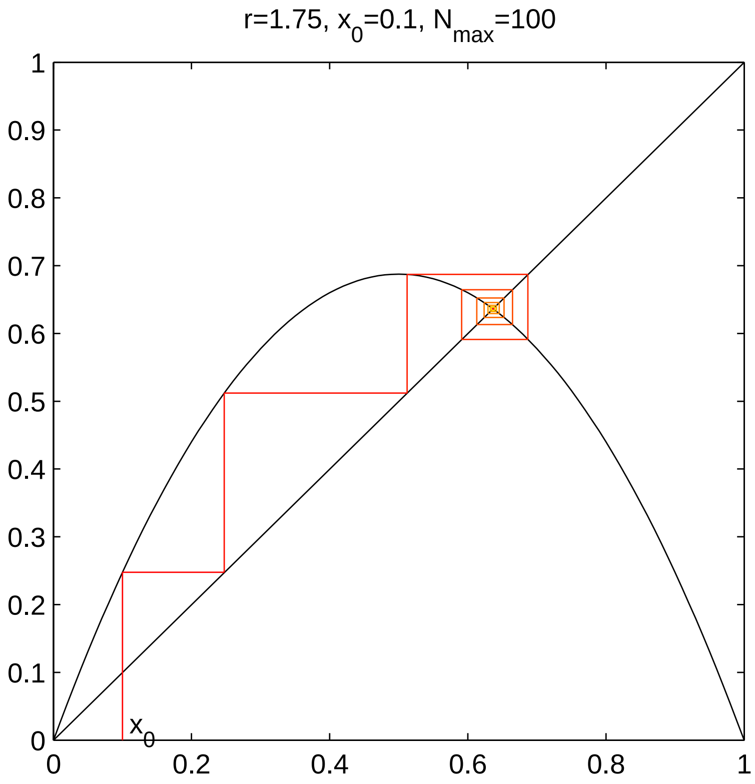


Figure 16.7: The "time series" of a fixed-point response of the quadratic iterator.

```
% Perform graphical iteration with a quadratic iterator
% T.W. Patzek, E77N, April 28, 2001
%
SS = get(0, 'ScreenSize');
figure('Visible','on', 'Units','Pixels',...
       'Resize','on', 'Name','Graphical Iteration', 'Numbertitle','off',...
       'Position',[50,50,0.708*SS(3),0.85*SS(4)], 'BackingStore', 'off');
get(gca)
set(gca,'XLim', [0,1], 'YLim',[0,1]);
drawnow
c= colormap(hsv(nmax));
%axis equal
x=0:0.01:1;
a=r+1;
y=a*x.*(1-x);
hold on
plot(x,y,'color','k');
plot([0,1],[0,1],'color','k');
xi=x0;
```

```
for i=1:nmax
    xip1=a*xi*(1-xi);
    if (i==1)
        plot([xi,xi],[0,xip1], 'color', c(i,:));
    else
        plot([xi,xi],[xi,xip1],'color', c(i,:));
    end
    plot([xi,xip1],[xip1,xip1],'color', c(i,:));
    xi=xip1;
end
s=sprintf('r=%g, x_0=%g, N_{max}=%g',r,x0,nmax)
title(s)
s='x_0';
text(x0+0.01,+0.015,s);
box on
set(gcf,'PaperPosition',[1 1 4.5 4.2])
fn=sprintf('qiterator2-%g%g.eps',x0,r)
%print('-depsc2', fn)
```

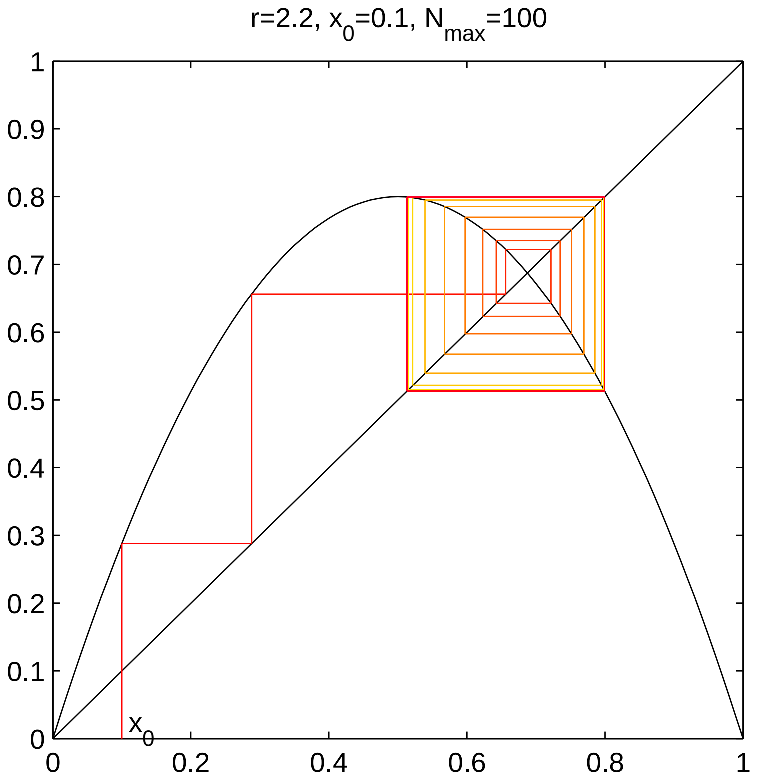


Figure 16.8: The “time series” of a cyclic response of the quadratic iterator.

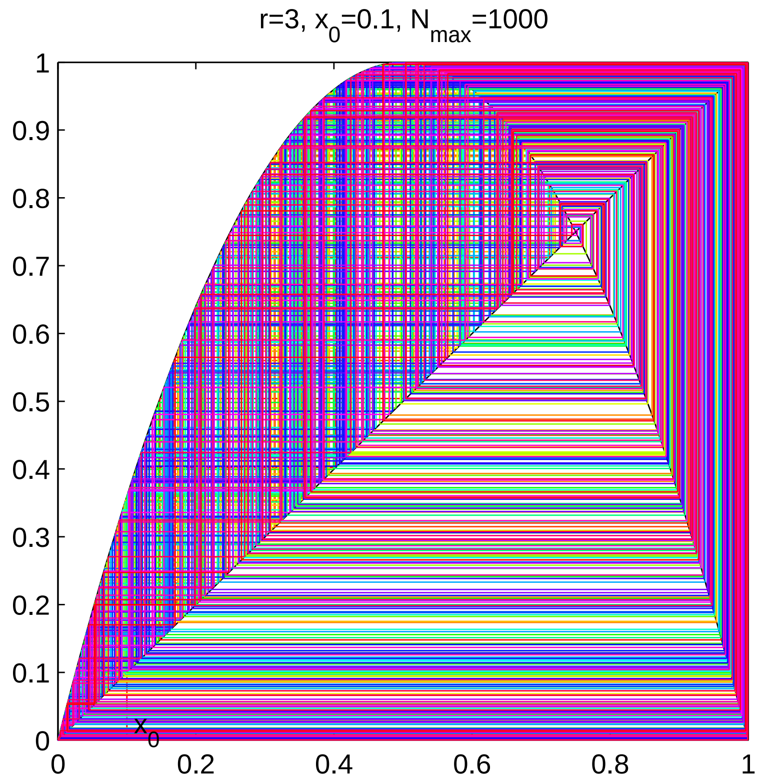


Figure 16.9: The "time series" of a chaotic response of the quadratic iterator.

Figures 16.6 - 16.9 illustrate the results.

16.4 Exercises

1. Prove by induction that the logistic population equation (16.2) and the quadratic iterator equation (16.1) are equivalent under the change of scale given by Eq. (16.3).

2. Prove by induction that the quadratic iterator (16.1) and the quadratic iterator (16.4) are equivalent under the change of scale (16.5).

Chapter 17

Polynomials and Taylor Series

17.1 What Are You Going To Learn?

You will learn how to work with polynomials using MATLAB. In Chapter 17 folder on fttp:\\petroleum.berkeley.edu, you will find the following MATLAB files:

taylorexp.m computes a Taylor Series expansion of the function $f(x) = exp(x)$ about $x_0 = 0$ on the interval $[a, b]$.

taylorlog.m computes a Taylor Series expansion of the function $f(x) = \ln(x)$ about $x_0 = 1$ on the interval $[a, b]$.

17.2 Why Is It Important?

Operations involving polynomials appear in interpolation, extrapolation, regression, numerical differentiation, numerical integration, just to name a few. Taylor series is the most basic polynomial approximation of a smooth function.

17.3 Useful Background

Most commonly used functions, e.g., $y = \sin(x)$, $y = \ln(x)$, $y = \sinh(x)$, as well as most results of physical measurements, cannot be calculated from a simple algebraic formula, such as $y = 3 + 2x$. If we want to *estimate* a value, $y = f(x)$, of a function[1], f, of a single independent variable[2], x, we have three choices:

[1] Also called a *dependent variable* or *ordinate.*

[2] Also called *abscissa.*

1. The function $f(x)$ is sufficiently *smooth*, i.e., it has *continuous* first, second, third, ..., nth derivative over some closed interval $[a, b]$ containing the point x. In this case, we can choose a *reference* abscissa[3] $x_0 \in [a, b]$. We then evaluate the function and its derivatives *only* at x_0, and form a polynomial TAYLOR series to approximate the function value at an arbitrary abscissa x in the neighborhood $[a, b]$ of x_0. Thus, for a sufficiently smooth function we can approximate its behavior in a neighborhood of a given point, x_0, from the knowledge of the function value and the values of the function's derivatives, all evaluated at x_0.

2. It may happen that we *do* know exactly the values of the abscissa, $x_1, x_2, \ldots, x_n$, but we do *not know* exactly the corresponding values of the ordinate, $y_1 = f(x_1), \ldots, y_n = f(x_n)$. Then we may choose to approximate the function globally with a simple known *model*, such as a straight line or a polynomial, so as to minimize the sum of squares of errors of the approximation of the function values. This approach is called the *general linear least squares fit*, or *regression*, and it will be discussed in Lecture 19 and 20.

3. The function $f(x)$ is merely *continuous*, but we know its values at many points: (x_1, y_1), $(x_2, y_2), \ldots, (x_n, y_n)$. If x is between the largest and smallest of the x_i's, we call the process of finding $f(x)$ *interpolation*; otherwise, we call this process *extrapolation* (Lecture 21).

Remark 17 Approximation with a TAYLOR series is never *exact*, in the sense that we do not know the exact value of the function at any point in a closed neighborhood of x_0. We do know, however, the conditions which make our approximation "close enough" to the function at all points x in this neighborhood. □

Remark 18 Interpolation is *exact* by definition, in the sense that the interpolating function passes through each and every point. □

Remark 19 Extrapolation is usually a bad practice, and it leads to completely meaningless estimations in almost all cases. □

There are two types of interpolants:

1. *Global interpolants.* The estimated value $\hat{y}$ depends on all the points (x_i, y_i), $i = 1, 2, \ldots, n$. The LAGRANGE interpolation polynomial of degree $n - 1$ is an example.

2. *Local interpolants.* The estimated value $\hat{y}$ depends only on the points in the neighborhood of x. The piecewise linear, quadratic or cubic interpolations are examples.

[3] For example, $x = 0$ or $x = 1$, at which the function and its derivatives are especially simple to calculate.

17.4 Polynomials

17.4.1 Representation

We write

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

MATLAB uses a different notation:

$$P_n(x) = a_{n+1} x^n + a_n x^{n-1} + \cdots + a_3 x^2 + a_2 x + a_1,$$

and therefore the coefficients can be represented in a row vector:

$$\boldsymbol{a} = [a_{n+1}, a_n, \ldots, a_3, a_2, a_1].$$

Remark 20 A polynomial of degree n has $n+1$ coefficients.

17.4.2 Evaluation of polynomials

Consider the polynomial $P(x) = 9x^3 - 5x^2 + 3x + 7$, and suppose we want to evaluate this polynomial at $x = 2$. You should *never* write something like this:

```
>> 9*2^3-5*2^2+3*2+7
```

Quoting the marvellous book *Numerical Recipes* by Press *et al.* [28]:

> *Come the (computer) revolution, all persons found guilty of such criminal behavior will be summarily executed and their programs won't be!*

To evaluate a polynomial at a point x, MATLAB has a built-in function: polyval. Study the following commands and their corresponding output:

```
>> P = [9,-5,3,7];
>> polyval(P,2)
ans =
    65
>> x = [-2:.05:2];
>> plot(x,polyval(P,x))
```

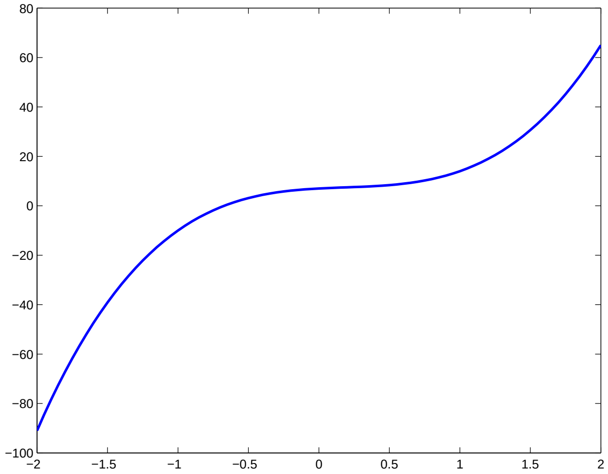


Figure 17.1: MATLAB plot of the polynomial $P(x) = 9x^3 - 5x^2 + 3x + 7$

17.4.3 Addition of polynomials

We add or subtract polynomials by adding or subtracting the coefficients of like powers of x. Given the polynomials

$$P_n(x) = \sum_{i=0}^{n} a_i x^i, \qquad Q_n(x) = \sum_{i=0}^{n} b_i x^i,$$

the sum of the two is

$$S_n(x) = P_n(x) + Q_n(x) = \sum_{i=0}^{n} (a_i + b_i) x^i.$$

Example: Let

$$\begin{aligned} P(x) &= 9x^3 - 5x^2 + 3x + 7 \\ Q(x) &= \mathbf{0}x^3 + 6x^2 - 1x + 2 \end{aligned}$$

```
>> P
P =
     9    -5     3     7
>> Q = [0,6,-1,2]
Q =
     0     6    -1     2
>> S=P+Q
S =
     9     1     2     9
>> plot(x,polyval(P,x),’-.’,’LineWidth’,1.5)
>> hold on
```

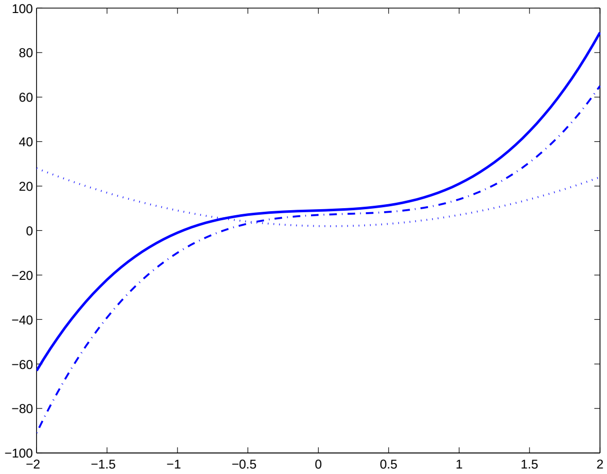


Figure 17.2: MATLAB plot of the polynomials $P(x) = 9x^3 - 5x^2 + 3x + 7$ ($-\cdot$), $Q(x) = 6x^2 - x + 2$ ($\cdots$), and $S(x) = P(x) + Q(x)$ ($-$)

```
>> plot(x,polyval(Q,x),':','LineWidth',1.5)
>> plot(x,polyval(S,x),'-','LineWidth',2)
```

17.4.4 Multiplication (convolution) of polynomials

We shall explain convolution of polynomials with an example, rather than with a general formula. Consider the same polynomials $P(x)$ and $Q(x)$ above. Their convolution (please note the operator $\star$) is also another polynomial, given by the usual distributive law:

$$\begin{aligned}
C(x) &= P(x) \star Q(x) \\
&= (9x^3 - 5x^2 + 3x + 7)(6x^2 - x + 2) \\
&= 54x^5 - 30x^4 + 18x^3 + 42x^2 \\
&\qquad\qquad - 9x^4 + 5x^3 - 3x^2 - 7x \\
&\qquad\qquad\qquad + 18x^3 - 10x^2 + 6x + 14 \\
&= 54x^5 - 39x^4 + 41x^3 + 29x^2 - x + 14
\end{aligned}$$

MATLAB has the built-in function conv to perform convolution of polynomials:

```
>> P = [9,-5,3,7]
P =
     9    -5     3     7
>> Q = [6,-1,2]
Q =
     6    -1     2
```

```
>> conv(P,Q)
ans =
    54   -39    41    29    -1    14
```

17.4.5 Division (deconvolution) of polynomials

In order to understand division of polynomials, we need to understand division of integers first. If we did not know how to operate with *rational numbers*, the precise meaning of $\frac{19}{5}$ is as follows: there exist two numbers, 3 and 5 (quotient and remainder, respectively), such that

$$19 = 5 \times \underbrace{\mathbf{3}}_{\text{quot}} + \underbrace{\mathbf{4}}_{\text{rem}}$$

The numbers involved in the equation above are all *integers*. Division of polynomials works in the same way. Given two polynomials, $U(x)$ and $V(x)$, there exist two *polynomials* $Q(x)$ and $R(x)$, such that:

$$\frac{U(x)}{V(x)} \Longleftrightarrow U(x) = V(x) \star Q(x) + R(x)$$

For example, if $U(x) = 2x^3 + x^2 + 5,\ V(x) = 2x + 3$, then

$$2x^3 + x^2 + 5 = (2x + 3) \star (\underbrace{x^2 - x + 1.5}_{\text{quot}}) + (\underbrace{0.5}_{\text{rem}})$$

MATLAB provides the function deconv to calculate the quotient and the remainder of a division of polynomials. For the example above:

```
>> U
U =
     2     1     0     5
>> V
V =
     2     3
>> [Q,R]=deconv(U,V)
Q =
    1.0000   -1.0000    1.5000
R =
         0         0         0    0.5000
```

17.4.6 Rational functions

These are functions of the form

$$R(x) = \frac{P_n(x)}{Q_m(x)}$$

where $P_n(x)$ and $Q_m(x)$ are polynomials. In general $R(x)$ is not a polynomial, but can be evaluated by evaluating each polynomial separately.

17.5 Taylor series expansion

Figure 17.3: Brook Taylor was born on August 18, 1685, in Edmonton, Middlesex, England, and died on December 29, 1731, in Somerset House, London, England.

The idea behind Taylor series[4] is to approximate a function $f(x)$ in the neighborhood of a point x_0 using a polynomial. The Taylor series expansion of a function $f(x)$ (which is assumed to be sufficiently differentiable) about a point x_0 is:

$$\begin{aligned} f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \frac{f'''(x_0)}{6}(x-x_0)^3 + \cdots \\ + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + R_{n+1} \end{aligned}$$

The term R_{n+1} is the remainder after n terms. Note that only the value of the function and its derivatives at *one* point is needed.

You may wonder where do the awkward coefficients $f^{(i)}(x_0)/i!$ come from. They come from imposing that the Taylor series should be exact if the function $f(x)$ were actually a polynomial. Suppose we have

$$f(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)^2 + a_3(x-x_0)^3 + \cdots$$

We can compute the coefficients $a_0,\ a_1,\ a_2,\ a_3\ldots$ by evaluating the function and its derivatives at the point $x = x_0$. This is how it works:

$$\begin{aligned} f(x_0) &= \big[a_0 + a_1(x-x_0) + a_2(x-x_0)^2 + a_3(x-x_0)^3 + \cdots\big]_{x_0} = a_0 \\ f'(x_0) &= \big[a_1 + 2a_2(x-x_0) + 3a_3(x-x_0)^2 + \cdots\big]_{x_0} = a_1 \\ f''(x_0) &= [2a_2 + 3\cdot 2a_3(x-x_0) + \cdots]_{x_0} = 2a_2 \\ f'''(x_0) &= [3\cdot 2a_3 + \cdots]_{x_0} = 3\cdot 2a_3 \end{aligned}$$

[4] Published by Brook Taylor in 1715 in his *Methodus incrementorum directa et inversa*. When $x_0 = 0$, the Taylor series is called Mclaurin series. In fact, the Taylor series had been known to James Gregory and Jean Bernoulli long before Taylor published it [2].

so we get the general relation

$$a_i = \frac{f^{(i)}(x_0)}{i!}, \quad i = 0, \ldots, n$$

and the formula for the TAYLOR series follows.

17.5.1 Example 1. Exponential function

This is the easiest possible example. We want to obtain the Taylor series expansion of $f(x) = \exp(x)$ about $x_0 = 0$. In this case

$$f(x) = f'(x) = f''(x) = \cdots = f^{(n)}(x) = \exp(x),$$

and thus

$$f(0) = f'(0) = f''(0) = \cdots = f^{(n)}(0) = \exp(0) = 1.$$

Hence, the TAYLOR series is:

$$\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

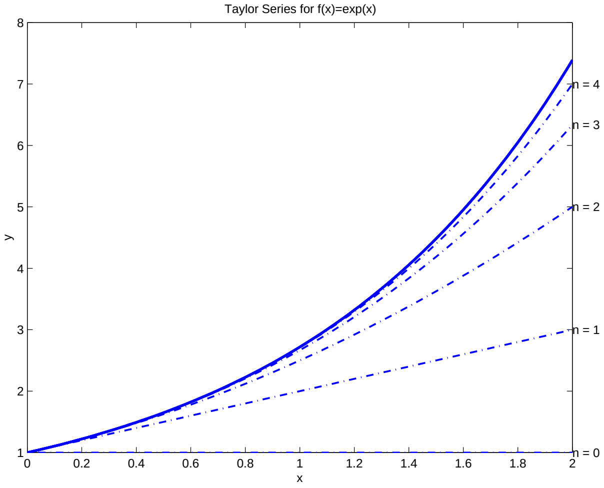


Figure 17.4: TAYLOR series approximation of $f(x) = \exp(x)$.

17.5.2 Example 2. Natural logarithm

We now consider the TAYLOR series expansion of $f(x) = \ln(x)$ about $x_0 = 1$. In this case

$$\begin{array}{rcr}
f(x) = \ln(x) & \longrightarrow & f(1) = 0 \\
f'(x) = x^{-1} & \longrightarrow & f'(1) = 1 \\
f''(x) = -x^{-2} & \longrightarrow & f''(1) = -1 \\
f'''(x) = 2x^{-3} & \longrightarrow & f'''(1) = 2 \\
f''''(x) = -6x^{-4} & \longrightarrow & f''''(1) = -6 \\
\dots & & \dots \\
f^{(i)}(x) = (-1)^{i-1}(i-1)!\,x^{-i} & \longrightarrow & f^{(i)}(1) = (-1)^{i-1}(i-1)!
\end{array}$$

Hence, the TAYLOR series is:

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} + \cdots = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

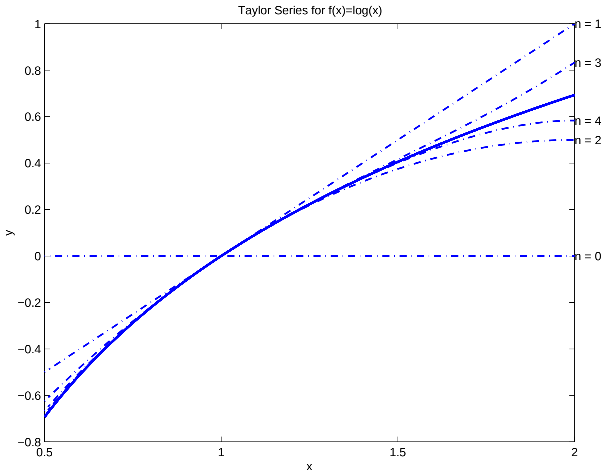


Figure 17.5: TAYLOR series approximation of $f(x) = \ln(x)$.

17.6 Taylor Series for e^x

```
function taylorexp(a,b,n)
%
% Function which computes a Taylor Series
% expansion of the function f(x)=exp(x)
% about x0=0 on the interval [a,b]
%
% Input: a = lower limit of the interval
%        b = upper limit of the interval
%        n = no of terms of the expansion
%
% Output: none
%         (the function generates a plot)
%
% Author: Ruben Juanes
% Last modified: 2/27/2001
%
% data set
np = 101;
x = linspace(a,b,np);

% polynomial coeffs of the Taylor Series
coeff = [n:-1:0];
for i = 1:n+1
    coeff(i) = 1/factorial(coeff(i));
end
texp = polyval(coeff,x);

% plot the approximation
plot(x,exp(x),'-','LineWidth',2)
hold on
plot(x,texp,'-.','LineWidth',1.5)
text(x(np),texp(np),['n = ',num2str(n)])
title('Taylor Series for f(x)=exp(x)')
xlabel('x'),ylabel('y')
```

17.7 Taylor Series for $\ln x$

```
function taylorlog(a,b,n)
%
% Function which computes a Taylor Series
% expansion of the function f(x)=log(x)
% about x0=1 on the interval [a,b]
%
% Input: a = lower limit of the interval
%        b = upper limit of the interval
%        n = no of terms of the expansion
%
```

```
% Output: none
%         (the function generates a plot)
%
% Author: Ruben Juanes
% Last modified: 2/27/2001
%
% data set
np = 101;
x = linspace(a,b,np);
xi = x-1;

% polynomial coeffs of the Taylor Series
coeff(n+1) = 0;
vsign = 1;
for i = n:-1:1
    coeff(i) = vsign*1/(n+1-i);
    vsign = -vsign;
end
tlog = polyval(coeff,xi);

% plot the approximation
figure(1)
plot(x,log(x),'-','LineWidth',2)
hold on
plot(x,tlog,'-.','LineWidth',1.5)
text(x(np),tlog(np),['n = ',num2str(n)])
title('Taylor Series for f(x)=log(x)')
xlabel('x'),ylabel('y')
```

Chapter 18

Solution of Systems of Linear Equations

18.1 What Are You Going To Learn?

In this chapter you will learn the rudiments of solving small systems of linear equations. Such systems arise in many practical problems in engineering and science.

- What are linear equations?
- Determined systems of linear equations
- Underdetermined systems of linear equations
- Overdetermined systems of linear equations
- Example: an electric circuit problem

The Chapter 18 folder on fttp:\\petroleum.berkeley.edu has the following files:

circuit.m solves an electric circuit problem from Palm.

lseq.m shows the behavior of 2x2 systems of linear equations: determined, underdetermined and overdetermined.

18.2 Why Is It Important?

Linear algebra is the bread and butter of engineering and science. In the end, we always solve a system of linear algebraic equations or linearize nonlinear equations locally and then solve the resulting linear equations. Thousands of books have been written on this subject; I personally like Berkeley's Demmel [8], Householder [17] and Higham [16].

18.3 Linear systems of equations

In this book we shall consider the following two standard problems of linear algebra:

- Solve a system of linear equations, $Ax = b$. Here A is a known n-by-n real or complex matrix, which is nonsingular (its *determinant* is not equal to zero); b is an n-by-1 known column vector; and x is an n-by-1 column vector of unknowns.

- Solve a least square problem by computing the vector x that minimizes $\|Ax - b\|_2$. Here A is m-by-n, b is m-by-1, x is n-by-1, and

$$\|Ax - b\|_2 \equiv \sqrt{\sum_{i=1}^{i=m} |(Ax)_i - b_i|^2}$$

  is called the *two-norm* of vector $Ax - b$, and we have omitted the column index of the components of the two column vectors under the square root, i.e., $b_i = b_{i1}$ etc.. If $m > n$, there are more equations than unknowns, the system of equations is *overdetermined*, and in general we cannot solve $Ax = b$ exactly. If $m < n$, the system is *underdetermined*, and has *no* or *infinitely* many solutions.

18.3.1 Introduction

Systems of linear equations look like this:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \cdots\ a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & \cdots\ a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & \ldots & & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & a_{m3}x_3 & + & \cdots\ a_{mn}x_n & = & b_m \end{array}.$$

Using matrix-vector notation:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix},$$

or in compact form

$$\underbrace{A}_{m\times n}\underbrace{x}_{n\times 1} = \underbrace{b}_{m\times 1},$$

where A is the matrix of coefficients, x is the vector of unknowns, and b is the vector of independent terms (the right hand side vector).

You need to know the following preliminaries before we start our discussion on the possible different cases which may arise.

- *Determinant* of a square matrix. If the determinant is zero, the matrix is singular, i.e., one or more rows may be expressed as a linear combination of other rows. The determinant can be computed in MATLAB with the function **det**.

- *Rank* of a rectangular matrix. The rank of a matrix is the number of linearly independent rows or columns, whichever is lower, i.e., the size of the maximum nonzero *sub-determinant*. The matrix rank can be computed in MATLAB with the function **rank**.

Example. Consider the following two matrices,

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \qquad B = \begin{pmatrix} 2 & -4 & 5 \\ -4 & -2 & 3 \\ 2 & 6 & -8 \end{pmatrix}.$$

```
>> A = [1,2;3,4];
>> B = [2,-4,5;-4,-2,3;2,6,-8];
>> det(A)
ans =
    -2
>> rank(A)
ans =
     2
>> det(B)
ans =
     0
>> rank(B)
ans =
     2
```

18.3.2 Determined systems

When the number of linearly independent equations is equal to the number of unknowns, $m = n$, this system of linear equations is determined. The precise condition is

$$\text{rank}(A) = \text{rank}([A, b]) = n,$$

where $[A, b]$ is a $n \times (n + 1)$ matrix, which consists of the original matrix A augmented with the right hand side vector b. In this case, the solution x exists and is unique.

Example. Consider the following 2 × 2 system of equations:

$$\begin{pmatrix} 1 & 2 \\ 3 & -4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}.$$

Here

$$A = \begin{pmatrix} 1 & 2 \\ 3 & -4 \end{pmatrix}, \quad [A, b] = \begin{pmatrix} 1 & 2 & 5 \\ 3 & -4 & 6 \end{pmatrix}$$

Each equation represents a straight line in the $x - y$ plane. The first line is given by $y = (5 - x)/2$, and the second one is $y = (3x - 6)/4$. The solution of the system is given by the location where the lines intersect (**Figure 18.1**).

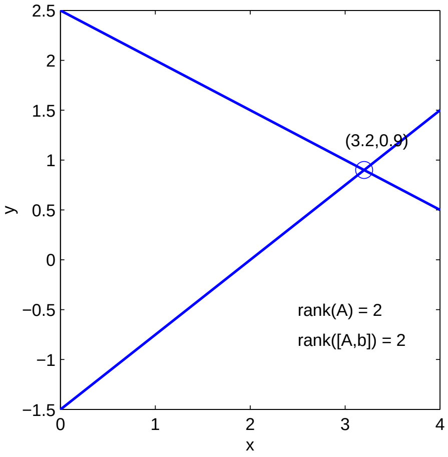


Figure 18.1: Graphical representation of a 2 × 2 determined system.

18.3.3 Underdetermined systems

In this case, the system does not contain enough information to solve for the unknowns, usually because there are fewer equations than unknowns ($m < n$) or because some of the equations are linearly dependent. We have two subcases.

Underdetermined systems with no solution

This happens when

$$\text{rank}(A) < \text{rank}([A, b]) = n.$$

Example. Consider the following 2 × 2 system of equations:

$$\begin{pmatrix} 1 & -2 \\ 2 & -4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}.$$

Here

$$A = \begin{pmatrix} 1 & -2 \\ 2 & -4 \end{pmatrix}, \quad [A, b] = \begin{pmatrix} 1 & -2 & 5 \\ 2 & -4 & 6 \end{pmatrix}$$

Each equation represents a straight line in the $x - y$ plane. The first line is given by $y = (x - 5)/2$, and the second one is $y = (x - 3)/2$. These lines are parallel and never intersect (**Figure 18.2**).

Underdetermined systems with infinite solutions

This happens when

$$\text{rank}(A) = \text{rank}([A, b]) < n.$$

Example. Consider the following 2 × 2 system of equations:

$$\begin{pmatrix} 1 & -2 \\ 2 & -4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \end{pmatrix}.$$

Both equations represent the same line in the $x - y$ plane. Any point on that line is a solution of the system (**Figure 18.3**).

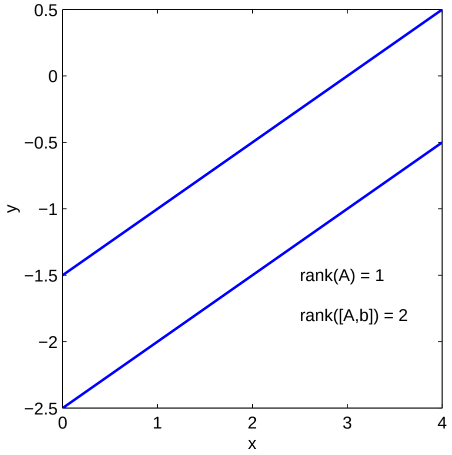


Figure 18.2: A 2 × 2 underdetermined system with no solution.

18.3.4 Overdetermined systems

The system has more independent equations than unknowns ($m > n$). The precise condition is

$$n = \text{rank}(A) < \text{rank}([A, b]).$$

A solution does not exist.

Example. Consider the following 3 × 2 system of equations:

$$\begin{pmatrix} 1 & 2 \\ 3 & -4 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 8 \end{pmatrix}.$$

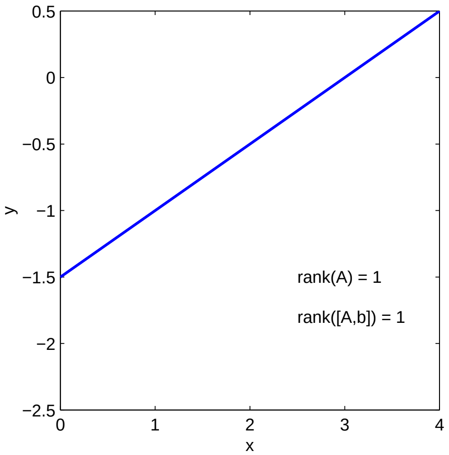


Figure 18.3: A 2 × 2 underdetermined system with infinite solutions.

Each equation represents a straight line in the $x - y$ plane. There is not a point (x, y) at which all three lines intersect each other (**Figure 18.4**).

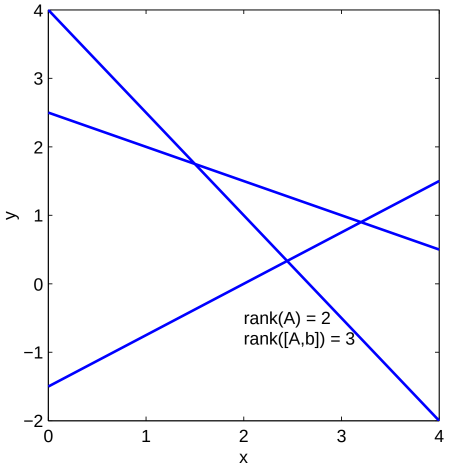


Figure 18.4: Graphical representation of a 3 × 2 overdetermined system.

18.3.5 Solving systems of equations with MATLAB

We now concentrate on the usual case of a system with square ($m = n$) and nonsingular ($\det(A) \neq 0$) matrix:

$$\underbrace{A}_{n\times n}\underbrace{x}_{n\times 1} = \underbrace{b}_{n\times 1} .$$

There are two ways to solve the system:

1. Using *Gaussian elimination*: this numerical technique is similar to the way you are used to when solving systems, i.e., elimination and substitution. In MATLAB this is done with the *left division* operator (\):

   ```
   >> x = A\b
   ```

   This is a very efficient method for systems of moderate size.

2. Using the *inverse matrix*: The inverse of a nonsingular square matrix A is a matrix A^{-1} such that

   $$A^{-1}A = AA^{-1} = \boldsymbol{I} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & 1 \end{pmatrix},$$

   where $\boldsymbol{I}$ is the identity matrix. Therefore, we can write

   $$Ax = b \quad \longrightarrow \quad x = A^{-1}b.$$

   This is done in MATLAB using the *matrix inverse* function (**inv**):

   ```
   >> x = inv(A)*b
   ```

   This operation is computationally much more expensive than Gaussian elimination and is never used in practice (just for academic purposes and very small systems).

Example: We want to solve for the currents i_1 through i_5 in the circuit below (**Figure 18.5**) given the values of the resistances R_1 to R_5 and the two applied voltages V_1, V_2.

Applying Kirchhoff's voltage law for each loop and charge conservation at the nodes, we end up with the following system of equations

$$\begin{array}{cccccl} R_1 i_1 & & & +R_4 i_4 & & = V_1 \\ & R_2 i_2 & & -R_4 i_4 & +R_5 i_5 & = 0 \\ & & -R_3 i_3 & & +R_5 i_5 & = V_2 \\ i_1 & -i_2 & & -i_4 & & = 0 \\ & i_2 & -i_3 & & -i_5 & = 0 \end{array}$$

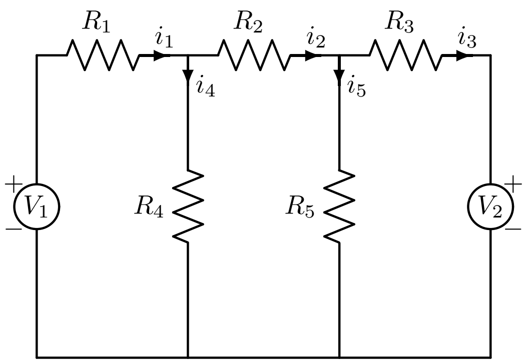


Figure 18.5: Electric circuit.

or in matrix-vector form

$$\begin{pmatrix} R_1 & 0 & 0 & R_4 & 0 \\ 0 & R_2 & 0 & -R_4 & R_5 \\ 0 & 0 & -R_3 & 0 & R_5 \\ 1 & -1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \end{pmatrix} = \begin{pmatrix} V_1 \\ 0 \\ V_2 \\ 0 \\ 0 \end{pmatrix}$$

Below we include the MATLAB script file to calculate the currents in this electric circuit.

```
%
% Simple script file to solve an
% electric circuit
%
% Reference: Palm III, page 328
%
% Author: Ruben Juanes
% Last revised: 3/8/01

clear all
close all

% Data
R1 = 5; R2 = 100; R3 = 200; R4 = 150; R5 = 250;
V1 = 100; V2 = 50;

% Set up matrix-vector
A = [R1, 0, 0, R4, 0;...
      0,R2, 0,-R4,R5;...
      0, 0,-R3, 0,R5;...
      1,-1, 0, -1, 0;...
      0, 1,-1, 0, -1];
```

```
b = [V1; 0; V2; 0; 0];
x = A\b;
disp('currents = ')
disp(x)
```

And here is the output:

```
>> circuit
currents =
    0.9544
    0.3195
    0.0664
    0.6349
    0.2531
```

Chapter 19

Least Squares Fit to a Straight Line

19.1 What Are You Going To Learn?

You will learn how to fit many data points to a simple model, a straight line. In Chapter 20 you will learn how to fit your data to a parabola, or some other polynomial of degree much less than the number of data points.

In Chapter 19 folder on fttp:\\petroleum.berkeley.edu, you will find the following MATLAB files:

cigarettes.m generates `Sample`, a structure containing the following fields: Name, Weight (g), Nicotine (mg) , Tar (mg) and CO (mg). The data presented there are taken from Mendenhall and Sincich.

CigDemo.m loads the cigarette data into the `Sample` structure, performs rudimentary statistics and (covariances and correlation coefficients) and plots the results using `plotcig`.

plotcig.m generates standard plots for the cigarette data.

LLL.m performs the weighted least square fit of data with a linear model $y = a + bx$.

LSDemo.m Generates examples of least square fits using `LLL` and the MATLAB standard `polyfit`, and plots the results.

If you are interested in learning more about statistics, please consult my all-time favorite by Mandel, *The Statistical Analysis of Experimental Data* [22]. My daughter's AP Statistics handbook, *Introduction to the Practice of Statistics* [24], is also a good choice. You can learn a lot about regression from the *Applied Regression Analysis* [9].

19.2 Why Is It Important?

Electronic sensors and computers flood[1] the engineers and scientists with "information" that is not readily useful. We measure ever more parameters of every conceivable natural or industrial process at ever-shorter intervals. As a result, the raw numbers inundate us, and often we cannot grasp their meaning. What we really want is to distill the ugly numbers down to a few simple relationships hidden in them. This is no simple task.

In particular, the statistical method of *least squares* can be used to examine vast data sets and to draw meaningful conclusions about the dependencies that may exist among the various *response variables* and *explanatory variables* in these sets. This method of analysis is called regression[2].

19.3 Useful Definitions

Definition 20 Random variable is a variable whose numerical value is an outcome of a random process. A measurement whose outcome is subject to random error is a random variable. □

Definition 21 Mean, Variance, Standard Deviation: Let $y_1, y_2, y_3, \ldots, y_N$ represent a *(random) sample* of size N from any *population*[3]. The following formulæ express the *sample* estimates of the *population* mean, $\tilde{\mu}$, and the variance, σ^2, respectively:

$$\begin{aligned}\text{Estimate of } \tilde{\mu} = \mu &= \frac{y_1 + y_2 + \cdots + y_N}{N} = \frac{\sum_{i=1}^N y_i}{N} \\ \text{Estimate of } \sigma^2 = s^2 &= \frac{\sum_{i=1}^N (y_i - \mu)^2}{N-1}\end{aligned} \tag{19.1}$$

The sample estimate of a standard deviation, σ, generally denoted by s, is:

$$\text{Estimate of } \sigma = s = \sqrt{\frac{\sum_{i=1}^N (y_i - \mu)^2}{N-1}} \tag{19.2}$$

Note that sample estimates of mean, variance and standard deviation may or may not be representative of the corresponding parameters of the population. More advanced statistics can help you to determine whether a sample is *representative* of a population. □

[1] Recall E77 Introduction and Lecture 1, as well as the book by Roszak [29].

[2] The word "regression" was introduced by an English anthropologist Sir Francis Galton at the turn of the 19th century. Galton studied heredity and observed that the children of tall parents were taller than average, but not as tall as their parents. He called this phenomenon "regression towards mediocrity"; hence the name of the statistical method to be summarized in this lecture.

[3] Go to Mandel [22] for further explanations.

Definition 22 Covariance is a measure of statistical association between two random variables. Let $y_1, y_2, y_3, \ldots, y_N$ and $z_1, z_2, \ldots, z_N$ be two random variables[4]. Formally, the covariance is defined as the mean of the product of $(y-\mu_y)$ and $(z-\mu_z)$:

$$\mathrm{Cov}(y,z) = \frac{\sum_{i=1}^{N}(y_i-\mu_y)(z_i-\mu_z)}{N-1} \tag{19.3}$$

Note that the covariance pertains to *populations*, but its value is estimated from *samples*. □

Definition 23 The coefficient of correlation of random samples y and z is defined as

$$r(y,z) = \frac{\mathrm{Cov}(y,z)}{s_y s_z} \tag{19.4}$$

By using $z = y$ and $z = -y$, one can show that the correlation coefficient is always between -1 and 1. □

Let us now look[5] at a *sample* of the *population* of all cigarettes sold in the U.S., **Table 19.1**. The means, variances and standard deviations of the measurements included in the sample are listed in Table 19.1, and the respective covariances and correlation coefficients are listed in **Table 19.2**.

The tar content is positively correlated with carbon monoxide (CO) produced and nicotine content. The nicotine content is also positively correlated with CO produced. The respective correlation coefficients are above 0.92, indicating significant positive correlation. The cigarette weight and CO produced are also positively correlated, but the correlation is poor ($r = 0.46$). The results are plotted in **Figure 19.1**.

In the example above, we have divided the sample into two types of variables. In each plot, a single *independent* or *explanatory* variable (tar content, CO produced, or cigarette weight) was used to fit a single *dependent* or *response* variable (nicotine content, CO produced or tar content).

The independent variable can either be set to a desired value (e.g., cigarette weight or tar content) or else take values that can be measured but cannot be controlled (e.g., CO emissions). As a result of changes that are deliberately made, or just take place in the independent variable, the dependent variable takes on different values. Here we want to find out if a single dependent variable follows a simple model function of a single independent variable or is unaffected by the changes of the independent variable:

Response variable =

Model function of explanatory variable + Random error

[4] For example, y_i and z_i are the measurements of two different parameters of a process sampled at discrete times $t = t_i$, $i = 1, 2, \ldots, N$. Each measurement is subject to random error.

[5] This data set was downloaded through the Internet from the following source: gopher://jse.stat.ncsu.edu/11/jse under the folder `JSE Dataset Archive`. The data presented there is taken from MENDENHALL and SINCICH [23] and is a subset of the data produced by the Federal Trade Commission. It was submitted by LAUREN MCINTYRE, Department of Statistics, North Carolina State University.

Table 19.1: Tar and nicotine contents, weight, and carbon monoxide produced by the popular cigarettes.

Brand	Tar (mg)	Nicotine (mg)	Weight (g)	CO (mg)
Alpine	14.1	0.86	0.9853	13.6
Benson & Hedges	16.0	1.06	1.0938	16.6
Bull Durham	29.8	2.03	1.1650	23.5
Camel Lights	8.0	0.67	0.9280	10.2
Carlton	4.1	0.40	0.9462	5.4
Chesterfield	15.0	1.04	0.8885	15.0
Golden Lights	8.8	0.76	1.0267	9.0
Kent	12.4	0.95	0.9225	12.3
Kool	16.6	1.12	0.9372	16.3
L & M	14.9	1.02	0.8858	15.4
Lark Lights	13.7	1.01	0.9643	13.0
Marlboro	15.1	0.90	0.9316	14.4
Merit	7.8	0.57	0.9705	10.0
Multi Filter	11.4	0.78	1.1240	10.2
Newport Lights	9.0	0.74	0.8517	9.5
Now	1.0	0.13	0.7851	1.5
Old Gold	17.0	1.26	0.9186	18.5
Pall Mall Light	12.8	1.08	1.0395	12.6
Raleigh	15.8	0.96	0.9573	17.5
Salem Ultra	4.5	0.42	0.9106	4.9
Tareyton	14.5	1.01	1.0070	15.9
TRUE	7.3	0.61	0.9806	8.5
Viceroy Rich Light	8.6	0.69	0.9693	10.6
Virginia Slims	15.2	1.02	0.9496	13.9
Winston Lights	12.0	0.82	1.1184	14.9
Mean, μ	12.2	0.88	0.9703	12.5
Variance, s^2	32.1	0.13	0.0077	22.5
Standard Deviation, s	5.7	0.35	0.0877	4.7

Table 19.2: Covariances and correlation coefficients

	Nic. vs Tar.	Nic. vs CO	CO vs Weight	Tar vs CO
Cov	1.959	1.554	0.1929	25.71
r	0.9766	0.9259	0.4640	0.9574

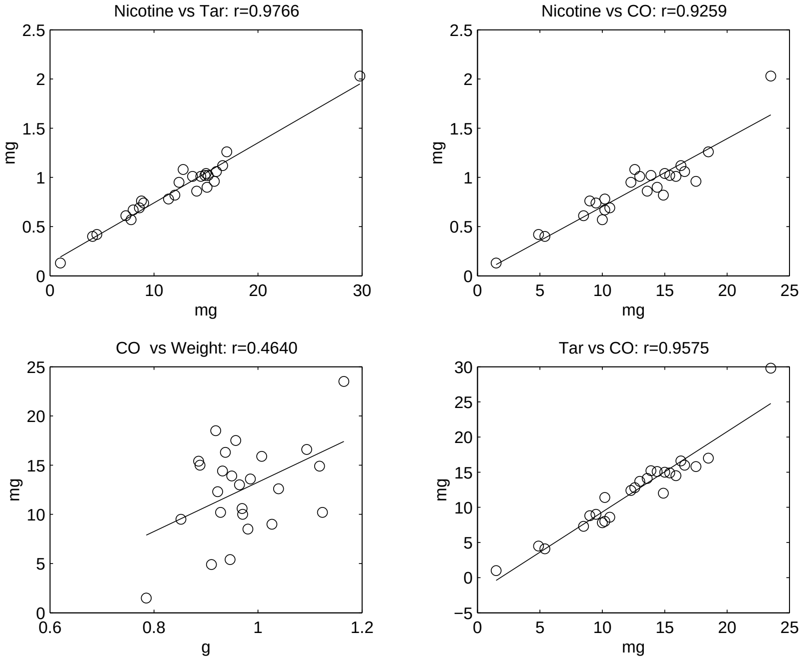


Figure 19.1: Least square fits of the cigarette data in Table 19.1

The model function is assumed to be *known* (in practice we can test several simple functions and pick the one that works best, see Lecture 16). The distribution of the random errors is usually assumed to be *normal* (it follows a bell curve) with zero mean and a standard deviation defined by the uncertainty of a given measurement, i.e., the individual measurements are assumed to be *unbiased* and their errors are assumed to be *independent*.

Remark 21 The least square fit *does not* honor the individual data points, but only their average trend. □

19.4 Fitting Data to a Straight Line

This section follows closely Section 15.2 in the *Numerical Recipes in C* [28] . Suppose that we have N measurements of a *response variable* (e.g., nicotine content), $\{y_i\}$, at discrete values of an *explanatory variable* (e.g., tar content or time), $\{x_i\}$, where N is very large. At first, we want to consider the simplest possible model of our data, a straight line:

$$y(x) = a_0 + a_1 x = y(x; a_0, a_1) \tag{19.5}$$

This problem is called *linear regression*. We assume that the uncertainty (noise), σ_i, associated with each measurement y_i is known[6], and that we know exactly each value of x_i.

To measure how well our model agrees with the data, we use a simple merit function

$$\chi^2(a_0, a_1) = \sum_{i=1}^{N} \left(\frac{y_i - a_0 - a_1 x_i}{\sigma_i} \right)^2 \tag{19.6}$$

This particular function is called in statistics the *chi-square* function or the *goodness-of-fit* function.

Equation (19.6) is minimized to obtain a_0 and a_1. At its minimum the derivatives of $\chi^2(a_0, a_1)$ with respect to a_0 and a_1 vanish:

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_0} &= -2 \sum_{i=1}^{N} \frac{y_i - a_0 - a_1 x_i}{\sigma_i^2} = 0 \\ \frac{\partial \chi^2}{\partial a_1} &= -2 \sum_{i=1}^{N} \frac{x_i(y_i - a_0 - a_1 x_i)}{\sigma_i^2} = 0 \end{aligned} \tag{19.7}$$

Equations (19.7) can be rearranged in a more convenient form if we define the following parameters:

$$\begin{aligned} S \equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2}, \quad S_x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2}, \quad S_y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ S_{xx} \equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2}, \quad S_{xy} \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{19.8}$$

With definitions (19.8), Eqs. (19.7) become

$$\begin{aligned} a_0 S + a_1 S_x &= S_y \\ a_0 S_x + a_1 S_{xx} &= S_{xy} \end{aligned} \tag{19.9}$$

The solution of these two linear equations in a_0 and a_1 is simply

$$\begin{aligned} D &\equiv S S_{xx} - (S_x)^2 \\ a_0 &= \frac{S_{xx} S_y - S_x S_{xy}}{D} \\ a_1 &= \frac{S_{xy} S - S_x S_y}{D} \end{aligned} \tag{19.10}$$

Equation (19.10) gives the solution for the best-fit model parameters a_0 and a_1.

[6] In many texts, these uncertainties are taken to be equal and scaled to one, i.e., all $\sigma_i = 1$.

19.5 Normal Equations

The individual equations in the merit function (19.6) can be written as

$$\frac{1}{\sigma_i}a_0 + \frac{x_i}{\sigma_i}a_1 = \frac{y_i}{\sigma_i}, \quad i = 1, 2, \ldots, N \tag{19.11}$$

or in matrix-vector form

$$\underbrace{\begin{pmatrix} 1/\sigma_1 & x_1/\sigma_1 \\ 1/\sigma_2 & x_2/\sigma_2 \\ \vdots & \vdots \\ 1/\sigma_N & x_N/\sigma_N \end{pmatrix}}_{(N\times 2)\ X} \underbrace{\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}}_{(2\times 1)\ a} = \underbrace{\begin{pmatrix} y_1/\sigma_1 \\ y_2/\sigma_2 \\ \vdots \\ y_N/\sigma_N \end{pmatrix}}_{(N\times 1)\ y} \tag{19.12}$$

$$Xa = y$$

Remark 22 The system of linear equations (19.12) has 2 unknowns and $N \gg 2$ equations. This system is overdetermined and it *cannot* be solved for a_0 and a_1. However, the "best" values of a_0 and a_1 *can* be found that minimize the sum of squares of the deviations between the straight line fit and the data; hence the name "least square fit." □

In matrix-vector form, the 2×2 system of equations (19.9) is

$$\underbrace{\begin{pmatrix} \sum_{i=1}^{N} \frac{1}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \end{pmatrix}}_{(2\times 2)\ A} \underbrace{\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}}_{(2\times 1)\ a} = \underbrace{\begin{pmatrix} \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \end{pmatrix}}_{(2\times 1)\ b}.$$

Here $\boldsymbol{a} = [a_0, a_1]^T$ is the vector of the unknown fit parameters. These equations can also be written in a more familiar form by noting that:

$$\underbrace{A}_{(2\times 2)} = \underbrace{\begin{pmatrix} 1/\sigma_1 & 1/\sigma_2 & \ldots & 1/\sigma_N \\ x_1/\sigma_1 & x_2/\sigma_2 & \ldots & x_N/\sigma_N \end{pmatrix}}_{(2\times N)} \underbrace{\begin{pmatrix} 1/\sigma_1 & x_1/\sigma_1 \\ 1/\sigma_2 & x_2/\sigma_2 \\ \vdots & \vdots \\ 1/\sigma_N & x_N/\sigma_N \end{pmatrix}}_{(N\times 2)} = X^T X,$$

$$\underbrace{b}_{(2\times 1)} = \underbrace{\begin{pmatrix} 1/\sigma_1 & 1/\sigma_2 & \ldots & 1/\sigma_N \\ x_1/\sigma_1 & x_2/\sigma_2 & \ldots & x_n/\sigma_N \end{pmatrix}}_{(2\times N)} \underbrace{\begin{pmatrix} y_1/\sigma_1 \\ y_2/\sigma_2 \\ \vdots \\ y_N/\sigma_N \end{pmatrix}}_{(N\times 1)} = X^T y,$$

so we get the final form of the *Normal Equations*:

$$\boxed{\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{a} = \boldsymbol{X}^T \boldsymbol{y}.} \tag{19.13}$$

Remark 23 The reason for multiplying both sides of the overdetermined system $Xa = y$ by X^T is to minimize the sum of the errors squared (thus the name *least squares*), not just to get a determined system (the latter task could be achieved with infinitely many different $2 \times N$ arrays). □

19.6 Analysis of Variance

We are not done yet. We must still estimate the following:

1. How much of the variation in the dependent variable data has been explained by the regression line. (From Figure 19.1 we see that the CO emissions are not explained well by the "best" linear function of the cigarette weight. On the other hand, the nicotine content depends linearly on the tar content.)

2. What are the probable uncertainties in the estimates of a_0 and a_1. (Obviously the measurement errors in the data must introduce uncertainty in the determination of the model parameters. If the measurements are independent, then each contributes its own bit of uncertainty to the parameters.)

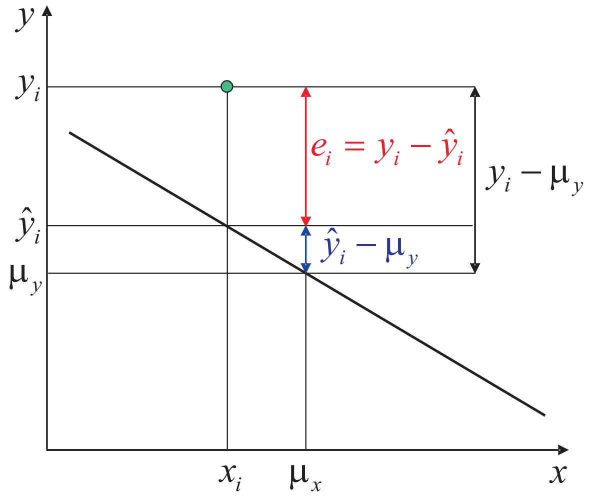


Figure 19.2: Geometry of identity (19.15).

19.6.1 How Good is the Correlation?

Let us denote the least square prediction of the value of y_i as $\hat{y}_i$, i.e.,

$$\hat{y}_i = a_0 + a_1 x_i \tag{19.14}$$

Then the following identity holds:

$$(y_i - \mu_y) = (y_i - \hat{y}_i) + (\hat{y}_i - \mu_y). \tag{19.15}$$

Geometrically, Eq. (19.15) means that the deviation of a measured value, y_i, from the mean of the measured values, μ_y, is the sum of the *residual* or *error of approximation* of the measured value, $e_i := y_i - \hat{y}_i$, and of the deviation of the the approximate value, $\hat{y}_i$, from the mean, μ_y, **Figure 19.2**.

If we square both sides of Eq. (19.15) for each i and add all terms $i = 1, 2, \ldots, N$, we obtain (See Exercise 4):

$$\sum_{i=1}^{N}(y_i - \mu_y)^2 = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \sum_{i=1}^{N}(\hat{y}_i - \mu_y)^2 \tag{19.16}$$

In words, Eq. (19.16) can be explained as follows:

$$\begin{pmatrix}\text{Sum of squares}\\ \text{about the mean}\end{pmatrix} = \begin{pmatrix}\text{Sum of squares}\\ \text{about the LS fit}\end{pmatrix} + \begin{pmatrix}\text{Sum of squares}\\ \text{due to LS fit}\end{pmatrix} \tag{19.17}$$

Now it is clear the some of the variation of the y's about their mean can be ascribed to the regression line, and some, $\sum(y_i - \hat{y}_i)^2$, to the sad fact that the actual measurements do not all follow a straight line. If they all did, then the sum of squares about the regression (also called the *residual sum of squares*) would be zero and the correlation would be perfect! The imperfection of the correlation is measured by the R^2 statistic:

$$R^2 = \frac{\text{SS due to LS fit given } a_0}{\text{Total SS, corrected for } \mu_y} = \frac{\sum_{i=1}^{N}(\hat{y}_i - \mu_y)^2}{\sum_{i=1}^{N}(y_i - \mu_y)^2} \tag{19.18}$$

Therefore R^2 measures the proportion of the total variation about the mean, μ_y, explained by the least squares fit.

19.6.2 What Are the Uncertainties in a and b?

The analysis of variance of a and b is too involved to be followed here, but the result is [9, 28]:

$$\begin{aligned}\sigma_{a_0}^2 &= \sum_{i=1}^{N}\left(\frac{S_{xx} - S_x x_i}{D\sigma_i}\right)^2 = \frac{S_{xx}^2 S - 2S_{xx}(S_x)^2 + (S_x)^2 S_{xx}}{D^2} = \frac{S_{xx}}{D}\\ \sigma_{a_1}^2 &= \sum_{i=1}^{N}\left(\frac{Sx_i - S_x}{D\sigma_i}\right)^2 = \frac{S^2 S_{xx}^2 - 2S(S_x)^2 + (S_x)^2 S}{D^2} = \frac{S}{D}\end{aligned} \tag{19.19}$$

These are the variances in a_0 and a_1, respectively. We will need one more number to characterize the probable uncertainty of the parameter estimation. That number is the *covariance* of a_0 and a_1, and it is given by

$$\text{Cov}(a_0, a_1) = -\frac{S_x}{D} \tag{19.20}$$

Usually, the covariance is normalized to obtain the *coefficient of correlation* between the uncertainty in a_0 and the uncertainty in a_1:

$$r_{a_0 a_1} = \frac{\mathrm{Cov}(a_0, a_1)}{\sigma_{a_0}\sigma_{a_1}} = -\frac{S_x}{\sqrt{SS_{xx}}} \tag{19.21}$$

19.7 Better Least Squares Equations

Formulæ (19.10) are susceptible to roundoff error. A better way of writing them is:

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \quad i = 1, 2, \ldots, N \tag{19.22}$$

and

$$S_{tt} = \sum_{i=1}^{N} t_i \tag{19.23}$$

Then (please verify by substitution)

$$\begin{aligned} a_1 &= \frac{1}{S_{tt}}\sum_{i=1}^{N}\frac{t_i y_i}{\sigma_i} \\ a_0 &= \frac{S_y - S_x a_1}{S} \end{aligned} \tag{19.24}$$

In addition, the uncertainties in a_0 and a_1 can be estimated as:

$$\begin{aligned} \sigma_{a_0}^2 &= \frac{1}{S}\left(1 + \frac{S_x^2}{SS_{tt}}\right) \\ \sigma_{a_1}^2 &= \frac{1}{S_{tt}} \\ \mathrm{Cov}(a_0, a_1) &= -\frac{S_x}{SS_{tt}} \\ r_{a_0 a_1} &= \frac{\mathrm{Cov}(a_0, a_1)}{\sigma_{a_0}\sigma_{a_1}} \end{aligned} \tag{19.25}$$

19.8 Example of Least Square Fit

The code is listed in Section 19.10 and 19.11. The function to be fit with least squares is as follows:

$$\begin{aligned} \sigma_i &= 10(\mathrm{rand} + 10^{-5}) \\ e_i &= \text{Random Number from ND}(\mu = 0, \sigma = \sigma_i) \\ y_i &= -2.125 x_i + 4 + e_i, \\ i &= 1, 2, \ldots, N \end{aligned} \tag{19.26}$$

where rand is the uniform random distribution function, and ND is the normal distribution probability function. The results are shown in **Figures 19.3** and **19.4**.

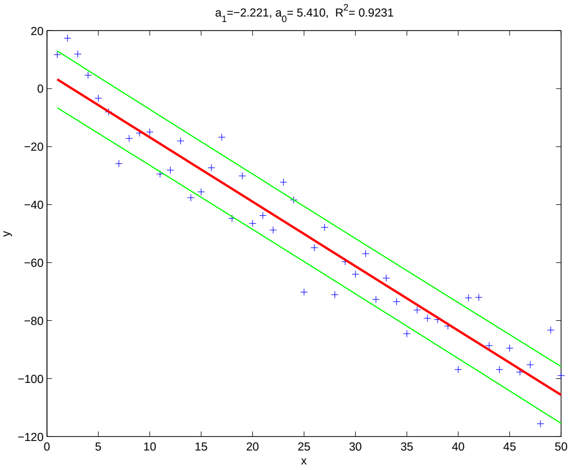


Figure 19.3: Unweighted least square fit using MATLAB's `polyfit` and `polyval` functions. The green lines are bounds on at least 50% of the data points being closer to the fit.

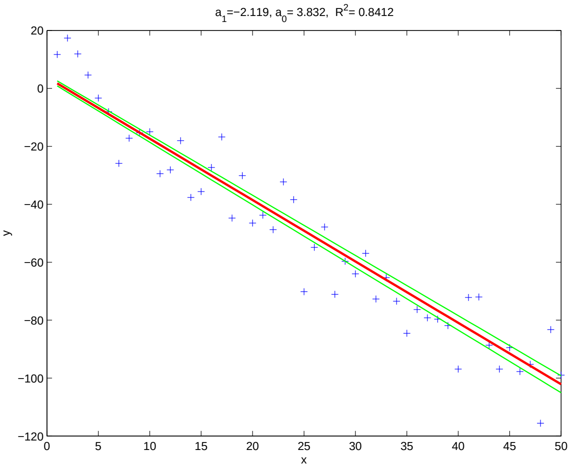


Figure 19.4: Weighted least square fit of data in Figure 1, using the `LLL.m` function. Note that the fit is closer to the main linear trade in the data than the MATLAB fit. This is because the outliers are weighted less. The two green lines are fit with plus and minus 3 standard deviations in the slope and intercept.

19.9 Exercises

1. Download `cigarette.m` and calculate the sample statistics using MATLAB.

2. Perform the least square linear fits of the data using `polyfit`, and plot the results using `polyval`.

3. Show that the sum of errors of the least squares approximation is zero:

$$\sum_{i=1}^{N} e_i \equiv \sum_{i=1}^{N} (y_i - \hat{y}_i) = 0$$

4. **Bonus:** Show that the cross-product term, $\sum_{i=1}^{N}(\hat{y}_i - \mu_y)(y_i - \hat{y}_i)$, in Eq. (19.16) vanishes. **Hint:** Use the equation for a_1, and the following identities, $(\hat{y}_i - \mu_y) = a_1(x_i - \mu_x)$, and $y_i - \hat{y}_i = y_i - \mu_y - a_1(x_i - \mu_x)$.

19.10 Least Square Fit Demo

```
function LSDemo()
% Generates examples of least square fits using LLL and
% MATLAB standard polyfit and plots the results.
% NOTE: The Class Reader fit almost always does better than polyfit!
%
% E77N Class Material, T. W. Patzek, 2002-10-27
clear all,close all,clc
PrintFlag =0;
FontSize=12;
if ~PrintFlag
    FontSize=14;
end
%Number of measurements
N = 50;
x = 1:1:N;
% Specify measurement error
Sigma = 10*(rand(1,N)+1e-05);
Mean = 0;
%---------------------------------------------------------------------------
% Normal error distribution with zero mean and
% a random STD about each y_i
%---------------------------------------------------------------------------
Error=Mean+Sigma.*randn(1,N);
y = -2.125*x + 4 + Error;
fprintf(’\n\nTRUE:      a_1=-2.125, a_0= 4.000  (plus random error)\n’);

%---------------------------------------------------------------------------
% MATLAB’s least square fit<<<<<<<<<<<<<<<<<<<<<<
%---------------------------------------------------------------------------
[p,S] = polyfit(x,y,1);
% Use polyval to evaluate the polynomials<<<<<<<<<<<<<<<<
[yLS,delta] = polyval(p,x,S);
% Obtain R^2 statistic
ybar = mean(y);
RSquare = sum((yLS-ybar).^2)/sum((y-ybar).^2);

figure(1)
hold on
plot(x,y,’+’)
plot(x,yLS,’color’,’r’,’linewidth’,2);
plot(x,yLS+delta,’color’,’g’,’linewidth’,1);
plot(x,yLS-delta,’color’,’g’,’linewidth’,1);
set(gca,’FontSize’,FontSize);
s=sprintf(’a_1=%6.3f, a_0=%6.3f,  R^2=%7.4f’,...
    p(1),p(2),RSquare);
fprintf(’MATLAB:  %s\n’,s);
title(s), xlabel(’x’), ylabel(’y’), box on
if PrintFlag
```

```
    print(’-dpsc2’, ’E77N-Figure15.3.eps’);
end
%---------------------------------------------------------------------------
% Class Reader LS Fit
%---------------------------------------------------------------------------
[a,b,SigmaA, SigmaB, RabSquare,RSquare]=LLL(x,y,Sigma);
p2=[b,a];
[yLS2] = polyval(p2,x);
p3=[b+3*SigmaB,a+3*SigmaA];
[yLS3] = polyval(p3,x);
p4=[b-3*SigmaB,a-3*SigmaA];
[yLS4] = polyval(p4,x);

figure(2)
plot(x,y,’+’)
hold on
set(gca,’FontSize’,FontSize);
plot(x,yLS2,’color’,’r’,’linewidth’,2);
%Plot maximum deviation lines
plot(x,yLS3,’color’,’g’,’linewidth’,1);
plot(x,yLS4,’color’,’g’,’linewidth’,1);
s=sprintf(’a_1=%6.3f, a_0=%6.3f,  R^2=%7.4f’,...
    p2(1),p2(2),RSquare);
fprintf(’READER: %s\n’,s);
title(s), box on, xlabel(’x’), ylabel(’y’)
if PrintFlag
    print(’-dpsc2’, ’E77N-Figure15.4.eps’);
end
%Compare polyfit with the Reader solution
[yLS2] = polyval(p2,x);
figure(3)
plot(x,y,’+’)
hold on
set(gca,’FontSize’,FontSize);
plot(x,yLS ,’color’,’b’,’linewidth’,2);
plot(x,yLS2,’color’,’r’,’linewidth’,2);
legend(’Data’,’MATLAB’,’Reader’)
title(’MATLAB vs Class Reader’), box on, xlabel(’x’), ylabel(’y’)
```

19.11 Least Square Fit Algorithm Based on Eqs. (19.22-19.24)

```
 function [a,b,SigmaA, SigmaB, rab, RSquare]=...
    LLL(xs,ys,Sigmas)
%------------------------------------------------------
%  Function LLL performs the least square
%  fit of data with a linear model y=a+b*x
%
```

```
%   Input arguments:
%       xs     = vector of independent variable values
%       ys     = vector of dependent variable values
%       Sigmas = vector of STD's in ys (if you do not
%                know them input a vector of ones)
%   Output arguments:
%       a      = intercept of fit line (=a_0)
%       b      = slope of fit line (=a_1)
%       SigmaA = standard deviation of a
%       SigmaB = standard deviation of b
%       rab    = correlation coefficient between a and b
%       RSquare= R^2 statistic of correlation
%
% Written by T. W. Patzek, March 07, 2001
% UC Berkeley, 437 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834 or 510-486-5322,
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/26/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.26 17:00:00 $
%-----------------------------------------------------
%                    PRELIMINARIES
if     nargin < 1
    error('You must input vectors of xs, ys and sigmas')
elseif nargin < 2
    error('You must input vectors of ys and sigmas')
elseif nargin < 3
    error('You must input vector of sigmas')
end

% Check for consistency
% Force all inputs to be column vectors
xs     = (xs(:)')';
ys     = (ys(:)')';
Sigmas = (Sigmas(:)')';
llx    = length(xs);
lly    = length(ys);
lls    = length(Sigmas);
if (llx~=lly)
    error('Vectors x and y must have the same length');
end
N=llx;
if (lls ~= N)
    error('Sigma must have the same length as x and y');
end
%-----------------------------------------------------
%              START THE CODE
%         Class Reader Section 19.7
%-----------------------------------------------------
S  = sum(1./(Sigmas.^2));
```

```
Sx = sum(xs./(Sigmas.^2));
Sy = sum(ys./(Sigmas.^2));
ts = (xs-Sx/S)./Sigmas;

Stt = sum(ts.*ts);
b   = sum((ts.*ys)./Sigmas)/Stt;
a   = (Sy-Sx*b)/S;
SigmaA = sqrt((1+Sx^2/(S*Stt))/S);
SigmaB = sqrt(1/Stt);
CovAB  = -Sx/(S*Stt);
rab    = CovAB/(SigmaA*SigmaB);
ybar = mean(ys);
RSquare = sum((a+b*xs-ybar).^2)/sum((ys-ybar).^2);
if (RSquare > 1) RSquare = 1; end;
```

Chapter 20

Least Squares Fit to a Polynomial

20.1 What Are You Going To Learn?

You will learn how to fit a very large number of data points, N, to a polynomial of degree much less than the number of data points minus one. This approach is called *general linear least square fit.*

In Chapter 20 folder on fttp:\\petroleum.berkeley.edu look up the following MATLAB files:

GLLS.m performs least squares fit of data with a nonlinear model $y = a_1 + a_2x + a_3x^2 + \ldots$.

BasisFunctions.m employs monomials of x as basis functions for the least square fit. Any other set of basis functions could be used.

GLLSDemo.m Generates examples of least square fits using `GLLS` and the MATLAB standard `polyfit`, and plots the results.

20.2 Why Is It Important?

If the dependent, or response variable does not follow a linear trend, you should try a more complicated *model function* of the independent, or explanatory variable. In this lecture, the model function will be a sum of monomials of degrees 0 through $M - 1 \ll N$ of the independent variable. Our task is to find the "best" coefficient of each monomial, $a_0, a_1, a_2, \ldots, a_{M-1}$, thus obtaining the best polynomial fit of the data. The model function is linear in the fit coefficients (hence the term *linear*), and involves monomial powers greater than one (hence the term *general*).

20.3 General Linear Least Squares

In the last lecture you learned how to fit data to a straight line. Here we shall learn how the simplest linear least squares model can be generalized. This section follows closely Section 15.4 in the *Numerical Recipes in C* [28].

Again, suppose that we have N measurements $\{y_i\}$ at discrete values of the independent variable $\{x_i\}$, where N is very large. Now we want to fit our data not just to a linear combination of $x^0 = 1$ and x^1 (namely $a_0 + a_1 x$), but rather to a linear combination of any M specified functions of x. For example, we might choose as the fitting functions 1, x, x^2, $\ldots$, x^{M-1}, in which case their general linear combination is a polynomial of degree $M-1$. For convenience, we will start our summation index from *one*, not from *zero* as in Lecture 14:

$$y(x) = a_1 + a_2 x + a_3 x^2 + ... + a_M x^{M-1} = y(x; a_1, a_2, ...a_M) \tag{20.1}$$

Alternatively, the functions could be *sines* and *cosines*, in which case their general combination would be a *harmonic series.*

The general form of this model of data is

$$y(x) = \sum_{i=1}^{M} a_i X_i(x) \tag{20.2}$$

where $X_1(x), ..., X_M(x)$ are arbitrary, fixed, but independent functions of x, called the *basis functions*; for example, $X_1(x) = 1$, $X_2(x) = x$, $\ldots$, $X_M(x) = x^{M-1}$.

Remark 24 The functions $X_i(x)$ can be wildly nonlinear. In our context, the word “linear” refers only to the linear combination of the basis functions, using the parameters a_i. This method is called the general linear least squares fit. □

As before, we assume that the uncertainty (noise) σ_i associated with each measurement y_i is known, and that we know exactly each value of the independent variable x_i.

To measure how well our model agrees with the data, we use the same merit function:

$$\chi^2(a_1, ...a_M) = \sum_{i=1}^{N} \left(\frac{y_i - \sum\limits_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right)^2 \tag{20.3}$$

As before, this particular function is called the *chi-square* function or the *goodness-of-fit* function.

Once again, we shall pick as best parameters those values of $a_1, a_2, ..., a_M$ that minimize Eq. (20.3). Several different methods are used to find this minimum. Here we shall discuss only one of the two particularly useful, the normal equations method. The second, very best method, called the singular value decomposition (SVD), is beyond the scope of this course. To get started, we need to define notation.

Let A be an array whose $N \times M$ elements are constructed from the M basis functions evaluated at the N abscissas, x_i, and from the N measurement errors, σ_i:

$$A_{ij} \equiv \frac{X_j(x_i)}{\sigma_i}, \quad i = 1, 2, ..., N, \quad j = 1, 2, ..., M \tag{20.4}$$

The array A is called the *design matrix* of the fitting problem. Observe that in general A has more rows than columns, $N \gg M$, because there should be many more data points than parameters.

$$\begin{array}{rc} & \longleftarrow \qquad \text{basis functions} \qquad \longrightarrow \\ A = & \begin{pmatrix} \frac{X_1(x_1)}{\sigma_1} & \frac{X_2(x_1)}{\sigma_1} & \dots & \frac{X_M(x_1)}{\sigma_1} \\ \frac{X_1(x_2)}{\sigma_2} & \frac{X_2(x_2)}{\sigma_2} & \dots & \frac{X_M(x_2)}{\sigma_2} \\ \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \dots & \vdots \\ \frac{X_1(x_N)}{\sigma_N} & \frac{X_2(x_N)}{\sigma_N} & \dots & \frac{X_M(x_N)}{\sigma_N} \end{pmatrix} \end{array} \tag{20.5}$$

We shall also define a column vector $\boldsymbol{y}$ of length N by

$$y^T = \begin{pmatrix} \frac{y_1}{\sigma_1} & \frac{y_2}{\sigma_2} & \cdots & \frac{y_N}{\sigma_N} \end{pmatrix} \tag{20.6}$$

and denote the column vector whose elements are the parameters to be fitted, $a_1, a_2, \dots, a_M$ as a.

20.3.1 Solution With the Normal Equations

At its minimum the derivatives of $\chi^2(a_1, a_2, \dots, a_M)$ with respect to $a_1, a_2, \dots, a_M$ vanish:

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_1} &= -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_1(x_i) = 0 \\ \frac{\partial \chi^2}{\partial a_k} &= -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) = 0, \\ & k = 2, 3, \dots, M \end{aligned} \tag{20.7}$$

Equations (20.7) can be rewritten in a more convenient if we interchange the order of summations, move the vector $\boldsymbol{y}$ to the right hand side, and write them as the matrix equation:

$$\sum_{j=1}^{M} \left\{ \underbrace{\left[\sum_{i=1}^{N} \frac{1}{\sigma_i^2} X_j(x_i) X_k(x_i) \right]}_{(A^T A)_{kj}} a_j \right\} = \underbrace{\sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i X_k(x_i) \right]}_{(A^T y)_k}, \quad k = 1, 2, 3, \ldots, M \tag{20.8}$$

So, finally, the general least square fit problem has been recast into the *Normal Equation* form:

$$\boxed{(\boldsymbol{A}^T \boldsymbol{A})\boldsymbol{a} = \boldsymbol{A}^T \boldsymbol{y}} \tag{20.9}$$

The array $A^T A$ is a square $M \times M$ matrix, multiplying the column vector of M unknowns a. The right hand side is a known column vector of length M. In MATLAB, the system of linear equations (20.9) can be solved easily with the "backslash" operator.

The inverse matrix

$$C = (A^T A)^{-1} \tag{20.10}$$

is closely related to the standard uncertainties in the estimated parameters $\boldsymbol{a}$. In particular, it turns out that

$$\sigma^2(a_j) = C_{jj} \tag{20.11}$$

If we do not want to calculate the uncertainties in the estimated parameters, the "backslash" operator is just fine. Otherwise, we must calculate the inverse matrix, with e.g., `inv(A'A)`, GAUSS-JORDAN elimination, or any other technique.

Remark 25 At this point, you need to be forewarned that the solution of a least-squares problem from the normal equations is susceptible to roundoff error. This and other difficulties in the least squares solutions are resolved when the SVD method is used. Therefore, the SVD method is preferred above all other methods. □

Remark 26 In addition, in practice we need a bookkeeping trick that will allow us to "freeze" certain values of the fitted parameters a_k. We may know these frozen values from some theory or other experiments. This task is accomplished with the vector of freeze flags, each equal to zero or one. The zero flag corresponding to a parameter a_k, signals that the value of this parameter should not be changed by least squares. For a nice implementation, see [28]. □

20.3.2 Examples

Let us consider "data points" generated from the following nonlinear function:

$$\begin{aligned} \sigma_i &= 10(\text{rand} + 10^{-5}) \\ e_i &= \text{Random Number from ND}(\mu = 0, \sigma = \sigma_i) \\ y_i &= 4 - 2x_i + 0.04x_i^2 - 0.001x_i^3 + e_i \\ i &= 1, 2, \ldots, N \end{aligned} \tag{20.12}$$

where rand is the uniform random distribution function, andND is the normal distribution probability function. We approximate these data with a cubic regression model:

$$y = a_1 + a_2 x + a_3 x^2 + a_4 x^3 \tag{20.13}$$

The results are shown in **Figure 20.1** and **20.2**. The code is listed in Section 20.5 and 20.6.

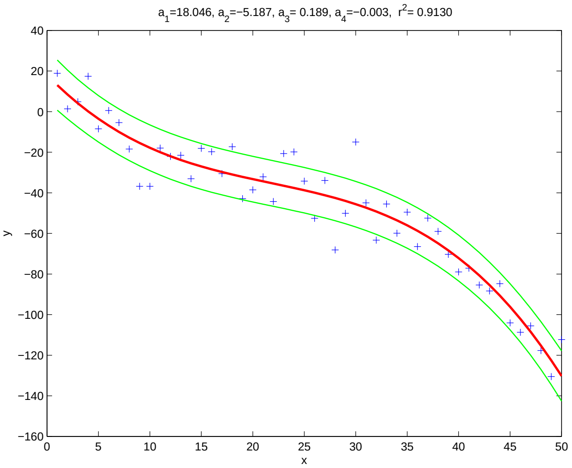


Figure 20.1: Unweighted cubic fit using MATLAB's `polyfit` and `polyval` functions. The green lines are bounds on at least 50% of the data points being closer to the fit.

20.4 Exercises

1. Approximate function (20.12) with the trigonometric basis functions, $X(x_i) = \cos(2(n-1)\pi x_i / L)$, $n = 1, 2, \ldots, M$, $L = \max(x) - \min(x)$. Test a couple of M's.

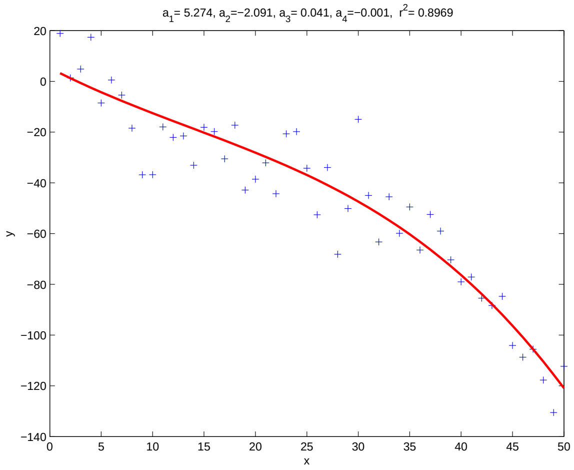


Figure 20.2: Weighted cubic fit using the function GLS.m. The outliers are discounted.

20.5 General Linear Least Square Fit Demo

```
function GLSDemo()
%
% GLLSDEMO = General Linear Least Squares Demo
% uses the nonlinear basis functions (here integer powers of x)
% Calls GLLS and MATLAB polyfit, and plots the results
% NOTE: The Class Reader fit almost always does better than polyfit!
%
% Written by T. W. Patzek, March 10, 2001
% UC Berkeley, 437 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834 or 510-486-5322,
% email: patzek@patzek.berkeley.edu
%
% Last revision 2002-10-27, TWP
% $Revision: 1.1 $  $Date: 2002.10.27 17:00:00 $
%---------------------------------------------------------------
clear all, close all, clc
PrintFlag =0;
FontSize=12;
LineWidth =1;
if ~PrintFlag
    FontSize=14;
    LineWidth =2;
end
%Number of measurements
N = 50;
x = 1:1:N;
% Specify measurement error
```

```
Sigma = 10*(rand(1,N)+1e-05);
Mean = 0;
%---------------------------------------------------------------------------
% Normal error distribution with zero mean and
% a random STD about each y_i
%---------------------------------------------------------------------------
Error=Mean+Sigma.*randn(1,N);
y = -0.001*x.^3+0.04*x.^2-2*x+4 + Error;
fprintf('\n\nTRUE:      a_1= 4.000, a_2=-2.000, a_3= 0.040, a_4=-0.001\n');
%---------------------------------------------------------------------------
% MATLAB's least square fit<<<<<<<<<<<<<<<<<<<<<<<
%---------------------------------------------------------------------------
[p,S] = polyfit(x,y,3);
% Use polyval to evaluate the polynomials!!!
[yLS,delta] = polyval(p,x,S);
%Get the correlation coefficient
R= corrcoef(y,yLS);
rSquare=R(1,2)^2;

figure(1)
plot(x,y,'+')
hold on
set(gca,'FontSize',FontSize);
plot(x,yLS,'color','r','linewidth',LineWidth);
plot(x,yLS+delta,'color','g','linewidth',1);
plot(x,yLS-delta,'color','g','linewidth',1);
s=sprintf(...
    'a_1=%6.3f, a_2=%6.3f, a_3=%6.3f, a_4=%6.3f,  r^2=%7.4f',...
    p(4),p(3),p(2),p(1),rSquare);
fprintf('MATLAB:  %s\n',s);

title(s), xlabel('x'), ylabel('y'), box on
if PrintFlag
    print('-dpsc2', 'E77N-Figure16.1.eps');
end
M=4; % Cubic polynomial
%---------------------------------------------------------------------
% Pointer to the function that stores and evaluates
% basis functions
%---------------------------------------------------------------------
fhandle = @BasisFunctions;
% Class Reader function
[aout, Covariances, RSquare]=GLLS(x, y, Sigma, M, fhandle);
p2=flipud(aout);
[yLS2] = polyval(p2,x);

figure(2)
plot(x,y,'+')
hold on
set(gca,'FontSize',FontSize);
```

```
plot(x,yLS2,'color','r','linewidth',LineWidth);
s=sprintf('a_1=%6.3f, a_2=%6.3f, a_3=%6.3f, a_4=%6.3f,  r^2=%7.4f',...
    p2(4),p2(3),p2(2),p2(1),RSquare);
fprintf('READER: %s\n',s);
title(s), xlabel('x'), ylabel('y'), box on
if PrintFlag
    print('-dpsc2', 'E77N-Figure16.2.eps');
end
%Compare polyfit with the Reader solution
figure(3)
plot(x,y,'+')
hold on
set(gca,'FontSize',FontSize);
[yLS2] = polyval(p2,x);
plot(x,yLS ,'color','b','linewidth',LineWidth);
plot(x,yLS2,'color','r','linewidth',LineWidth);
legend('Data','MATLAB','Reader')
title('MATLAB vs Class Reader'), box on, xlabel('x'), ylabel('y')
```

20.6 General Linear Least Square Fit Algorithm Based on Eqs. (20.9 - 20.10)

```
function [aout, Covar, RSquare]=...
    GLLS(xs, ys, Sigmas, M, fhandle)
%-----------------------------------------------------------
%  Function GLLS performs least squares
%  fit of data with a nonlinear model y=a1+a2*x+a3*x^2+...
%
%  Input arguments:
%       xs      = vector of independent variable
%       ys      = vector of dependent variable
%       Sigmas  = vector of STD's in ys (if you do not
%                 know them input a vector of ones)
%       M       = Order of approximation
%       fhandle = pointer to a function that evaluates
%                 basis functions at each abscissa
%
%   Output arguments:
%       aout    = vector of fit coefficients of length M:
%       yLS     = a(1)+a(2)*x+...a(M)*x^(M-1)
%                 Thus the a's must be flipped before using polyval
%       Covar   = matrix of covariances of fit coefficients
%       RSquare = R^2 statistic, or goodness of fit  (approximate expression)
%
% Written by T. W. Patzek, March 10, 2001
% UC Berkeley, 437 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834 or 510-486-5322,
% email: patzek@patzek.berkeley.edu
```

```
%
% Last revision 03/26/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.26 12:15:00 $
%-------------------------------------------------------------
%                          PRELIMINARIES
if     nargin < 1
    error(’You must input vector of xs’)
elseif nargin < 2
    error(’You must input vector of ys’)
elseif nargin < 3
    error(’You must input vector of Sigmas’)
elseif nargin < 4
    error(’You must input order of LS fit’)
elseif nargin < 5
     erorr(’You must specify pointer to basis functions’)
end
% Initialize inputs
aout    = [];
Covar   = [];
RSquare = 0;
if (M==0)
    error(’There is no parameters to be fit’);
end
% Check for consistency
% Force all inputs to be column vectors
xs     = (xs(:)’)’;
ys     = (ys(:)’)’;
Sigmas = (Sigmas(:)’)’;
llx    = length(xs);
lly    = length(ys);
lls    = length(Sigmas);
if (llx~=lly)
    error(’Vectors x and y must have the same length’);
end
N=llx;
if (lls ~= N)
    error(’Sigma must have the same length as x and y’);
end
%-----------------------------------------------------
%                     START THE CODE
%                  Class Reader Chapter 20
%-----------------------------------------------------
Betas = zeros(M,1);
Covar = zeros(M,M);
% Evaluate all basis functions at all abscissas
A      = feval(fhandle, xs, M, Sigmas);
ATA = A’*A;
B  = A’*(ys./Sigmas);
% Calculate the inverse of A’*A, or the covariance matrix
Covar   = inv(ATA);
```

```
aout    = Covar*B;
yLS     = polyval(flipud(aout),xs);
R = corrcoef(ys,yLS);
RSquare=R(1,2)^2;



function [Values] = BasisFunctions(xi, M, Sigmas)
%---------------------------------------------------------------
% This BasisFunctions function employs monomials of x
% Any other set of basis functions could be used
%
%   Input arguments:
%       xi     = column vector of independent variable
%       M      = number of fit coefficients
%       Sigmas = standard deviations of dependent variable
%   Output arguments:
%       Values = values of M basis functions evaluated at N=len
%                abscissas
%
% Written by T. W. Patzek, March 10, 2001
% Last revision 03/26/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.26 11:55:00 $
%---------------------------------------------------------------
Values = zeros(length(xi),M);
for i=1:M
    Values(:,i) = xi.^(i-1)./Sigmas;
end
```

Chapter 21

Interpolation and Approximation

21.1 What Are You Going To Learn?

You will learn how to interpolate functions using linear polynomials and LAGRANGE polynomials.

In Chapter 21 folder on fttp:\\petroleum.berkeley.edu, you will find the following MATLAB files:

LinearInt.m performs piecewise linear interpolation.

LagrangeInt.m performs interpolation with the highest order LAGRANGE polynomial possible.

vibration.m demonstrates performance of piecewise linear and the LAGRANGE interpolations of a periodic function and a step function.

LagrangeDemo.m shows how 2, 3, and 4 points of an arbitrary function are interpolated by the LAGRANGE polynomials of order 1, 2, and 3.

CSDemo.m Demonstrates performance of cubic spline approximations of a periodic function and step function.

21.2 Why Is It Important?

Interpolation and approximation techniques are used every day in all branches of engineering and science.

21.3 MATLAB Shortcut

MATLAB has one-, two- and three-dimensional interpolation functions, `interp1`, `interp2` and `interp3`, respectively. In each function, you can specify one of the following interpolation methods:

1. Nearest neighbor interpolation: 'nearest'.
2. Linear interpolation (default): 'linear'.
3. Cubic spline interpolation: 'spline'.
4. Piecewise cubic Hermite interpolation: 'pchip'.

See MATLAB help for details.

21.4 Preliminaries

In the Chapter 17, we introduced TAYLOR series, i.e., the special *polynomial* approximation of a *differentiable* function:

$$\begin{aligned} P_n(x) = f(x_0) + \frac{f'(x_0)}{1!}(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \frac{f'''(x_0)}{3!}x^3 + \ldots \\ + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n \end{aligned} \tag{21.1}$$

which uniformly approximates the function f in a closed interval $[a, b]$, provided that the function f is sufficiently smooth. The difference between this polynomial and the function itself is an infinitesimally small quantity of the order higher than $(x - x_0)^n$, i.e.,

$$R_{n+1}(x) \equiv f(x) - P_n(x) = o(x - x_0)^n \tag{21.2}$$

Note the "small o" quantity in Eq. (21.2). We will discuss such quantities later in the course.

21.5 Approximation

There are many reasons to approximate a function. The type of approximation sought depends upon the application and the ease or difficulty with which it can be obtained. In all cases, the simplest approximating functions are polynomials, but trigonometric functions are also useful.

In general, a polynomial $P_n(x)$ of degree at most n approximates a function $f(x)$ in an interval $a \leq x \leq b$ if some measure of the deviation of the polynomial from the function in this interval is "small"

$$||f(x) - P_n(x)|| < \varepsilon << 1 \tag{21.3}$$

We call this deviation a *norm*. We should remember that what might be a good approximation in one norm is a poor approximation in another norm. Once we have defined the norm, three questions must be answered:

1. Does a polynomial exist, of a specified maximum degree, which minimizes the error?
2. If such a polynomial exits, is it unique?
3. How can it be determined, if it exists?

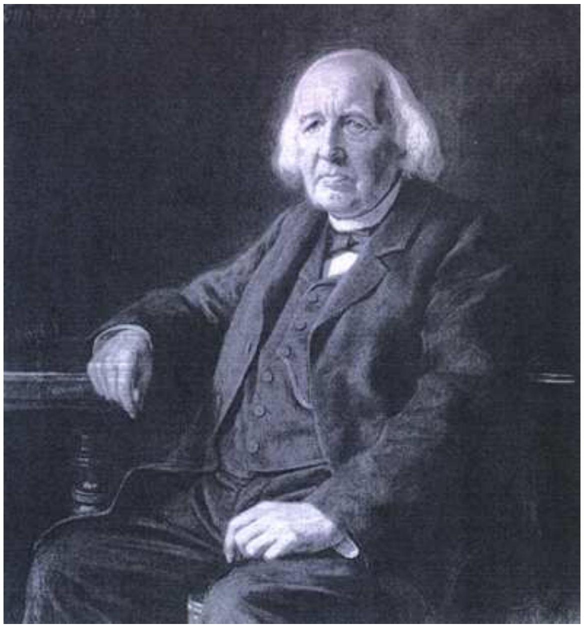


Figure 21.1: Karl Theodor Wilhelm Weierstrass, born October 31, 1815, in Ostenfelde, Westphalia (now Germany), died February 19, 1897, in Berlin, Germany.

Weierstrass's Approximation Theorem provides answers to these questions:

Theorem 1 *Let $f(x)$ be a function continuous in the closed interval $[a, b]$. Then for any $\varepsilon > 0$ there exists and integer $n = n(\varepsilon)$ and a polynomial $P_n(x)$ of degree at most n such that*

$$|f(x) - P_n(x)| < \varepsilon \tag{21.4}$$

for all $x \in [a, b]$. □

This theorem guarantees that arbitrarily close polynomial approximations are possible throughout a closed bounded interval, provided that the function being approximated is continuous. This statement gives no hint how to construct such a polynomial. Such hints were provided by Bernstein and many others.

21.6 Interpolation

An approximation polynomial, which is equal to the function it approximates at a number of specified points is called an *interpolation* polynomial. Given the $n+1$ distinct points x_i , $i = 0, 1, ..., n-1, n$, and corresponding function values $y_i = f(x_i)$, the interpolation of degree at most n minimizes

$$\sum_{i=0}^{n} |y_i - P_n(x_i)| \tag{21.5}$$

The minimum value of Eq. (21.5) is zero. Thus, we seek a polynomial

$$P_n(x) = \sum_{k=0}^{n} a_k x^k \tag{21.6}$$

for which

$$P_n(x_i) \equiv y_i, \quad i = 0, 1, \ldots, n \tag{21.7}$$

By considering the coefficients a_k in Eqs. (21.6) as unknowns, we have a system of $n+1$ linear equations:

$$\begin{aligned} P_n(x_0) &= y_0 = a_0 + a_1 x_0 + \cdots + a_n x_0^n \\ P_n(x_1) &= y_1 = a_0 + a_1 x_1 + \cdots + a_n x_1^n \\ &\vdots \\ P_n(x_n) &= y_n = a_0 + a_1 x_n + \cdots + a_n x_n^n. \end{aligned} \tag{21.8}$$

The system of equations (21.8) has a unique solution if the coefficient matrix is non-singular. The determinant of this matrix is called a VANDERMONDE determinant and it can be easily evaluated to yield

$$\begin{vmatrix} 1 & x_0 & ... & x_0^n \\ 1 & x_1 & ... & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_n & ... & x_n^n \end{vmatrix} = \prod_{i>j} (x_i - x_j) = \prod_{j=0}^{n-1} \left[\prod_{i=j+1}^{n} (x_i - x_j) \right] . \tag{21.9}$$

Since $\{x_i\}$ are distinct points, the determinant is nonzero and a unique solution to Eqs. (21.8) exists.

Rather than solve Eqs. (21.8), one may use an alternative procedure to obtain the interpolation procedure directly. Set

$$P_n(x) = \sum_{j=0}^{n} f(x_j) \phi_{n,j}(x) \tag{21.10}$$

where the $n+1$ functions $\phi_{n,j}(x)$ are nth degree polynomials. We note that $P_n(x_i) = 0$ if the polynomials $\phi_{n,j}(x)$ satisfy

$$\phi_{n,j}(x_i) = \begin{cases} 1 \text{ if } i = j \\ 0, \text{ otherwise} \end{cases} \qquad i, j = 0, 1, 2, ..., n \tag{21.11}$$

Such polynomials are easily constructed

$$\phi_{n,j} = \frac{(x-x_0)(x-x_1)...(x-x_{j-1})(x-x_{j+1})...(x-x_n)}{(x_j-x_0)(x_j-x_1)...(x_j-x_{j-1})(x_j-x_{j+1})...(x_j-x_n)} \qquad (21.12)$$
$$j = 0, 1, 2, ...n$$

The interpolation polynomial (21.10) is called the LAGRANGE interpolation polynomial and the coefficients (21.11) are called the LAGRANGE interpolation coefficients. We can use the product notation for each ϕ, which yields

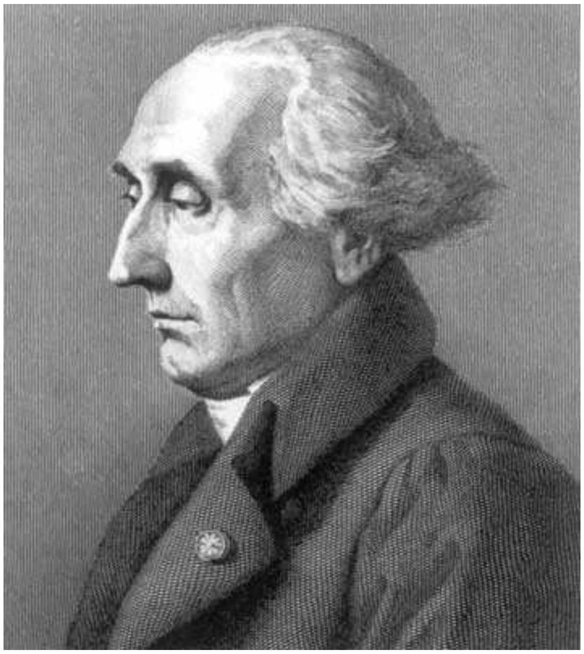


Figure 21.2: JOSEPH-LOUIS LAGRANGE, born January 25, 1736, in Turin, Sardinia-Piedmont (now Italy), died April 10, 1813, in Paris, France.

$$P_n(x) = \sum_{j=0}^{n} f(x_j) \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{x-x_k}{x_j-x_k} \qquad (21.13)$$

Here is how it works: Let's take an arbitrary function, say, $f(x) = 3x\sin^2(x)$, and pick up 2, 3 or 4 arbitrary points, $x = x_0$, $x = x_1$, $x = x_2$, $x = x_3$ (they do not have to start at the leftmost end of the interval shown in **Figure 21.3**).

We assume that we know the values of the function at these points: $f(x_0) = y_0$, $f(x_1) = y_1$, $f(x_2) = y_2$, $f(x_3) = y_3$. We will now construct the Lagrange interpolation polynomials of order one, two and three. As we remember from elementary geometry, through any two points we can draw a unique straight

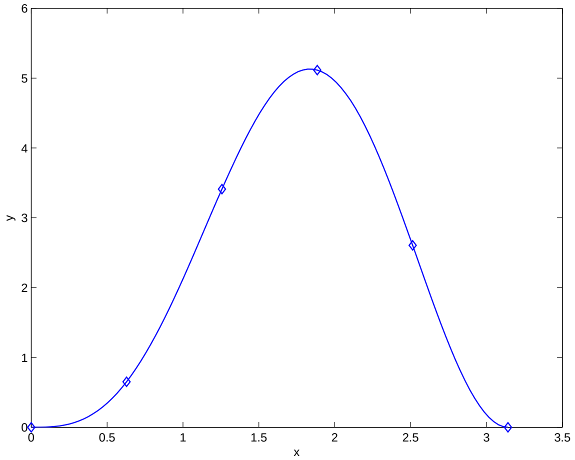


Figure 21.3: Suppose that we want to interpolate $f(x) = 3x\sin(x)$ using a linear, quadratic or cubic Lagrange interpolator, passing through 2, 3 or 4 consecutive points denoted by the diamonds, and starting from the second point on the left.

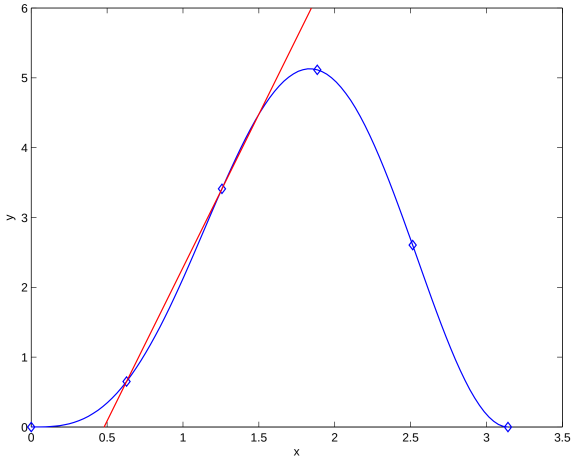


Figure 21.4: Linear interpolation of $f(x)$ that passes through points x_1 and x_2.

line. Through any three points that do not belong to a line, we can draw a unique parabola, and through any four points that do not belong to a parabola, we can draw a unique cubic.

The equation of the line passing through two points can be written as

$$y = \frac{x - x_1}{x_0 - x_1} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 \tag{21.14}$$

and the result is shown in **Figure 21.4**.

The parabola passing through points $x_0 - x_2$ has the following equation:

$$y = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 \tag{21.15}$$

and the result is shown in **Figure 21.5**.

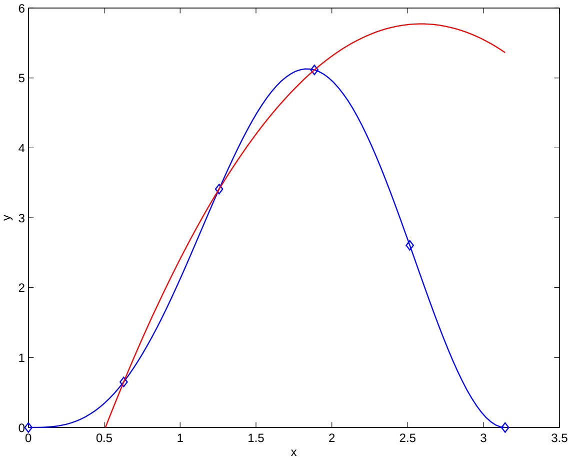


Figure 21.5: Quadratic interpolation of $f(x)$ that passes through points x_1, x_2 and x_3.

The cubic curve that passes through points $x_0 - x_3$ has the following equation:

$$\begin{aligned} y = &\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}y_0 + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}y_1 + \\ &\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}y_2 + \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}y_3 \end{aligned} \tag{21.16}$$

and the result is shown in **Figure 21.6**. All these calculations and plots are performed by the `LagrangeDemo.m` function, listed in Section 21.8.

21.7 Lagrange Interpolations of Different Orders

Suppose that we know 25 discrete sensor positions at some time, and these positions are given by the following equation, **Figure 21.7**:

$$f(x_i) = 3\sin\left(\frac{i}{1.1}\right) + 4$$
$$i = 1, 2, \ldots, 25$$

We want to approximate these positions using linear interpolation (straight line through each pair of sensor positions), **Figure 21.8**, and a 24th degree

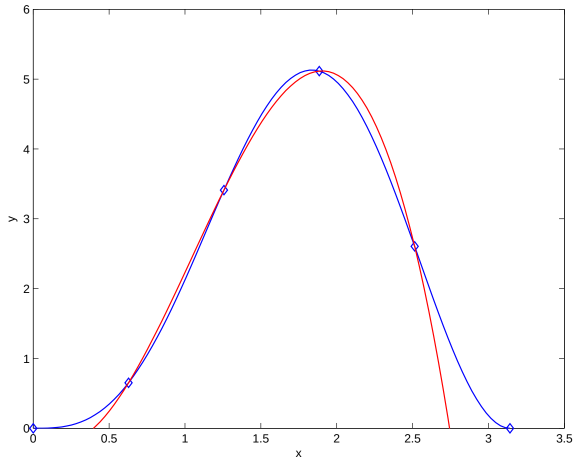


Figure 21.6: Cubic interpolation of $f(x)$ that passes through points x_1, x_2, x_3 and x_4.

Lagrange polynomial, **Figure 21.9**. In the linear LAGRANGE interpolation case, we calculate 10 interpolated points between each pair of sensor positions. In the 24th degree LAGRANGE polynomial interpolation case, there is only one unique polynomial passing through all 25 sensor positions. As we can see for the *very* smooth sensor position data, the LAGRANGE interpolation does a good job.

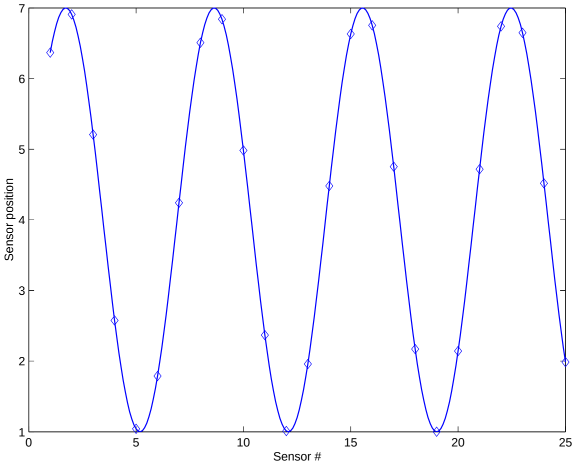


Figure 21.7: Discrete amplitudes (diamonds) of 25 sensors to be interpolated by piecewise linear and polynomial LAGRANGIAN schemes (Files `vibration.m`, `LinearInt.m` and `LagrangeInt.m`.)

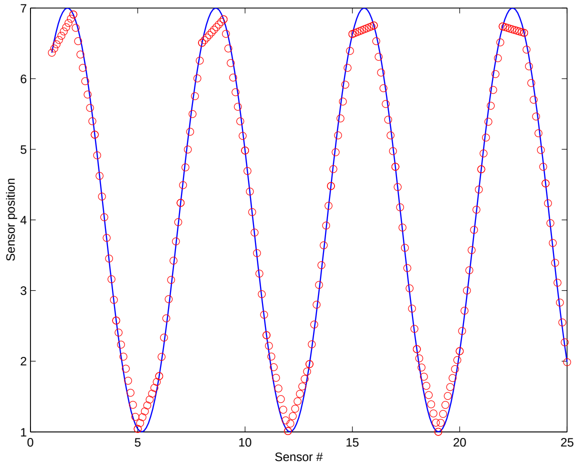


Figure 21.8: Result of piecewise linear LAGRANGE interpolation of the 25 sensor positions in Figure 21.7, with 10 points between each pair of data points.

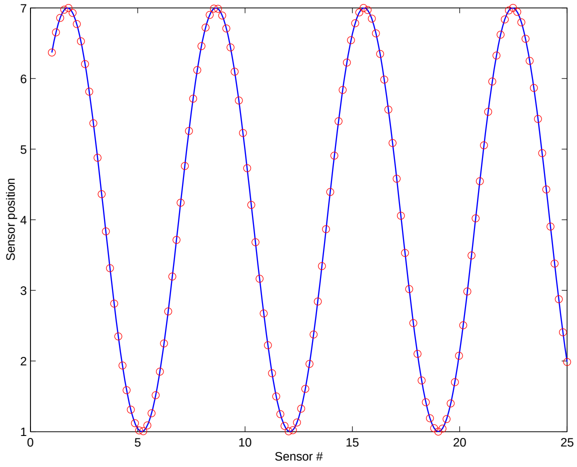


Figure 21.9: Result of the 24th degree LAGRANGE polynomial interpolation of the 25 sensor positions in Figure 21.7. Note that the 24th degree LAGRANGE polynomial is not local, i.e., it retains information about the global function behavior.

Suppose that now that approximate 10 discontinuous sensor positions with the following equation, **Figure 21.10**:

$$f(x_i) = \begin{cases} 1 & \text{if } i \leq 5 \\ 24.5 & \text{if } i > 5 \end{cases}$$

$$i = 1, 2, \ldots, 10$$

We approximate these positions using linear interpolation (straight line through each pair of sensor positions), **Figure 21.11**, and a 9th degree Lagrange polynomial, **Figure 21.12**. In the linear LAGRANGE interpolation case, we calculate 10 interpolated points between each pair of sensor positions. In the 9th degree LAGRANGE polynomial interpolation case, there is only one unique polynomial passing through all 10 sensor positions. As we can see for the *very* discontinuous sensor position data, the LAGRANGE interpolation fails miserably. The MATLAB code is listed in listed in Section 21.9.

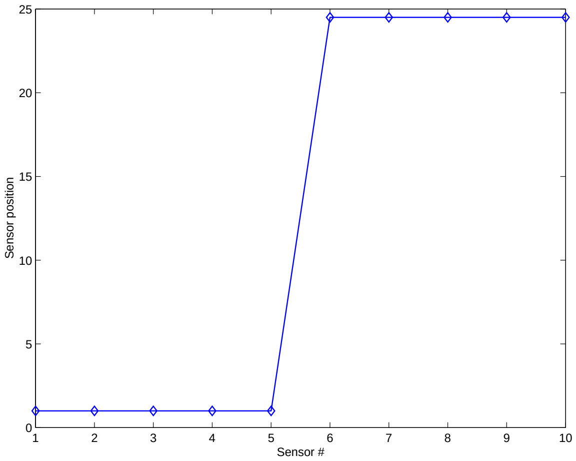


Figure 21.10: Step function in sensor amplitudes.

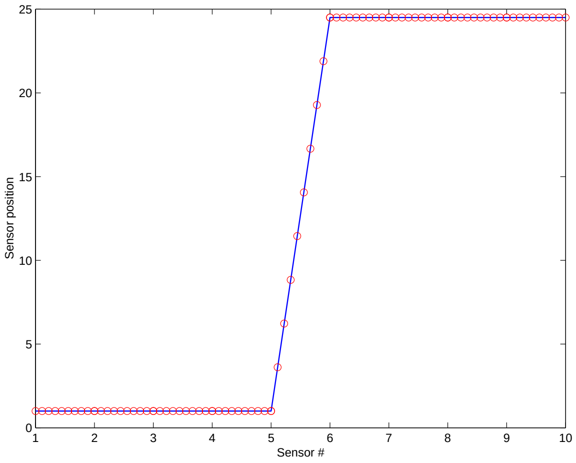


Figure 21.11: Piecewise linear LAGRANGE interpolation of a step function works very well.

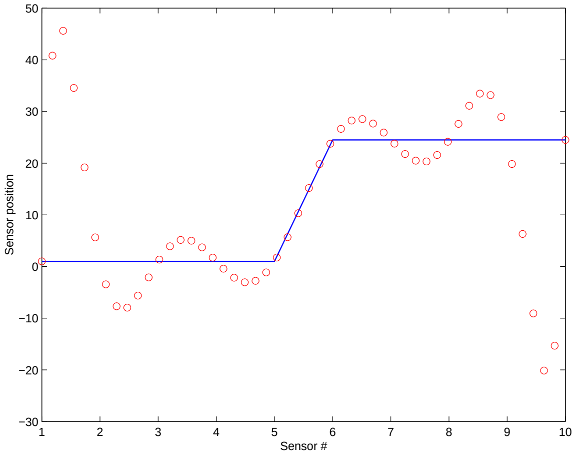


Figure 21.12: 9th-degree polynomial LAGRANGE interpolation of a step function fails miserably.

21.8 Lagrange Interpolation Demo

```
function LagrangeDemo()
%-----------------------------------------------------------------------
% Shows how 2, 3,and 4 points of an arbitrary function
% are interpolated by Lagrange polynomials of order 1, 2, and 3
%
% Written by T. W. Patzek, March 05, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-643-5834
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/29/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.29 17:00:00 $
%-----------------------------------------------------------------------

clear all, close all, clc
PrintFlag  = 0; %<<<<<No hard copy, 1 otherwise
LineWidth = 1;
if ~PrintFlag
    LineWidth=2;
end

x  = (0:0.2:1)*pi;
y  = 3*x.*(sin(x)).^2;
xc =(0:0.01:1)*pi;
yc = 3*xc.*(sin(xc)).^2;
ll = length(x);
```

```
figure(1), hold on
if ~PrintFlag
     set(gca, ’FontSize’,14);
end
plot(x,y,’d’, ’linewidth’,LineWidth)
plot(xc,yc,’linewidth’,LineWidth)
xlabel(’x’), ylabel(’y’), box on
axis([0,3.5,0,6]);
if ~PrintFlag
    title(’f(x)=3 x sin^2(x)’);
else
    print(’-dpsc2’, ’E77N-Figure17.1.eps’)
end
% Linear Lagrange polynomial
% Pick any two points, say 2 and 3
x0 = x(2);
x1 = x(3);
y0 = y(2);
y1 = y(3);
% Construct the linear Lagrange interpolator
P1 = (x-x1)/(x0-x1)*y0 +(x-x0)/(x1-x0)*y1;

figure(2), hold on
if ~PrintFlag
     set(gca, ’FontSize’,14);
end
plot(x,y,’d’, ’linewidth’,LineWidth)
plot(xc,yc,’linewidth’,LineWidth);
plot(x,P1,’color’,’r’,’linewidth’,LineWidth)
xlabel(’x’), ylabel(’y’), box on
axis([0,3.5,0,6]);
if ~PrintFlag
   title(’Linear Lagrange interpolation  through 2 points’)
else
    print(’-dpsc2’, ’E77N-Figure17.2.eps’)
end
% Quadratic Lagrange polynomial
% Pick any three points, say 2, 3 and 4
x0 = x(2);
x1 = x(3);
x2 = x(4);

y0 = y(2);
y1 = y(3);
y2 = y(4);
% Construct the quadratic Lagrange interpolator
P2 = (xc-x1).*(xc-x2)/((x0-x1)*(x0-x2))*y0+...
     (xc-x0).*(xc-x2)/((x1-x0)*(x1-x2))*y1+...
     (xc-x0).*(xc-x1)/((x2-x0)*(x2-x1))*y2;
```

```
figure(3), hold on
if ~PrintFlag
     set(gca, ’FontSize’,14);
end
plot(x,y,’d’, ’linewidth’,LineWidth)
plot(xc,yc,’linewidth’,LineWidth);
plot(xc,P2,’color’,’r’,’linewidth’,LineWidth)
xlabel(’x’), ylabel(’y’), box on
axis([0,3.5,0,6]);
if ~PrintFlag
    title(’Quadratic Lagrange interpolation through 3 points’)
else
    print(’-dpsc2’, ’E77N-Figure17.3.eps’)
end

% Cubic Lagrange polynomial
% Pick any four points, say 2,3,4 and 5
x0 = x(2);
x1 = x(3);
x2 = x(4);
x3 = x(5);

y0 = y(2);
y1 = y(3);
y2 = y(4);
y3 = y(5);
% Construct the cubic Lagrange interpolator
P3 = (xc-x1).*(xc-x2).*(xc-x3)/((x0-x1)*(x0-x2)*(x0-x3))*y0+...
     (xc-x0).*(xc-x2).*(xc-x3)/((x1-x0)*(x1-x2)*(x1-x3))*y1+...
     (xc-x0).*(xc-x1).*(xc-x3)/((x2-x0)*(x2-x1)*(x2-x3))*y2+...
     (xc-x0).*(xc-x1).*(xc-x2)/((x3-x0)*(x3-x1)*(x3-x2))*y3;

figure(4), hold on
if ~PrintFlag
     set(gca, ’FontSize’,14);
end
plot(x,y,’d’, ’linewidth’,LineWidth)
plot(xc,yc,’linewidth’,LineWidth);
plot(xc,P3,’color’,’r’,’linewidth’,LineWidth)
xlabel(’x’), ylabel(’y’), box on
axis([0,3.5,0,6]);
if ~PrintFlag
    title(’Cubic Lagrange interpolation through 4 points’)
else
    print(’-dpsc2’, ’E77N-Figure17.4.eps’)
end

figure(5), hold on
if ~PrintFlag
     set(gca, ’FontSize’,14);
```

```
end
plot(x,y,'d', 'linewidth',LineWidth)
plot(x,P1,'color','m','linewidth',LineWidth,...
    'marker','none','linestyle','--')
plot(xc,P2,'color','k','linewidth',LineWidth,...
    'marker','none','linestyle','-.')
plot(xc,P3,'color','r','linewidth',LineWidth,...
    'marker','none','linestyle',':')
plot(xc,yc,'linewidth',LineWidth);
xlabel('x'), ylabel('y'), box on
legend('Function','Linear', 'Quadratic', 'Cubic',2)
axis([0,3.5,0,6]);
if ~PrintFlag
    title('Lagrange interpolation')
else
    print('-dpsc2', 'E77N-Figure17.1a.eps')
end
```

21.9 Interpolation of Sensor Positions

```
function vibration()
%---------------------------------------------------------
% Demonstrates performance of piecewise linear and
% Lagrange interpolations of a periodic function and a
% step function
%
% Written by T. W. Patzek, March 05, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-643-5834
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/29/2002, TWP
% $Revision: 1.0 $  $Date: 2002.03.29 17:00:00 $
%---------------------------------------------------------
clear all, close all, clc
PrintFlag = 0;
LineWidth =1;
FontSize =12;
if ~PrintFlag
    LineWidth =2;
    FontSize = 14;
end

%-------------------------------------------------
%                    Smooth function
%-------------------------------------------------
% Discrete sensor positions to be interpolated
xsensor=1:1:25;
vib(1,:)=xsensor;
vib(2,:) = 3*sin(xsensor/1.1) + 4;

x = 1:0.1:25;
y = 3*sin(x/1.1) + 4;

% Plot the sensors and the function they represent
figure(1), hold on
plot(x,y,'-', 'linewidth',LineWidth)
plot(vib(1,:), vib(2,:),'d');
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
    title('Discrete positions of 25 sensors')
else
    print('-dpsc2', 'E77N-Figure17.5.eps')
end

% Perform piecewise interpolations of xsensor, ysensor arrays
[xint, yint]= LinearInt(vib(1,:),vib(2,:),10);

figure(2), hold on
plot(x,y,'-', 'linewidth',LineWidth);
plot(xint,yint,'marker', 'o', 'color', 'r', 'linestyle', 'none')
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
    title('Piecewise linear interpolation')
else
    print('-dpsc2', 'E77N-Figure17.6.eps')
end

% Perform Lagrange interpolations of xsensor, ysensor arrays
[xint, yint]= LagrangeInt(vib(1,:),vib(2,:));

figure(3), hold on
plot(x,y,'-', 'linewidth',LineWidth);
plot(xint,yint,'marker', 'o', 'color', 'r', 'linestyle', 'none')
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
    title('Lagrangian polynomial interpolation')
else
    print('-dpsc2', 'E77N-Figure17.7.eps')
end
%-------------------------------------------------
%                    Step function
%-------------------------------------------------
x = 1:1:10;
y=[ones(1,5), 24.5*ones(1,5)];
```

```
%Plot the step function
figure(4)
plot(x,y,'d-', 'linewidth',LineWidth)
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
title('Discrete positions of 10 sensors')
else
    print('-dpsc2', 'E77N-Figure17.8.eps')
end

% Perform piecewise interpolations of xsensor, ysensor arrays
[xint, yint]= LinearInt(x,y,10);

figure(5), hold on
plot(x,y,'-', 'linewidth',LineWidth);
plot(xint,yint,'marker', 'o', 'color',...
    'r', 'linestyle', 'none')
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
    title('Piecewise linear interpolation')
else
    print('-dpsc2', 'E77N-Figure17.9.eps')
end

% Perform Lagrange interpolations of xsensor, ysensor arrays
[xint, yint]= LagrangeInt(x,y);

figure(6), hold on
plot(x,y,'-', 'linewidth',LineWidth);
plot(xint,yint,'marker', 'o', 'color', 'r', 'linestyle', 'none')
set(gca,'FontSize',FontSize);
xlabel('Sensor #'), ylabel('Sensor position'), box on
if ~PrintFlag
    title('Lagrangian polynomial interpolation')
else
    print('-dpsc2', 'E77N-Figure17.10.eps')
end

function [x, PnOfx]= LagrangeInt(xpts,ypts)
%----------------------------------------------------------------
%  Input arguments:
%       xpts  = vector of x values
%       ypts  = vector of y values
%
%  Output arguments:
%       x     = vector of interpolated x's
%       PnOfx = vector of y's interpolated with the
%               highest order Lagrange polynomial possible
```

```
%
% Written by T. W. Patzek, March 05, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-643-5834
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/20/2006, TWP
% $Revision: 1.0 $  $Date: 2006.03.20 09:00:00 $
%---------------------------------------------------------------
x = []; PnOfx = [];
if nargin <1
    error(’You must specify the input array’);
end
% Set the degree of the Lagrange polynomial
% to the number of points
% Only column vectors are allowed
xpts   = (xpts(:)’)’;
ypts   = (ypts(:)’)’;
ll     = length(xpts);
lly    = length(ypts);
if (ll˜=lly)
    error(’xpts and ypts vectors must have equal length’);
end

n = length(xpts);
xmax = max(xpts);
xmin = min(xpts);

% I am going to generate 5*n points using
% Lagrange interpolation
x = linspace(xmin,xmax,5*n);
lagrange = ones(n,5*n);
PnOfx = zeros(size(x));
for j=1:n
    for k=1:n
        if(k==j)
            % Do nothing
        else
            % Accumulate the product (function phi in Chapter 21)
            lagrange(j,:) = lagrange(j,:).* ...
            ((x-xpts(k))/(xpts(j)-xpts(k)));
        end
    end
    % Equation 13 in Chapter 21....
    PnOfx = lagrange(j,:)*ypts(j) + PnOfx;
end

function [xint, yint] = LinearInt(x,y,n)
%---------------------------------------------------------------
%  Input arguments:
```

```
%       x = vector of x values
%       y = vector of y values
%       n = number of points in between consequtive data points
%           to perform piecewise linear interpolation
%
%  Output arguments:
%       xint = vector of linearly interpolated x’s
%       ys = vector of linearly interpolated y’s
%
% Written by T. W. Patzek, March 05, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-643-5834
% email: patzek@patzek.berkeley.edu
%
% Last revision 03/20/2006, TWP
% $Revision: 1.0 $  $Date: 2006.03.20 09:15:00 $
%---------------------------------------------------------------
xint = []; yint = [];
if nargin < 1
    error(’You must specify the input array’);
elseif nargin <2
    n = 10;
end
%Only column vectors are allowed
x   = (x(:)’)’;
y   = (y(:)’)’;
ll  = length(x);
lly = length(y);
if (ll~=lly)
    error(’x and y vector must have equal length’);
end

intx = zeros(ll-1,n);
inty = zeros(ll-1,n);
% Linear interpolation
for j=1:(size(x)-1)
    % Fill in n points between two consecutive x data points
    intx(j,:) = linspace(x(j),x(j+1),n);
    inty(j,:) = (y(j+1)-y(j))/(x(j+1)-x(j))*(intx(j,:)-x(j))+y(j);
end
xint = intx(:);
yint = inty(:);
```

Chapter 22

Numerical Differentiation

22.1 What Are You Going To Learn?

You will learn how calculate the first derivative and higher derivatives of functions that are known only at discrete points. You will learn to appreciate that numerical differentiation is tricky and may lead to very large errors.

In Chapter 22 folder on fttp:\\petroleum.berkeley.edu look up the following MATLAB files:

derivatives.m approximates the input data with cubic splines over the interval $[x_{\min}, x_{\max}]$ that can extend beyond the interval min(xdata), max(xdata). It also calculates the first and second derivative of a function using central differences of the interpolated data.

DiffDemo.m demonstrates performance of cubic spline approximations of a periodic function and its first two derivatives. Calls `derivatives`.

22.2 Why Is It Important?

Any rate-of-change parameter (the first derivative) is the slope of a system response. Engineers must perform their tasks in finite time; therefore, rates are of interest. Sometimes, accelerations, i.e., the second derivatives may also be important.

22.3 MATLAB Shortcut

In MATLAB, `Y = diff(X)` calculates differences between adjacent elements of `X`. If `X` is a multidimensional array, you have control over which dimension to take the differences. Once you obtained the differences, you might be able to calculate the approximate derivatives. See MATLAB Help for more details.

22.4 Finite Differences

22.4.1 First Derivative

Take x_0 as an arbitrary fixed starting point and let $h > 0$ be the spacing between the adjacent points. Then consider the points

$$x_j = x_0 + jh, \quad j = 0, \pm 1, \pm 2, ... \tag{22.1}$$

Note that x_0 is not the end-point, and neither are the points $x_{-1} = x_0 - h$, etc., to the left of it, **Figure 22.1**. We shall assume that all $x_j \in [a, b]$.

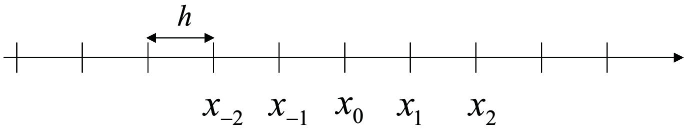


Figure 22.1: Equally spaced points on the x-axis.

Associated with equally spaced points is the forward difference, which is defined by

$$\Delta_+ f(x) \equiv f(x+h) - f(x) \tag{22.2}$$

Therefore

$$\begin{aligned} \Delta_+ f(x_0) &= f(x_0 + h) - f(x_0) = f(x_1) - f(x_0) \\ \Delta_+ f(x_1) &= f(x_1 + h) - f(x_1) = f(x_2) - f(x_1) \\ &\dots \end{aligned} \tag{22.3}$$

We can also associate with equally spaced points the backward difference, defined as

$$\Delta_- f(x) \equiv f(x) - f(x-h) \tag{22.4}$$

Therefore

$$\begin{aligned} \Delta_- f(x_0) &= f(x_0) - f(x_0 - h) = f(x_0) - f(x_{-1}) \\ \Delta_- f(x_1) &= f(x_1) - f(x_1 - h) = f(x_1) - f(x_0) \\ &\dots \end{aligned} \tag{22.5}$$

Finally, we can associate with equally spaced points the central difference, defined as

$$\Delta_\pm f(x) \equiv f(x+h) - f(x-h) \tag{22.6}$$

Therefore

$$\begin{aligned} \Delta_\pm f(x_0) &= f(x_0 + h) - f(x_0 - h) = f(x_1) - f(x_{-1}) \\ \Delta_\pm f(x_1) &= f(x_1 + h) - f(x_1 - h) = f(x_2) - f(x_0) \\ &\dots \end{aligned} \tag{22.7}$$

Remark 27 Before differencing a function, we need to interpolate the irregularly spaced function values to an equal spacing, h, using a robust interpolation scheme. If the interpolating polynomial oscillates, our regularly spaced function values may be meaningless. □

The first derivative is the slope of a function graph. If a function has the first derivative at a point x_0, then

$$\begin{aligned} f'(x_0) &= \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} = \lim_{h\to 0} \frac{\Delta_+ f(x_0)}{h} \\ f'(x_0) &= \lim_{h\to 0} \frac{f(x_0)-f(x_0-h)}{h} = \lim_{h\to 0} \frac{\Delta_- f(x_0)}{h} \\ f'(x_0) &= \lim_{h\to 0} \frac{f(x_0+h)-f(x_0-h)}{2h} = \lim_{h\to 0} \frac{\Delta_\pm f(x_0)}{2h} \end{aligned} \tag{22.8}$$

Given that the function values are known sufficiently well, how small should h be to approximate the first derivative?

To answer this question, we need to assume that our function has three continuous derivatives in a neighborhood of point x_0 and the fourth derivative exists at point x_0. Once this assumption is made, we may expand the function into a Taylor series:

$$f(x) \approx f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2}f''(x_0)(x-x_0)^2 + \frac{1}{6}f'''(x_0)(x-x_0)^3 \tag{22.9}$$

Then

$$\begin{aligned} f(x_0+h) &\approx f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \frac{1}{6}f'''(x_0)h^3 \\ f(x_0-h) &\approx f(x_0) - f'(x_0)h + \frac{1}{2}f''(x_0)h^2 - \frac{1}{6}f'''(x_0)h^3 \end{aligned} \tag{22.10}$$

We can use these function expansions to estimate the approximation error in the first derivative if we use the forward, backward and central finite difference

$$\begin{aligned} \frac{f(x_0+h)-f(x_0)}{h} &\approx +f'(x_0) + \frac{1}{2}f''(x_0)h + \frac{1}{6}f'''(x_0)h^2 \\ \frac{f(x_0)-f(x_0-h)}{h} &\approx f'(x_0) - \frac{1}{2}f''(x_0)h + \frac{1}{6}f'''(x_0)h^2 \\ \frac{f(x_0+h)-f(x_0-h)}{2h} &\approx f'(x_0) + \frac{1}{6}f'''(x_0)h^2 \end{aligned} \tag{22.11}$$

Remark 28 If h is small, $h \ll 1$, then h^2 is even smaller. Both the forward and the backward finite difference provide estimates of the first derivative that are first order in h. The central difference provides an estimate of the first derivative that is second order in h. **Figure 22.2** compares the three methods. □

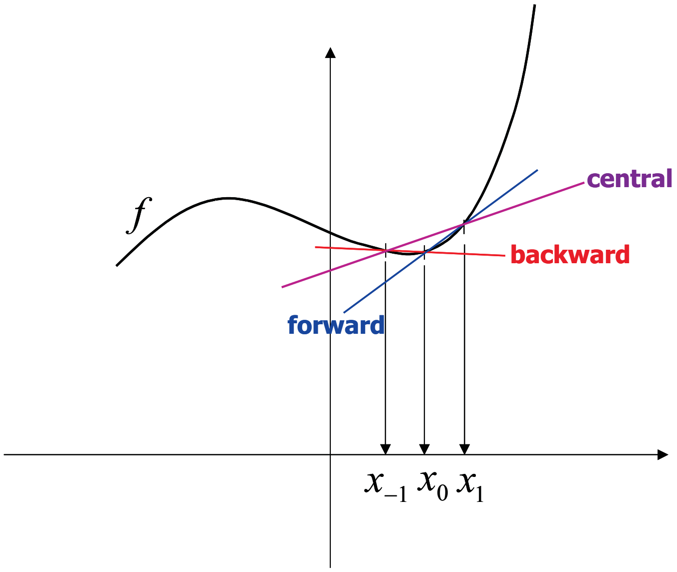


Figure 22.2: Slope approximations using the forward, backward and central finite differences.

22.4.2 Second Derivative

The second derivative of a function is related to its second differences.

The second difference can be calculated as the difference of the first differences:

$$\begin{aligned}\Delta_+^2 f(x) &= \Delta_+ f(x+h) - \Delta_+ f(x) = f(x+2h) - 2f(x+h) + f(x)\\ \Delta_-^2 f(x) &= \Delta_- f(x) - \Delta_- f(x-h) = f(x) - 2f(x-h) + f(x-2h)\\ \Delta_\pm^2 f(x) &= \Delta_\pm f(x+h) - \Delta_\pm f(x-h) = f(x+2h) - 2f(x) + f(x-2h)\end{aligned} \tag{22.12}$$

The second derivative can now be estimated as

$$\begin{aligned}f''(x_0) &\approx \frac{\Delta_+^2 f(x_0)}{h^2} + \text{H.O.T.}\\ f''(x_0) &\approx \frac{\Delta_-^2 f(x_0)}{h^2} + \text{H.O.T.}\\ f''(x_0) &\approx \frac{\Delta_\pm^2 f(x_0)}{4h^2} + \text{H.O.T.}\end{aligned} \tag{22.13}$$

Note that all these calculations involve points x_0, x_0+h, x_0+2h, or x_0, x_0-h, x_0-2h, or x_0 , x_0+2h, x_0-2h, i.e., over the interval $x_0 \pm 2h$.

What is the order of approximations (22.13)? Let us check just Eq. $(22.13)_3$ (central second difference). The remaining two, you should check at home as an exercise.

$$
\begin{aligned}
f(x_0+2h) &\approx f(x_0)+f'(x_0)2h+\frac{1}{2}f''(x_0)4h^2+\frac{1}{6}f'''(x_0)8h^3+\frac{1}{24}f^{IV}(x_0)16h^4\\
f(x_0-2h) &\approx f(x_0)-f'(x_0)2h+\frac{1}{2}f''(x_0)4h^2-\frac{1}{6}f'''(x_0)8h^3+\frac{1}{24}f^{IV}(x_0)16h^4
\end{aligned}
\qquad (22.14)
$$

Adding the two equations (22.14), moving the term $2f(x_0)$ to the left hand side and dividing through $4h^2$ gives

$$\frac{f(x_0+2h)-2f(x_0)+f(x_0-2h)}{(2h)^2} \approx \frac{1}{3}f^{IV}(x_0)h^2 \qquad (22.15)$$

Thus the central second difference approximates the second derivative to order of h^2, i.e., it is a second order approximation. (You ought to anticipate the remaining two answers.)

There is one more way of constructing the central difference

$$\Delta^2 f(x) = \Delta_+ f(x) - \Delta_- f(x) = f(x+h) - 2f(x) + f(x-h) \qquad (22.16)$$

and the second derivative is approximated as

$$f''(x_0) \approx \frac{\Delta^2 f(x_0)}{h^2} = \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} + \text{H.O.T.} \qquad (22.17)$$

Order of the latter approximation is calculated directly by adding Eqs. (22.10) and solving the result for the second derivative.

Remark 29 At a cost of requiring a wider support, the central second finite difference provides a second order approximation of the second derivative. □

What happens when $x_0 - h < a$ or $x_0 + h > b$?

Remark 30 At end-points of the interval of interest, we need extra function values to the left and to the right. Now we must have a robust extrapolation algorithm. □

Remark 31 To calculate the first and the second derivative of smooth-enough functions, consider using cubic spline interpolation. If the function is positive, but has a very large variation, consider approximating its logarithm with cubic splines, and then transforming back:

$$
\begin{aligned}
z &= \ln y\\
z' &= \frac{y'}{y} \Rightarrow y' = yz'\\
z'' &= \frac{y''y - y'^2}{y^2} \Rightarrow y'' = z''y + \frac{y'^2}{y}
\end{aligned}
\qquad (22.18)
$$

□

Example 17 Suppose that you collected discrete amplitudes (diamonds) of a sensor at 25 different times between 1 and 25s, **Figure 22.3**. We want to approximate these 25 amplitudes with a smooth function, and calculate their first and second derivatives with respect to time (sensor velocity and acceleration) To calculate the derivatives, we will use the central differences of the cubic spline approximation of the sensor positions, **Figure 22.4 - 22.6**. The code is listed in **Appendix 22.5**. □

Example 18 Suppose that you want to approximate a wildly varying function and its first two derivatives. You can proceed as in the example above, through a combination of the cubic spline approximation of the function and the central difference derivatives, **Figure 22.7 - 22.8**. Or you can take the logarithm of the function, the logarithmic derivatives, Eq (22.18), and reconstruct the original function and its derivatives **Figure 22.9 - 22.10**. The code is listed in Section 22.5. □

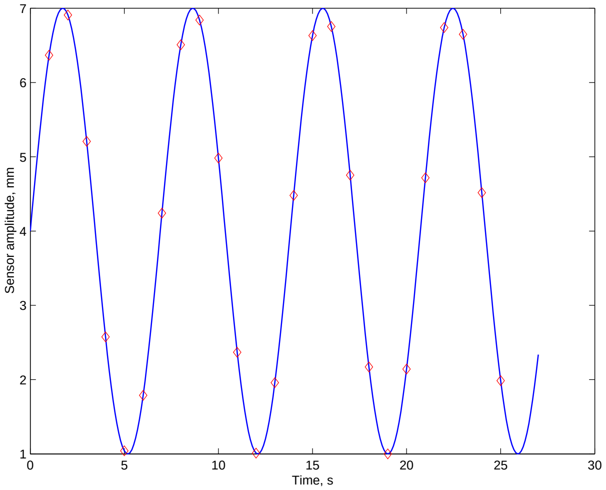


Figure 22.3: The diamonds are the measured sensor amplitudes. For convenience, I am showing you that these amplitudes belong to the curve (solid line) that extends beyond the measurements. Our task is to calculate the sensor velocity and acceleration. First we decide on the limits of approximation (interpolation and extrapolation if necessary). We pick time between 0 and 27 seconds. We also choose the interval of interpolation to be $h = 0.1$ s. The result of the spline interpolation is shown in **Figure 22.4**.

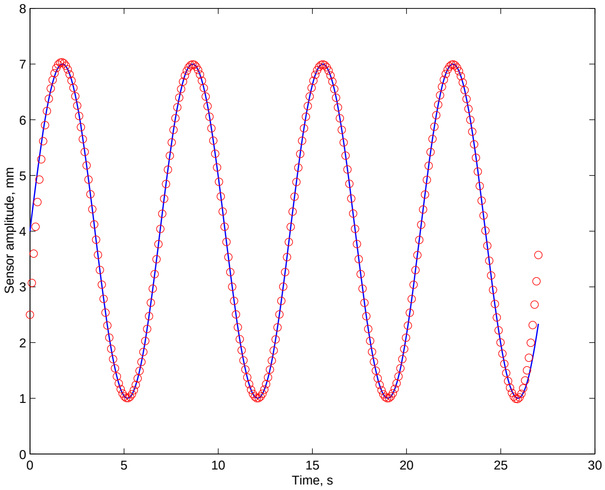


Figure 22.4: Interpolation and extrapolation of sensor positions using cubic splines (circles).

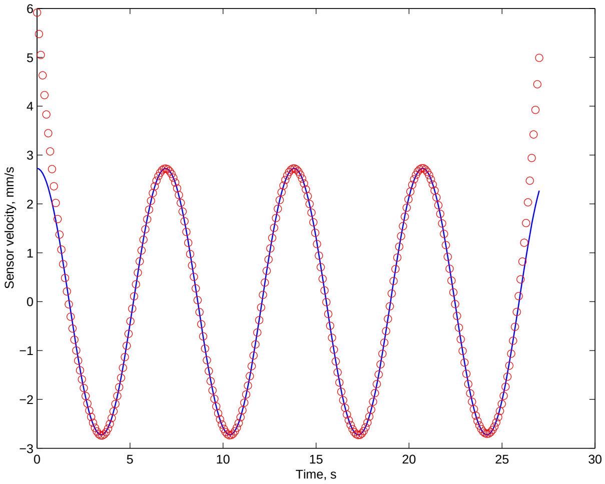


Figure 22.5: The exact first derivative (velocity) of sensor amplitude (solid line) and its numerical approximation by a central difference of the cubic spline approximation (circles).

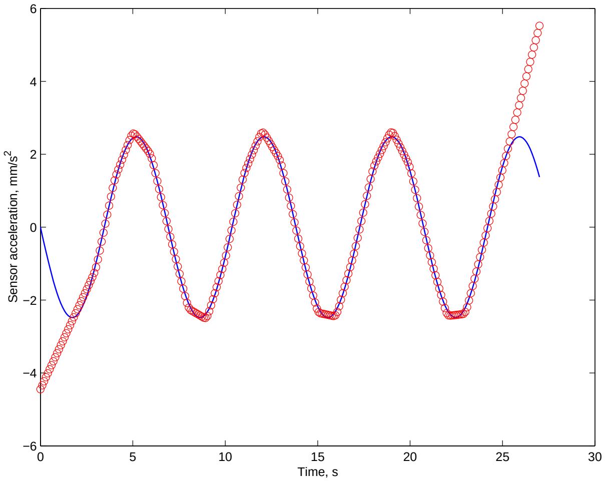


Figure 22.6: The exact second derivative (acceleration) of sensor amplitude (solid line) and its numerical approximation by a central difference of the cubic spline approximation (circles).

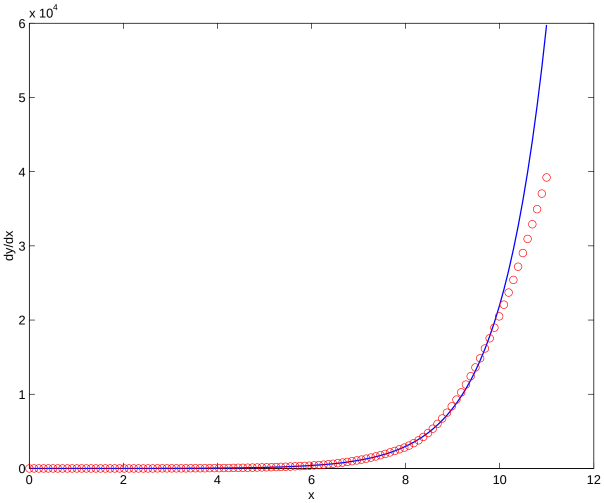


Figure 22.7: The first derivative of e^x using the cubic spline approximation and central difference.

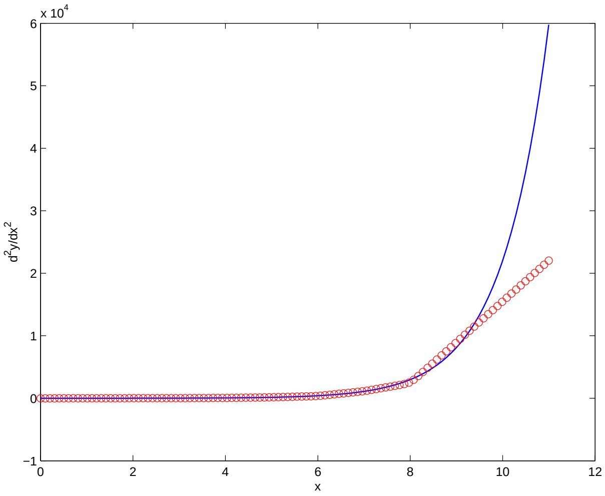


Figure 22.8: The second derivative of e^x using the he cubic spline approximation and central difference.

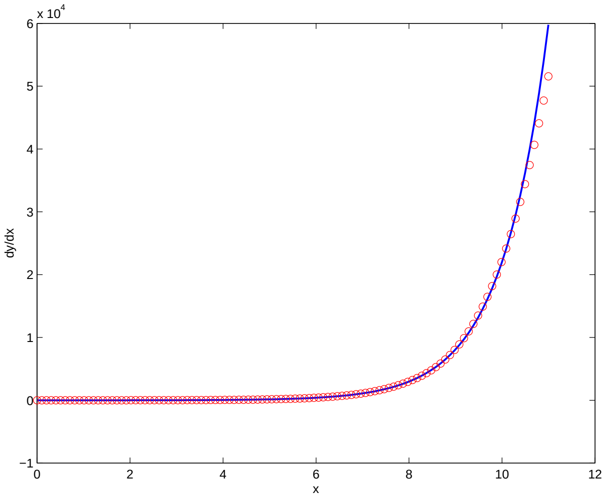


Figure 22.9: The first derivative of e^x by taking the logarithm and scaling back, Eq. (22.18).

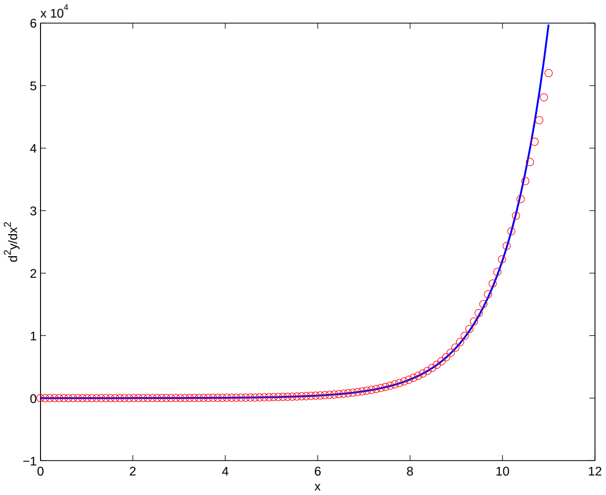


Figure 22.10: The second derivative of e^x by taking the logarithm and scaling back, Eq. (22.18).

22.5 Differentiation Demo

```
function DiffDemo()
%--------------------------------------------------------
% Demonstrates performance of cubic spline approximations
% of a periodic function and its first two derivatives
%
% Written by T. W. Patzek, March 12, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834,
% email: patzek@patzek.berkeley.edu
%
% Last revision 04/06/2002, TWP
% $Revision: 1.0 $  $Date: 2002.04.06 17:00:00 $
%--------------------------------------------------------
clear all, close all, clc
LW = 1;
PrintFlag = 0; %<<< Set to zero to see plot titles
% Discrete sensor amplitudes to be interpolated
% 1<=x<=25
% Times
xsensor  = 1:1:25;
vib(1,:) = xsensor;
% Amplitudes
vib(2,:) = 3*sin(xsensor/1.1) + 4;
% These discrete amplitudes belong to a smooth function
% Exact function beyond the sampling points
x = 0:0.1:27;
```

```
y = 3*sin(x/1.1) + 4;
% Exact first derivative
yprime = (3/1.1)*cos(x/1.1);
% Exact second derivative
ybis   = -(3/1.1/1.1)*sin(x/1.1);
% Calculate the derivatives from iterpolation,
% extrapolation and central differences
[xs,ys,yp,ypp] = derivatives(vib(1,:),vib(2,:),0,27,0.1);
% Plot the discrete sensor amplitudes and the function
% they represent

figure(3), hold on
plot(x,y,'-', 'linewidth',LW)
plot(vib(1,:), vib(2,:),'rd');
xlabel('Time, s'), ylabel('Sensor amplitude, mm'), box on
if (~PrintFlag)
    title('Discrete positions of 25 sensors')
else
    print -dpsc2 'E77N-Figure18.3.eps'
end

figure(4), hold on
plot(x,y,'-', 'linewidth',LW);
plot(xs, ys,'marker', 'o', 'color', 'r', 'linestyle', 'none')
xlabel('Time, s'), ylabel('Sensor amplitude, mm'), box on
if (~PrintFlag)
    title('Cubic spline interpolation')
else
    print -dpsc2 'E77N-Figure18.4.eps'
end

figure(5), hold on
plot(x,yprime,'-', 'linewidth',LW);
plot(xs, yp,'marker', 'o', 'color', 'r', 'linestyle', 'none')
xlabel('Time, s'), ylabel('Sensor velocity, mm/s'), box on
if (~PrintFlag)
    title('First derivative from CS and Central FD')
else
    print -dpsc2 'E77N-Figure18.5.eps'
end

figure(6), hold on
plot(x,ybis,'-', 'linewidth',LW);
plot(xs, ypp,'marker', 'o', 'color', 'r', 'linestyle', 'none')
xlabel('Time, s'), ylabel('Sensor acceleration, mm/s^2'), box on
if (~PrintFlag)
    title('Second derivative from CS and Central FD')
else
    print -dpsc2 'E77N-Figure18.6.eps'
end
```

```
%---------------------------------------------------------
%       Calculate the derivatives of e^x from splines and
%       from ln(e^x) and splines
%---------------------------------------------------------

x = 1:1:10;
y=exp(x);
[xs,ys,yp,ypp] = derivatives(x,y,0,11,0.1);

figure(7), hold on
plot(xs,exp(xs),’-’, ’linewidth’,LW);
plot(xs, yp,’marker’, ’o’, ’color’, ’r’, ’linestyle’, ’none’)
xlabel(’x’), ylabel(’dy/dx’), box on
if (~PrintFlag)
    title(’First derivative of e^x’)
else
    print -dpsc2 ’E77N-Figure18.7.eps’
end

figure(8), hold on
plot(xs,exp(xs),’-’, ’linewidth’,LW);
plot(xs, ypp,’marker’, ’o’, ’color’, ’r’, ’linestyle’, ’none’)
xlabel(’x’), ylabel(’d^2y/dx^2’), box on
if (~PrintFlag)
    title(’Second derivative of e^x’)
else
    print -dpsc2 ’E77N-Figure18.8.eps’
end
% Do the logarithmic derivative
z=log(y);
[xs,zs,zp,zpp] = derivatives(x,z,0,11,0.1);
% Transform back to the original derivatives
yp=zp.*ys;
ypp = zpp.*ys+yp.^2./ys;

figure(9), hold on
plot(xs,exp(xs),’-’, ’linewidth’,1.5);
plot(xs, yp,’marker’, ’o’, ’color’, ’r’, ’linestyle’, ’none’)
xlabel(’x’), ylabel(’dy/dx’), box on
if (~PrintFlag)
    title(’First derivative from z=ln(e^x)’)
else
    print -dpsc2 ’E77N-Figure18.9.eps’
end

figure(10), hold on
plot(xs,exp(xs),’-’, ’linewidth’,1.5);
plot(xs, ypp,’marker’, ’o’, ’color’, ’r’, ’linestyle’, ’none’)
xlabel(’x’), ylabel(’d^2y/dx^2’), box on
if (~PrintFlag)
```

```
    title(’Second derivative of z=ln(e^x)’)
else
    print -dpsc2 ’E77N-Figure18.10.eps’
end

function [x,y,yp,ypp] =  derivatives(xdata,ydata,xmin,xmax,h)
%---------------------------------------------------------
% A function that
%    1.
%Approximates the input data with cubic splines over
%       the interval [xmin,xmax] that can extend beyond
%       the interval min(xdata), max(xdata).
%    2. Calculates the first and second derivative of the
%       function using central differences of the interpo-
%       lated data
%   Input arguments:
%       xdata, ydata = vector of raw function data
%       xmin,xmax    = interval over which the function
%                      is interpolated and extrapolated
%       h            = interpolation interval
%
%   Output arguments
%       x   = vector of x’s for the derivatives
%       y   = vector of y’s for the derivatives
%       yp  = first derivative
%       ypp = second derivative
%
% Written by T. W. Patzek, March 13, 2001
% UC Berkeley, 425 Davis Hall, Berkeley, CA 94720
% Tel: 510-43-5834
% email: patzek@patzek.berkeley.edu
%
% Last revision 04/06/2002, TWP
% $Revision: 1.0 $  $Date: 2002.04.16 17:00:00 $
%-----------------------------------------------------
%                    PRELIMINARIES
% Intialize outputs
x   =[];
yp  =[];
ypp =[];

if     nargin < 1
    error(’You must input xdata’)
elseif nargin < 2
    error(’You must input ydata’)
end

% Make both arrays into row vectors, regardless of their initial
% shape
[mx,nx] = size(xdata); lx = mx*nx; xdata = reshape(xdata,1,lx);
```

```
[my,ny] = size(ydata); ly = my*ny; ydata = reshape(ydata,1,ly);

if (lx ~=ly)
    error('Arrays xdata and ydata must have the ame sizes')
end

% if necessary, sort xdata to make them increasing
if any(diff(xdata)<0)
   [xdata,ix]=sort(xdata);
   ydata = ydata(ix);
end

if nargin < 3
    xmin = min(xdata);
    xmax = max(xdata);
    h=(xmax-xmin)/100;
elseif nargin < 4
    xmax = max(xdata);
    h=(xmax-xmin)/100;
elseif nargin < 5
    h=(xmax-xmin)/100;
end
%------------------------------------------------------
%                       START THE CODE
%------------------------------------------------------
% Go 2*h beyond the desired interval to calculate the
% second derivative at both endpoints

xsmin = xmin-2*h;
xsmax = xmax+2*h;

n  = fix((xsmax-xsmin)/h + 0.5);
xs = linspace(xsmin,xsmax,n);
ys = spline(xdata,ydata,xs);

% First derivative
div = 0.5/h;
for i=3:n-2
    yp(end+1)=div*(ys(i+1)-ys(i-1));
end
% Second derivative
div = div*div;
for i=3:n-2
    ypp(end+1)=div*(ys(i+2)-2*ys(i)+ys(i-2));
end
% The function approximation
y = ys(3:n-2);
x = xs(3:n-2);
```

Chapter 23

Numerical Integration

23.1 What Are You Going To Learn?

You will learn about a hierarchy of simple quadrature formulae that employ interpolation polynomials. In this lecture, I will follow very closely Chapter 4 of the *Numerical Recipes in C* [28].

In Chapter 23 folder on fttp:\\petroleum.berkeley.edu, look up the following MATLAB files:

Romberg.m integrates the function pointed to by `fhandle` from $x = a$ to $x = b$, and produces a triangular ROMBERG table, T, of results.

RombergF.m Put here your own function to integrate. The current example function is $y = e^x$.

RombergDemo.m demonstrates performance of ROMBERG's numerical integration algorithm described in `Romberg.m`. The `Romberg` function is about 20 times faster than the MATLAB `quadl` function.

23.2 MATLAB Shortcut

MATLAB has a built in function `quad` which implements an adaptive SIMPSON's scheme, and `quadl` which implements a high order method using an adaptive GAUSS/LOBATTO quadrature rule. Please check MATLAB help for details.

23.3 Why Is It Important?

In many practical problems, the function we should integrate is known only approximately and its indefinite integral is unknown. In these situations, we must integrate numerically, and we want to be confident that the numerical integral is reasonably close to the real one.

23.4 Introduction

Explicit formulae cannot be given for the indefinite integrals of most functions. Furthermore, in many problems the integrand, $f(x)$, is not known precisely but given by a series of discrete points (e.g., a time series), or is the solution of a differential equation, which cannot be solved explicitly.

Thus we shall seek appropriate numerical procedures to approximate the value of the definite integral, say

$$I\{f\} \equiv \int\limits_a^b f(x)dx \tag{23.1}$$

Here $[a, b]$ is a finite closed interval, and Eq. (23.1) should be read in words as "the operator I acting on all values of a function f is defined as a definite integral of f." The geometrical interpretation is, as you know, the area under the curve on the interval $[a, b]$ (see Figure 23.1).

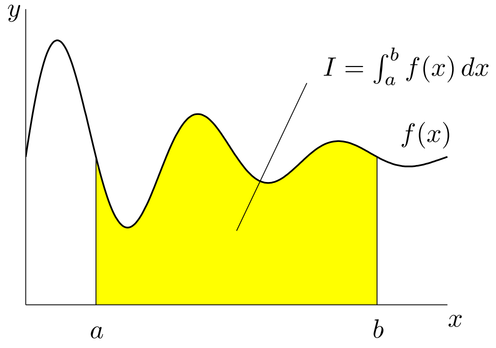


Figure 23.1: Geometrical interpretation of the integral $\int_a^b f(x)\,dx$

The types of approximation to Eq. (23.1) we consider here are all essentially of the form

$$I_n\{f\} \equiv \sum_{j=1}^{n} a_j f(x_j) \tag{23.2}$$

The sum in Eq. (23.2) is called a numerical quadrature or numerical integration formula. The n distinct points, $a \le x_j \le b$, are called the quadrature points or nodes and the coefficients a_j are called quadrature coefficients.

The basic problems in numerical integration are in choosing the nodes and coefficients so that $I_n\{f\}$ is a "close" approximation of $I\{f(x)\}$. As with polynomial approximation, different criteria can be used to measure the quadrature error,

$$E_n\{f\} \equiv I\{f) - I_n\{f\} \tag{23.3}$$

One particularly useful measure of the quadrature error is the so-called degree of precision, defined as the maximum integer m such that $E_n\{x^k\} = 0$ for

$k = 1, 2, ..., m$, but $E_n\{x^{m+1}\} \neq 0$. Thus if a quadrature formula has degree of precision m, all polynomials of degree at most m will be integrated exactly with this formula.

If a "close" approximation to $f(x)$ in $a \leq x \leq b$ is known, then the integral of the approximating function will be "close" to the integral of $f(x)$. In other words, if

$$\begin{gathered} |f(x) - g(x)| \leq \varepsilon \\ \left| \int_a^b f(x)dx - \int_a^b g(x)dx \right| \leq |b-a|\varepsilon \end{gathered} \tag{23.4}$$

This simple result is the motivation for developing most numerical integration methods. Of course, it is desirable that the approximating function has a simple explicit integral. Hence polynomial approximations are naturally suggested, and of these interpolation polynomials are most frequently used. All practical quadrature methods are *interpolatory.*

23.5 Closed Newton-Cotes Formulæ

First some notation: we have a sequence of abscissas, $x_0, x_1, ..., x_N, x_{N+1}$ which are spaced apart by a constant step h,

$$x_i = x_0 + ih, \quad i = 0, 1, ..., N+1 \tag{23.5}$$

A function $f(x)$ has known values at the x_i's,

$$f(x_i) \equiv f_i \tag{23.6}$$

Closed quadrature formulæuse points x_0 and x_{N+1}, while open formulædo not. In the subsequent sections, we will use a somewhat different notation, starting from x_1 and ending with x_N for closed intervals. This change in notation should not lead to confusion. Now we are ready to review several simple integration methods, which employ increasingly more interior points and approximate the true integral to increasingly higher order.

23.5.1 The Trapezoidal Rule

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'') \tag{23.7}$$

Here the error term $O(\ )$ signifies that the true answer differs form the estimate by an amount that is the product of some numerical coefficient times h^3 times the value of the function's second derivative somewhere in the interval of integration. The coefficient is known, and it can be found in all standard texts on numerical analysis. The point at which the second derivative should be evaluated is not known. If it were known, we would use this knowledge and approximate the integral to a higher order. Since a product of a known and an unknown is an unknown, we streamline the notation and write only $O(\;)$ instead of the full formula. The trapezoidal rule in Eq. (23.7) is a "single panel" algorithm. Equation (23.7) approximates the function with a straight line, i.e., it is exact up to and including a polynomial of degree one, or $f(x) = x$.

23.5.2 Simpson's Rule

One may anticipate that there exists a three-point formula exact up to polynomials of degree 2. This is true, and lucky error cancellation makes it exact up to and including order 3, i.e., $f(x) = x^3$ is integrated exactly.

$$\int\limits_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \qquad (23.8)$$

Note that this formula produces the integral over two intervals of size h each. Hence, the coefficients of Eq. (23.8) add up to 2, i.e., we have two "panels".

There is no lucky error cancellation in the four-point formula, so it is exact for polynomials up to and including degree 2. This is SIMPSON's 3/8 rule:

$$\int\limits_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4\right] + O(h^5 f^{(4)}) \qquad (23.9)$$

How many panels do we have here?

23.5.3 Boole's Rule

The five-point formula benefits from luck again, and it is exact for polynomials up to and including degree 5:

$$\int\limits_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] + O(h^7 f^{(6)}) \qquad (23.10)$$

How many panels are present here? At this point, the quadrature formulae stop being named after famous persons, so we stop too.

23.6 Extended Formulae (Closed)

If we use Eq. (23.7) $N-1$ times, to do the integration in the intervals $[x_1, x_2]$, $[x_2, x_3]$, ..., $[x_{N-1}, x_N]$, and then add the results, we obtain an "extended", or "composite" formula for the integral from x_1 to x_N.

23.6.1 Extended Trapezoidal Rule

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + ... + f_{N-1} + \frac{1}{2}f_N\right] + O(Nh^3 f'') \qquad (23.11)$$

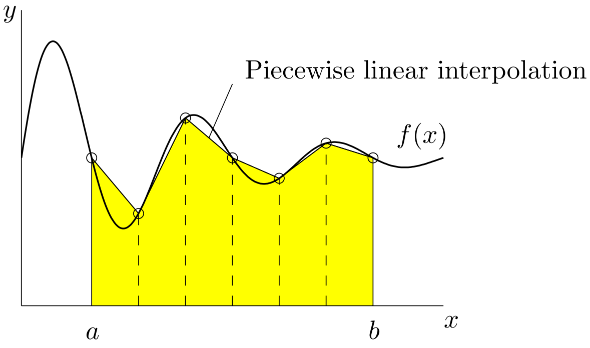


Figure 23.2: Geometrical interpretation of the trapezoidal rule.

The error term may be rewritten as follows

$$O(Nh^3 f'') = O\left(\frac{(x_N - x_1)^3}{N^2} f''\right) \qquad (23.12)$$

The latter notation is clearer, because we usually keep the endpoints fixed and want to know how much the error will decrease if we double the number of points (it will decrease 4 times).

23.6.2 Extended Simpson's Rule

If we apply Eq. (23.7) to successive non-overlapping pairs of intervals, we obtain the extend SIMPSON's rule:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\begin{array}{l} \frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + ... \\ +\frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N \end{array}\right] + O\left(\frac{(x_N - x_1)^5}{N^4} f^{(4)}\right) \qquad (23.13)$$

Note the alternating coefficients 4/3 and 2/3.

23.7 Romberg Integration

The algorithms in the previous section can be unified with new notation, T_k^m, where the subscript k labels the order of the approximation (the number of

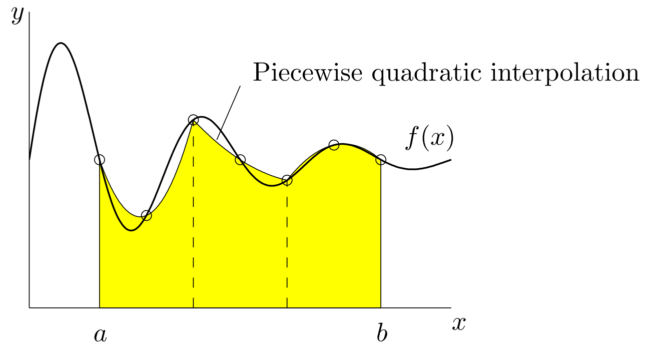


Figure 23.3: Geometrical interpretation of SIMPSON's rule.

panels equals 2^k), and the superscript m identifies the level of integration algorithm:

$$\begin{array}{ccc} m=0 & \text{Trapezoidal rule} & T_k^0 \\ m=1 & \text{Simpsons' rule} & T_k^1 \\ m=2 & \text{Boole's rule} & T_k^2 \end{array} \tag{23.14}$$

etc. The equations of the previous section can be rewritten in terms of this new notation:

$$\begin{aligned} T_k^1 &= T_k^0 + \frac{1}{3}\left(T_k^0 - T_{k-1}^0\right) \\ T_k^2 &= T_k^1 + \frac{1}{15}\left(T_k^1 - T_{k-1}^1\right) \\ T_k^3 &= T_k^2 + \frac{1}{63}\left(T_k^2 - T_{k-1}^2\right) \end{aligned} \tag{23.15}$$

The generalization of these results leads to the equation for the ROMBERG algorithm:

$$T_k^{m+1} = T_k^m + \frac{1}{4^{m+1}-1}\left(T_k^m - T_{k-1}^m\right) \tag{23.16}$$

This algorithm usually gives much more accurate results for sufficiently smooth functions with little more work than required in trapezoidal rule quadrature. Of the extended trapezoidal rule, the ROMBERG algorithm removes all terms in the error series up to but not including $O(1/N^{2k})$. ROMBERG's algorithm is a good example of a very general idea that has the name of RICHARDSON's deferred approach to the limit: Perform some numerical algorithm for several decreasing values of a parameter h, and then extrapolate to the continuum limit $h = 0$. For more information, please look up [28] and [1].

Chapter 24

Ordinary Differential Equations

24.1 What Are You Going To Learn?

In this chapter you will learn how to integrate the ordinary differential equations of first order subject to some initial conditions with:

- EULER's forward method
- Variable step RUNGE-KUTTA method
- MATLAB built-in `ode45` function

EULER's forward method relies on a forward difference approximation to a derivative. EULER's method is the most elementary numerical scheme of integration, but it is only first-order accurate and may require very small time steps.

RUNGE-KUTTA's methods give more accurate calculation of the dependent variable at the next desired time level by calculating several intermediate values of the right-hand-side of the equation.

Chapter 24 folder on fttp:\\petroleum.berkeley.edu has the following files:

BaseballDemo.m demonstrates performance of the forward EULER method, the 4^{th} order **Runge-Kutta** method, and the default MATLAB ODE integration function `ode45`.

FwdEuler.m implements the forward EULER method with a constant step.

rk4.m is the fourth order RUNGE-KUTTA method with a constant step.

LotkaVolterra.m integrates the LOTKA-VOLTERRA population balance of predators and prey, and plots the results.

RHSfun.m is the right hand side of the system of ODE's to be integrated with any of the integration schemes above.

RHSLV.m is the right hand side of the LOTKA-VOLTERRA system.

24.2 Why Is It Important?

As you will see from the quotation later, the material of this lecture covers pretty much what most engineers know about solving numerically first order nonlinear ordinary differential equations. If you acquaint yourself with the example problems and the MATLAB code samples below, you will be almost as good as most practicing engineers. You will also recognize the need to debug your numerical integration schemes against analytical solutions of test problems. Otherwise you will never know how good your numerical integration routine is.

24.3 MATLAB Shortcut

MATLAB's `ode45` function solves non-stiff ordinary differential equations; other functions exist, **Table 24.1**. You can improve accuracy of `ode45` by changing its default parameters with the `odeset` function.

Table 24.1: MATLAB functions which integrate ordinary differential equations.

—Solver	Problem Type	Order of Accuracy	When to Use
—ode45	Non-stiff	Medium	Most of the time. This should be the first solver you try.
—ode23	Non-stiff	Low	If using crude error tolerances or solving moderately stiff problems.
—ode113	Non-stiff	Low to high	If using stringent error tolerances or solving a computationally intensive ODE file.
—ode15s	Stiff	Low to medium	If ode45 is slow because the problem is stiff.
—ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
—ode23t	Moderately Stiff	Low	If the problem is only moderately stiff and you need a solution without numerical damping.
—ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

24.4 Conversion of ODEs into the standard MATLAB form

Suppose that the problem we solve results in a third order ordinary differential equation (ODE) of the form:

$$y''' \equiv \frac{\mathrm{d}^3 y}{\mathrm{d}x^3} = f(x, y, y', y'') \tag{24.1}$$

with the initial conditions

$$y(0) = y_{10}, \qquad y'(0) = y_{20}, \qquad y''(0) = y_{10} \tag{24.2}$$

By defining the new variables

$$\begin{aligned} y_1 &:= y \\ y_2 &:= y' \\ y_3 &:= y'' \end{aligned} \tag{24.3}$$

we can write

$$\begin{aligned} \frac{\mathrm{d}y_1}{\mathrm{d}x} &= y_2 \\ \frac{\mathrm{d}y_2}{\mathrm{d}x} &= y_3 \\ \frac{\mathrm{d}y_3}{\mathrm{d}x} &= f(x, y_1, y_2, y_3) \end{aligned} \tag{24.4}$$

or

$$\begin{aligned} Y' &= F(x, Y) \\ Y(0) &= Y_0 \end{aligned} \tag{24.5}$$

where

$$\begin{aligned} Y &:= \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \\ F &:= \begin{pmatrix} y_2 \\ y_3 \\ f_(x, Y) \end{pmatrix} \\ Y_0 &:= \begin{pmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{pmatrix} \end{aligned} \tag{24.6}$$

24.5 Simple Example: Throwing a Ball

Suppose that at time $t = 0$, I throw upwards a base ball at an angle α with respect to horizontal. Initial velocity of the ball is $\boldsymbol{v_0}$, and the velocity magnitude is v_0. We want to determine the trajectory of the ball. The problem can be formulated mathematically in a Cartesian coordinate system with me at standing at $x = 0$ and the $y-$axis pointing vertically upwards:

$$\begin{aligned} v_x &= v_0 \cos\alpha && (24.7) \\ v_y &= v_0 \sin\alpha - gt && (24.8) \end{aligned}$$

We remember that the distance travelled by the ball in the $x-$ and $y-$direction is

$$\begin{aligned} \frac{\mathrm{d}s_x}{\mathrm{d}t} &= v_x \\ \frac{\mathrm{d}s_y}{\mathrm{d}t} &= v_y \\ s_x(0) &= 0,\ s_y(0) = y_0 \end{aligned} \qquad (24.9)$$

24.5.1 Euler's Forward Method

Equations (24.9) form a system of two (here independent) *Ordinary Differential Equations*, abbreviated as ODE's, with appropriate initial conditions (y_0 is my height). These two equations can be integrated directly with respect to time and the results is

$$\begin{aligned} s_x(t) &= (v_0 \cos\alpha)t \\ s_y(t) &= y_0 + (v_0 \sin\alpha)t - \frac{1}{2}gt^2 \end{aligned} \qquad (24.10)$$

So here we know the answer exactly, and we will try to obtain the same answer numerically using the simplest numerical integration scheme, EULER's Forward Method.

For us, an ODE involves a single unknown function, e.g., $y(t)$, (here $y = s_x$ and $y = s_y$) and its derivatives, e.g., $y\,'(t)$, $y\,''(t)$, ... An ODE equation alone is not enough to determine the value of y for all times: we need an *initial condition*. An ODE, together with an initial condition, constitutes an *Initial*

Value Problem (IVP):

$$\begin{pmatrix} y_1' \\ y_2' \\ \vdots \\ y_{n-1}' \\ y_n' \end{pmatrix} = \begin{pmatrix} f_1(t, y_1, \ldots, y_n) \\ f_2(t, y_1, \ldots, y_n) \\ \vdots \\ f_{n-1}(t, y_1, \ldots, y_n) \\ f_n(t, y_1, \ldots, y_n) \end{pmatrix}$$

$$\begin{pmatrix} y_1(0) \\ y_2(0) \\ \vdots \\ y_{n-1}(0) \\ y_n(0) \end{pmatrix} = \begin{pmatrix} y_{10} \\ y_{20} \\ \vdots \\ y_{(n-1)0} \\ y_{n0} \end{pmatrix}$$

or, in vector notation,

$$\begin{aligned} y\,' &= f(t, y) \\ y(0) &= y_0, \end{aligned}$$

where f is a function of both time t and the unknown y. (Here the function f is simply $v_x(t)$ or $v_y(t)$, so there is no dependence on the unknowns s_x or s_y.)

The forward EULER's method attempts to compute the value of y at time t_{k+1} given its value at time t_k. Note that we can always start the process since we know the initial value y_0 at $t = 0$. The idea is to use a finite difference approximation of y', and to evaluate $f(t, y)$ at the current time step t_i:

$$y_i' \approx \frac{y_{i+1} - y_i}{\Delta t} = f(t_i, y_i).$$

Solving for y_{i+1}, we get the forward Euler method:

$$\boxed{y_{i+1} = y_i + f(t_i, y_i)\Delta t.}$$

The code is listed in Section 24.7.

24.5.2 Runge-Kutta Method

The EULER method formula is

$$y_{i+1} = y_i + hf(t_i, y_i). \tag{24.11}$$

which advances a solution from t_i to $t_{i+1} \equiv t_i + h$. This formula is asymmetrical: it advances the solution through an interval h, but uses derivative information only at the beginning of that interval. Therefore (please verify it as an exercise by expansion in TAYLOR series) the approximation error is only one power of h higher (i.e., it is smaller) than the correction on the right side of Eq. (24.11). In other words, the error is $\mathcal{O}(h^2)$.

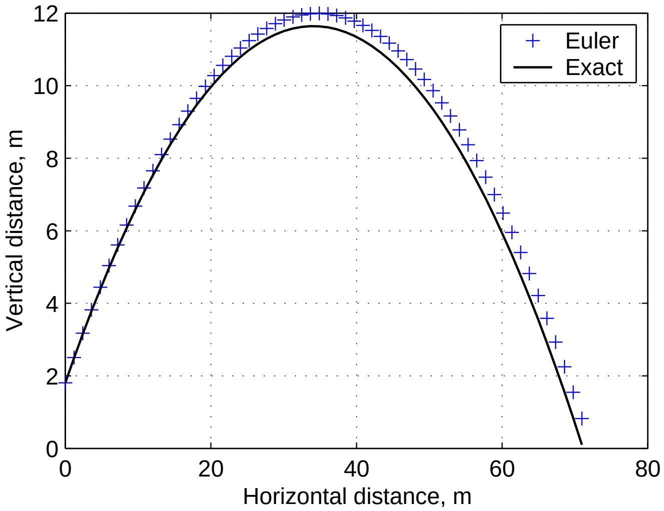


Figure 24.1: Plot of the baseball trajectory for $v_0 = 100$ km/h, $\alpha = 30$ deg.

There are several reasons why we do not recommend Euler's method for practical use: (1) the method is not very accurate when compared with other methods that run at equivalent step size, and (2) it is not very stable either.

Consider, however, taking step like (24.11) to take a "trial" step to the middle of the interval. Then use the values of both t and y at that midpoint to compute the "real" step across the whole interval. The corresponding mathematical expressions are:

$$\begin{aligned} k_1 &= hf(t_i, y_i) \\ k_2 &= hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \\ y_{i+1} &= y_i + k_2 + \mathcal{O}(h^3) \end{aligned} \tag{24.12}$$

As the error term indicates, this symmetrization cancels out the first order error term, making the method *second order*. Remember that a numerical method is called nth order if its remainder term is $\mathcal{O}(h^{n+1})$. In fact, (24.12) is called the *second-order* Runge-Kutta or *midpoint* method.

We can continue this trial-correction game to eliminate the error terms order by order. This is the basic idea behind the Runge-Kutta method. By far the most used is the classical fourth-order Runge-Kutta formula, which is very slick:

$$\begin{aligned}
k_1 &= hf(t_i, y_i) \\
k_2 &= hf\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1\right) \\
k_3 &= hf\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2\right) \\
k_4 &= hf(t_i + h, y_i + k_3) \\
y_{i+1} &= y_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(h^5) \qquad (24.13)
\end{aligned}$$

```
function [t,y] = rk4(RHSfunp,ICs,tmin,tmax,h,v0,alpha)
% Fourth order Runge-Kutta method with a constant timestep h
% Input arguments:
%   RHSfunp = pointer to the RHS function, here in m/s
%   IC's    = row vector of initial conditions, here in m
%   tmin    = starting time, s
%   tmax    = end time,s
%   h       = time step, s
%   v0      = initial velocity in m/s (problem specific parameter)
%   alpha   = angle in radians (problem specific parameter)
% Output arguments:
%   t       = vector of integration times
%   y       = array of rows of integrated function values
%             (here distances)
% TWP, E77N, April 4, 2001
%-----------------------------------------------------------------

% Number of integration steps
nstep  = (tmax-tmin)/h;
%
% Initialization
t(1)   = tmin;
y(1,:) = ICs;
hh     = 0.5*h;
%
% Time loop with the fourth order R-K inside
for istep = 1:nstep
    f1 = feval(RHSfunp,t(istep),y(istep,:),v0,alpha);
    k1 = h*f1';
    f2 = feval(RHSfunp,t(istep)+hh,y(istep,:)+0.5*k1,v0,alpha);
    k2 = h*f2';
    f3 = feval(RHSfunp,t(istep)+hh,y(istep,:)+0.5*k2,v0,alpha);
    k3 = h*f3';
    f4 = feval(RHSfunp,t(istep)+h,y(istep,:)+k3,v0,alpha);
    k4 = h*f4';
    y(istep+1,:) = y(istep,:) +(k1/6+k2/3+k3/3+k4/6);
    t(istep+1)   = tmin + istep*h;
end
```

```
return
```

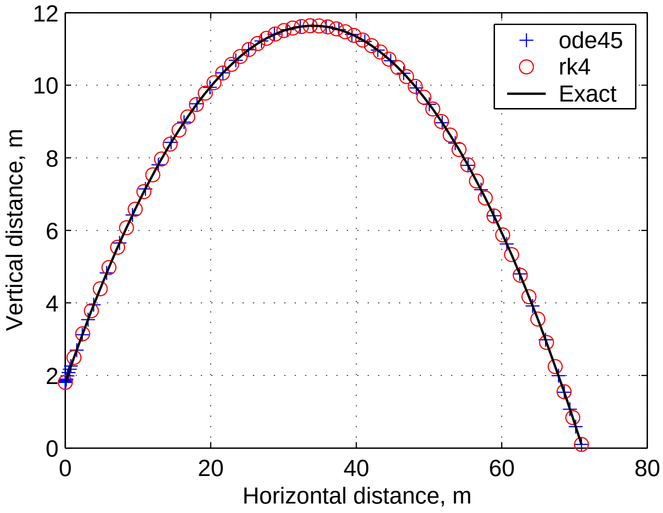


Figure 24.2: Plot of the baseball trajectory for $v_0 = 100$ km/h, $\alpha = 30$ deg. This example uses the MATLAB built in function `ode45.m` and our own function `rk4.m`

24.6 Predator-Prey Processes

The earliest representations of predator-prey behavior were constructed independently by Lotka (1925) and Volterra (1926). Although their point of departure was identical, Volterra probed far more deeply. He discussed several possible cases, the simplest being that of two associated species. One species finding sufficient food in the environment, would multiply indefinitely when left to itself, whilst the other would perish for lack of food if left alone; but the second feeds upon the first, and so the two species can coexist together.

Denote

$$
\begin{aligned}
N_1(t) &= \text{number of prey (or hosts)}\\
N_2(t) &= \text{number of predators (or parasites)}
\end{aligned}
$$

at time t. Suppose that in the absence of predators prey increase at rate r_1, whilst in the absence of prey predators die at rate r_2. The the simplest deterministic description of coexistence of the two species is

$$
\begin{aligned}
\frac{\mathrm{d}N_1}{\mathrm{d}t} &= N_1(r_1 - b_1 N_2)\\
\frac{\mathrm{d}N_2}{\mathrm{d}t} &= N_2(-r_2 + b_2 N_1) \qquad (24.14)
\end{aligned}
$$

Let us demonstrate this process with the following particular example:

$$\begin{aligned}\frac{\mathrm{d}N_1}{\mathrm{d}t} &= N_1(1.5 - 0.1N_2)\\ \frac{\mathrm{d}N_2}{\mathrm{d}t} &= N_2(-0.25 + 0.01N_1) \end{aligned} \tag{24.15}$$

With the initial conditions

$$\begin{aligned} N_1(0) &= 1, 5, 10, 15, 20\\ N_2(0) &= 15 \end{aligned} \tag{24.16}$$

The code is listed in Section 24.8.

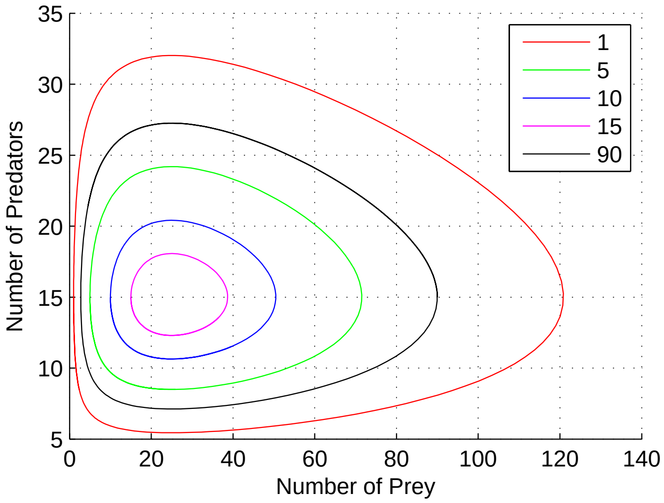


Figure 24.3: Trajectories of prey and predator populations. This example uses the MATLAB built in function `ode45.m` and our own function `LotkaVolterra.m`

For many scientific users, fourth-order Runge-Kutta method is not just the first word on ODE integrators, but the last word as well. Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Bulirsch-Stoer predictor-corrector methods can be very much more efficient in problems where very high accuracy is a requirement.

— WILLIAM H. PRESS *et al.*, *Numerical Recipes* (1992)

24.7 MATLAB Code for the Baseball Problem

Here we give you the EULER forward method implemented for the particular baseball problem (`FwdEuler.m`), the right hand side function (RHSfun.m), and the demo function that calls the EULER integration routine and performs plotting (`BaseballDemo.m`).

BaseballDemo.m

```
function BaseballDemo()
% Demonstrates performance of the forward Euler method,
% the 4th order Runge-Kutta method, and the default MATLAB
% ODE integration function ode45.
% TWP, E77N, April 4, 2001
clear all; close all; clc
%
% Zero initial conditions
ICs = [0,1.81];
%
% My time interval and time step
tmin=0; tmax=2.95; h=0.05;
%
% Angle in radians
alpha = 30*pi/180;
% Initial velocity in m/s, assuming that I am a famous pitcher
v0 = 100*1000/3600;
%
```

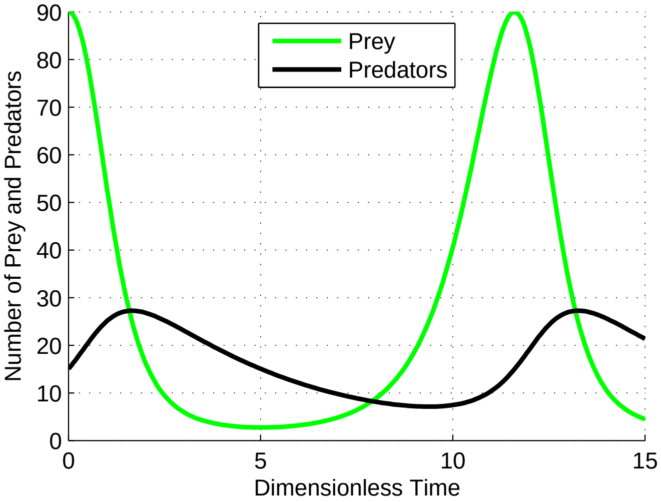


Figure 24.4: History of prey and predator populations for $N_1(0) = 20$, $N_2(0) = 15$. This example uses the MATLAB built in function `ode45.m` and our own function `LotkaVolterra.m`

```
% Forward Euler method with a constant timestep h
[t,y] = FwdEuler(@RHSfun,ICs,tmin,tmax,h,v0,alpha);
%
% Comparison with direct integration
t = tmin:h:tmax;
xd = ICs(1)+v0*cos(alpha)*t;
yd = ICs(2)+v0*sin(alpha)*t-9.81/2*t.^2;
%
figure(1)
hold on
plot(y(:,1),y(:,2),...
    'color','b','marker','+','linestyle','none');
plot(xd,yd,'color','k','linewidth',1);
grid on, box on
xlabel('Horizontal distance, m');
ylabel('Vertical distance, m');
legend('Euler','Exact')
set(gcf,'PaperPosition',[1 1 4.5 3.2])
%print -depsc2 baseball1.eps
%---------------------------------------------------------------------------
%Comparison with our Runge-Kutta method<<<<<<<<<<<
[tRK4,yRK4] = rk4(@RHSfun,ICs,tmin,tmax,h,v0,alpha);
%---------------------------------------------------------------------------
%Comparison with MATLAB Runge-Kutta method<<<<<<<
[tRK45,yRK45] = ode45(@RHSfun,[tmin,tmax],ICs,[],v0,alpha);
%
figure(2); hold on
plot(yRK45(:,1),yRK45(:,2),...
    'color','b','marker','+','linestyle','none');
plot(yRK4(:,1),yRK4(:,2),...
    'color','r','marker','o','linestyle','none');
plot(xd,yd,'color','k','linewidth',1);
grid on, box on
xlabel('Horizontal distance, m');
ylabel('Vertical distance, m');
legend('ode45','rk4','Exact')
set(gcf,'PaperPosition',[1 1 4.5 3.2])
%print -depsc2 baseball2.eps
```

FwdEuler.m

```
function [t,y] = FwdEuler(RHSfunp,ICs,tmin,tmax,h,v0,alpha)
% Forward Euler method with a constant timestep h
% Input arguments:
%   RHSfunp = pointer to the RHS function, here in m/s
%   IC's    = row vector of initial conditions, here in m
%   tmin    = starting time, s
%   tmax    = end time,s
%   h       = time step, s
%   v0      = initial velocity in m/s (problem specific parameter)
%   alpha   = angle in radians (problem specific parameter)
% Output arguments:
%   t       = vector of integration times
%   y       = array of rows of integrated function values
%             (here distances)
%
% TWP, E77N, April 4, 2001
%-----------------------------------------------------------------

% Number of integration steps
nstep  = (tmax-tmin)/h;
%
% Initialization
t(1)   = tmin;
y(1,:) = ICs;
%
% Euler loop
for istep = 1:nstep
    f = feval(RHSfunp,t(istep),y(istep,:),v0,alpha);
    y(istep+1,:) = y(istep,:) + f'*h;
    t(istep+1)   = tmin + istep*h;
end
return
```

RHSfun.m

```
function [f] = RHSfun(t,y, v0,alpha)
% The right hand sides of the system of ODE's
% to be coded by you for each problem as a >>COLUMN<< vector
% TWP, E77N, April 4, 2001
% -------------------------------------------------------------------------
% the RHS function MUST return a >>COLUMN<< vector
f = zeros(2,1); % We  have 2 equations
v_x = v0*cos(alpha);
v_y = v0*sin(alpha)-9.81*t;
f(1,1) = v_x;
f(2,1) = v_y;
```

24.8 MATLAB Code for the L-V Equation

```
function LotkaVolterra()
% Integrates the Lotka-Volterra population balance of
% predators and prey.
% TWP, E77, April 4, 2001
% Last modified, TWP 2006-04-10
clear all; close all; clc
PrintFlag = 1;
%
% Initial conditions
ICs = [1,5,10,15,90;
```

```
         15,15,15,15,15];
% My time interval
tmin=0; tmax=15;
% Parameters of Lotka-Volterra equations
r1=1.5; r2=0.25; b1=0.1; b2=0.01;
%
% MATLAB Runge-Kutta method with variable time step
figure(1); hold on
cs = ['r','g','b','m','k'];
options = odeset('Refine',4,'RelTol',1e-6);
for i=1:length(ICs)
    [tRK45,yRK45] = ode45(@RHSLV,[tmin,tmax],ICs(:,i),...
        options, r1,r2,b1,b2);
    plot(yRK45(:,1),yRK45(:,2),'color',cs(i),'marker',...
        'none','linestyle','-');
end
grid on
xlabel('Number of Prey');
ylabel('Number of Predators');
legend(num2str(ICs(1,1)),num2str(ICs(1,2)),num2str(ICs(1,3)),...
    num2str(ICs(1,4)),num2str(ICs(1,5)) )
set(gcf,'PaperPosition',[1 1 4.5 3.2])
if PrintFlag
    print -depsc2 lotkavolterra1.eps
end
figure(2); hold on
plot(tRK45(:),yRK45(:,1),'color','g','marker','none',...
    'linestyle','-','linewidth',2)
plot(tRK45(:),yRK45(:,2),'color','k','marker','none',...
    'linestyle','-','linewidth',2)
grid on
xlabel('Dimensionless Time');
ylabel('Number of Prey and Predators');
legend('Prey','Predators','Location','N')
set(gcf,'PaperPosition',[1 1 4.5 3.2])
if PrintFlag
    print -depsc2 lotkavolterra2.eps
end
figure(3); hold on
x = 1:1:(length(tRK45)-1);
y1 = diff(tRK45);
plot(x,y1)
xlabel('Time step #');
ylabel('Time step length');
```

RHSLV.m

```
function [f] = RHSLV(t,y, r1,r2,b1,b2)
% The right hand sides of the Lotka-Volterra system of ODE's
% Must be coded by you as a >>>COLUMN<<< vector
% TWP, E77N, April 4, 2001
%
f = zeros(2,1);
f(1,1) = y(1)*(r1-b1*y(2));
f(2,1) = y(2)*(-r2+b2*y(1));
```

Chapter 25

Sort & Search Algorithms, Big O Analysis

25.1 What are you going to learn?

In this lecture you shall learn about the rearrangement of items into ascending and descending order. You shall also learn about the computational cost of this rearrangement. We shall follow the thoroughly tested algorithms in Volume 3 of DONALD E. KNUTH's monograph entitled: *The Art of Computer Programming* [21]. This volume has 780 pages packed with information on general-purpose sorting and searching algorithms.

In Chapter 25 on fttp:\\petroleum.berkeley.edu, you will find the following functions

bubblesort.m Bubble sort algorithm based on KNUTH's monograph.

picksort.m Pick sort algorithm based on KNUTH's monograph.

qsort.m Quick sort algorithm based on *Numerical Recipes in C* [28].

binsearch.m Binary search algorithm based on KNUTH's monograph.

SortDemos.m Animation of sort of a vector of random numbers with bubble sort, pick sort and quick sort.

bubblesortwgraph.m Bubble sort animation function.

picksortwgraph.m Pick sort animation function.

qsortwgraph.m Quick sort animation function.

bigo.m Big O analysis. Definition of Big O description of truncation error by expanding $\sin(h)$ into Taylor series with two terms. Cost of initializing a two-dimensional array with and without prior memory allocation.

BubbleSortDemo.m Demonstration that bubble sort is $O(n^2)$.

PickSortDemo.m Demonstration that quick sort is $O(n^2)$.

QSortDemo.m Demonstration that quick sort is $O[n \log(n)]$.

MatlabSortDemo.m Demonstration that MATLAB sort() is $O[n \log(n)]$.

SortAndSearchDemo.m Demonstration that binary search is $O[\log(n)]$.

25.2 Why is it important?

Imagine how difficult life would be if dictionaries or phone books were not alphabetized. Similarly, we will see that the order in which items are stored in computer memory often has a profound influence on the speed and simplicity of algorithms that manipulate those items. Some of the most important applications of sorting are:

- *Solving the "togetherness problem",* in which all items with the same identification are brought together. Suppose that we have 10000 items in arbitrary order, many of which have equal values; suppose that we want to rearrange these items so that all items with equal values appear in consecutive positions.

- *Matching items from two or more files.* If several files have been sorted into the same order, it is possible to find all of the matching entries in one sequential pass through them, without backing up.

- *Searching for information by key values.* Sorting is also an aid to searching, hence it helps us make compute output more suitable for humans. As Knuth says, a listing that has been sorted into alphabetic order often looks quite authoritative even when associated with numerical information that has been incorrectly computed.

In real world business applications, sorting may be the most important part of computing workload. Just think about the Web search engines, such as Yahoo! or Alta Vista, and imagine what would happen if their answers to your complex questions popped on the screen in 30 minutes, not in 30 seconds.

25.3 Sorting

Sorting is the process of rearranging the elements of an array into numerical order (either ascending or descending). Given an initial array

$$R_1, R_2, \ldots, R_N,$$

we want to find a permutation $p(1), p(2), \ldots, p(N)$ of the indices $\{1, 2, \ldots, N\}$ that will put the array into nondecreasing order:

$$R_{p(1)} \leq R_{p(2)} \leq \cdots \leq R_{p(N)}.$$

R_1	R_2	R_3	← Record
89	37	41	Key
R	S	O	Satellite information
D	T	D	Satellite information
			Slot for a link

The definitive description of sorting algorithms is, again, in Donald E. Knuth, *The Art of Computer Programming, Vol. 3*, second edition (1998).

25.3.1 Bubble sort

Despite the quotes at the end of this lecture, we do describe bubble sort because we still think it is instructive. Perhaps the most obvious way to sort is to compare R_1 with R_2, interchanging them if they are out of order; then do the same to R_2 and R_3, R_3 and R_4, etc.. During this sequence, large numbers tend to move to the right, and in fact the largest record will move up to become R_N. Repeating this process will get the appropriate records into positions R_{N-1}, R_{N-2}, etc., so that all records will ultimately be sorted. The method is called "bubble sorting" because large elements "bubble up" to their proper position. After each pass, it is not hard to see that all records above and including the last one to be exchanged must be in their final positions, so they need not be examined on subsequent passes. Here is the MATLAB code:

```
function [R] = bubblesort(R)
% bubblesort.m
% Exchange selection sort of array R
%
% B1. Initialize BOUND
BOUND = length(R);
%
% B2. Loop in j
while (BOUND>1)
    t = 0;
    for j = 1:BOUND-1
        %
        % B3. Compare/exchange R(j) and R(j+1)
        if (R(j)>R(j+1))
            temp = R(j);
            R(j) = R(j+1);
            R(j+1) = temp;
            t = j;
        end
    end
    %
    % B4. Reset BOUND
    BOUND = t;
end
```

Example 19 Table 25.1 shows an example of bubble sort in action. The vertical bar | shows the position of `BOUND`. Records to the right of this bar are in their final position and need not be sorted.

Table 25.1: Example of bubble sort

Data		503		275		908		87		703	|
Pass 1		275		503		87		703	|	908	
Pass 2		275		87	|	503		703		908	
Pass 3		87	|	275		503		703		908	
Pass 4	|	87		275		503		703		908	

25.3.2 Straight insertion (Picksort)

The idea of straight insertion is very simple. (In fact, this is the way experienced card players sort their cards.) Assume that $1 < j \leq N$, and that records $R_1, \ldots, R_{j-1}$ have been rearranged so that

$$R_1 \leq R_2 \leq \cdots \leq R_{j-1}.$$

We compare the record R_j with R_{j-1}, R_{j-2}, $\ldots$, until we find out that R_j should be inserted between records R_i and R_{i+1}; then we move records R_{i+1}, $\ldots$, R_{j-1} up one space and put the new record into position $i+1$. Here is the MATLAB code:

```
function [R] = picksort(R)
% picksort.m
% Straight insertion sort of array R
%
% S1. Loop on j
for j = 2:length(R)
    %
    % S2. Set up i, temp
    i = j-1;
    temp = R(j);
    %
    % S3. Compare R(i):temp
    while R(i) > temp
        %
        % S4. Move R(i), decrease i
        R(i+1) = R(i);
        i = i-1;
        if (i == 0)  break;   end
    end
    %
    % S5. temp into R(i+1)
    R(i+1) = temp;
end
```

Example 20 Table 25.2 shows an example of straight insertion sort for the same original array as before. The symbol $\wedge$ shows the location at which the new record has to be inserted.

Table 25.2: Example of straight insertion

Data	503	275	908	87	703
Pass 1	$\wedge$ 503	: *275*			
Pass 2	275	503	$\wedge$: *908*		
Pass 3	$\wedge$ 275	503	908	: *87*	
Pass 4	87	275	503	$\wedge$ 908	: *703*
Pass 5	87	275	503	703	908

25.3.3 Other methods

Shellsort Improvement of straight insertion following a "divide-and-conquer" approach, which allows records to take long leaps instead of small steps.

Heapsort This is a very powerful method of sorting by selection, based on the construction of a binary tree structure.

Quicksort This is, on average, the fastest general-purpose sorting algorithm known to date. It is based on a partition-exchange procedure.

Quicksort (Not Required)

QuickSort is a sorting algorithm that sorts an n element input array in ascending order. The longest run time of QuickSort is $O(n^2)$, when the input array is already sorted. On average, the running time of QuickSort is $O(n \log n)$, which is remarkably efficient.

The following is a description of the QuickSort algorithm:

1. Choose one element in the array, `A`, that will be the pivot element, `p`, from `A(first...last)`.

2. Make one pass through the array, called the partition step, rearranging the elements such that:

   - The pivot is in its proper place.
   - Elements < than pivot are to the left of pivot, `A(first...p-1)`.
   - Elements ≥ to pivot are to the right, `A(p+1...last)`.

3. Recursively apply QuickSort to the part of the array that is to the left of the pivot, and to the part on the right.

Arranging the array elements around the pivot generates two smaller sorting problems - sort the left segments of the array and sort the right segment of the array. The relationships between the pivot and the array elements imply that once you solve the left and right sorting problems, the original sorting problem has been solved. Partitioning the array before the recursive calls places the pivot in its correct position and ensures that when the smaller array segments are sorted, their elements will be in the proper relation to the rest of the array. The QuickSort algorithm will eventually terminate. The left and right sorting problems are smaller than the original sorting problem, and each is closer to the base case, an array containing one element.

Of course, for speed, QuickSort should be implemented iteratively. An iterative implementation below, patterned after the *Numerical Recipes in C* is rather complicated:

```
function [R]= qsort(R)
% T.W. Patzek, E77N Class Material, 2002-04-28
% Quick sort algorithm adapted from Numerical
% Recipes in C, pages 333-334, Second Edition
% KNOWN BUG: Sometimes leaves first element out
% of order.  To be fixed next semester
M      = 7;
NSTACK = 50;
ir     = length(R);
il     = 1;
jstack = 0;
istack = zeros(1,NSTACK);
while(1)
   if ir-il<M % Pick sort when the subarray is small
      for j=(il+1):1:ir
         r = R(j);
         for i=(j-1):-1:1
            if (R(i)<=r) break; end;
            R(i+1) = R(i);
         end
         R(i+1) = r;
      end
      if (jstack == 0) break; end;
      % Pop stack and begin a new round of partitioning
      ir     = istack(jstack);
      jstack = jstack -1;
      il     = istack(jstack);
      jstack = jstack -1;
   else
      % Choose median of left and right elements as
      % the partitioning element r.
      k       = floor((il+ir)/2);
      % Also rearrange so that R(il+1)<=R(il)<=R(ir)
      temp    = R(il+1);
      R(il+1) = R(k);
      R(k)    = temp;
      if R(il+1)>R(ir)
         temp    = R(il+1);
         R(il+1) = R(ir);
         R(ir)   = temp;
      end
      if R(il)>R(ir)
         temp  = R(il);
         R(il) = R(ir);
         R(ir) = temp;
      end
      if R(il+1)>R(il)
         temp    = R(il+1);
         R(il+1) = R(il);
         R(il)   = temp;
      end
      % Initialize pointers to partitioning
      i=il+1;
      j=ir;
      r=R(il); % Partitioning element
      % Beginning of the innermost loop
      while(1)
         % Execute each while loop at least once
         % Scan up to find element >= r
         while (1)
            i=i+1;
            if (R(i)>=r) break; end;
         end
         % Scan down to find element <= r
         while(1)
            j=j-1;
            if (R(j)<=r) break; end
         end
         % Pointers crossed, scan is complete
         if (j<i) break; end;
         % Swap the offending elements
         temp = R(i);
         R(i) = R(j);
         R(j) = temp;
      end
      % Final swap of this round
      R(il)  = R(j);
      R(j)   = r;
      jstack = jstack+2;
      if (jstack>NSTACK)
         error('NSTACK too short for qsort');
      end
      % Push current partitions on the stack
      if (ir-i+1 >= j-il)
         istack(jstack)   = ir;
         istack(jstack-1) = i;
         ir               = j-1;
      else
         istack(jstack)   = j-1;
         istack(jstack-1) = il;
         il               = i;
      end
   end
end
```

25.4 Searching

In this section we shall concentrate on methods that are appropriate for searching a table whose elements satisfy

$$R_1 < R_2 < \cdots < R_N.$$

25.4.1 Binary search

The first method that suggests itself is to start by comparing the argument R to the middle element in the table; the result of this comparison tells which half of the table should be searched next, and the same procedure can be used again. Eventually we will have found the record or detected the two records embracing the argument.

Although the basic idea of binary search is straightforward, the details can be tricky. Here is a MATLAB implementation following Knuth:

```
function [lower,upper] = binsearch(R,K)
% Binary search of an ordered table
% April 24, 2002
%
% B1. Initialize
lower = 1;
upper = length(R);
% B2. Get midpoint
while (upper >= lower)
    mid = floor(0.5*(lower+upper));
    % B3. Compare
    if (K < R(mid))
```

```
        % B4. Adjust upper
        upper = mid-1;
    else
        % B5. Adjust lower
        lower = mid+1;
    end
end
% B6. Brackets crossed, swap them
temp  = lower;
lower = upper;
upper = temp;
```

Example 21 Table 25.3 shows two examples of binary search. The brackets [and] indicate the position of the lower and upper limits of the search, respectively. The underlined record represents the middle element. In both examples the search terminates after four steps.

Table 25.3: Examples of binary search

(a) Searching for $K = 653$

[061	087	154	170	275	426	503	$\underline{509}$	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	$\underline{677}$	703	765	897	908]
061	087	154	170	275	426	503	509	[512	$\underline{612}$	653]	677	703	765	897	908
061	087	154	170	275	426	503	509	512	612	[$\underline{653}$]	677	703	765	897	908

(b) Searching for $K = 400$

[061	087	154	170	275	426	503	$\underline{509}$	512	612	653	677	703	765	897	908]
[061	087	154	$\underline{170}$	275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275	$\underline{426}$	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[$\underline{275}$]	426	503	509	512	612	653	677	703	765	897	908
061	087	154	170	275]	[426	503	509	512	612	653	677	703	765	897	908

25.5 Analysis of algorithms

An algorithm must always terminate after a finite number of steps. In fact, this should be a reasonable number so that we get an answer in a reasonable time using a computer. The study of the quantitative behavior of algorithms (such as the number of times each step is executed) is called *analysis of algorithms*.

25.5.1 *O*-notation

We often want to quantity approximately, rather than exactly. The O-notation allows us to replace the "≈" sign by "=" and to quantify the degree of accuracy;

for example,

$$f(x+\tfrac{1}{n}) = f(x) + f'(x)\tfrac{1}{n} + O((\tfrac{1}{n})^2).$$

The last term is read as "big-oh of one over n squared." In general, let $\epsilon_n = O(f(n))$. The precise meaning is this: there are positive constants M and n_0 such that $|\epsilon_n| \leq M|f(n)|$, for all integers $n \geq n_0$. We do not say what the constants M and n_0 are because we usually do not know them.

Example 22 We know that

$$1^2 + 2^2 + \cdots + n^2 = \tfrac{1}{3}n(n+\tfrac{1}{2})(n+1) = \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n.$$

It follows that

$$\begin{aligned}
1^2 + 2^2 + \cdots + n^2 &= O(n^4),\\
1^2 + 2^2 + \cdots + n^2 &= O(n^3),\\
1^2 + 2^2 + \cdots + n^2 &= \tfrac{1}{3}n^3 + O(n^2).
\end{aligned}$$

The first equation is rather crude, but not incorrect. The last equation is the strongest statement. □

Example 23 Let's expand sin(h) in Taylor series about 0:

$$\begin{aligned}
\sin(h) &\approx h - \frac{1}{6}h^3 + \frac{1}{5!}h^5,\\
\sin(h) &\approx h - \frac{1}{6}h^3 + O(h^n)
\end{aligned}$$

Suppose that we do not know the exponent n. To see what n is we could plot the difference between the sin(x) and its polynomial approximation, divided by h^m, $m = 1, 2, \ldots$, and check when this ratio becomes constant. In other words, there should be a positive constant M, such that

$$\frac{|\sin(h) - (h - \frac{1}{6}h^3)|}{h^m} \leq M \tag{25.1}$$

for all h larger than the smallest h we used. The result of this analysis is shown in **Figure 25.1**. □

The O-notation is very useful in analyzing algorithms because it leaves out unimportant details. It can also be manipulated algebraically, with some care. The most important consideration is that of *one-way inequalities.* We write

$$\tfrac{1}{2}n^2 + n = O(n^2) \ \longleftarrow \text{correct}$$

and not

$$O(n^2) = \tfrac{1}{2}n^2 + n \ \longleftarrow \text{wrong!!}$$

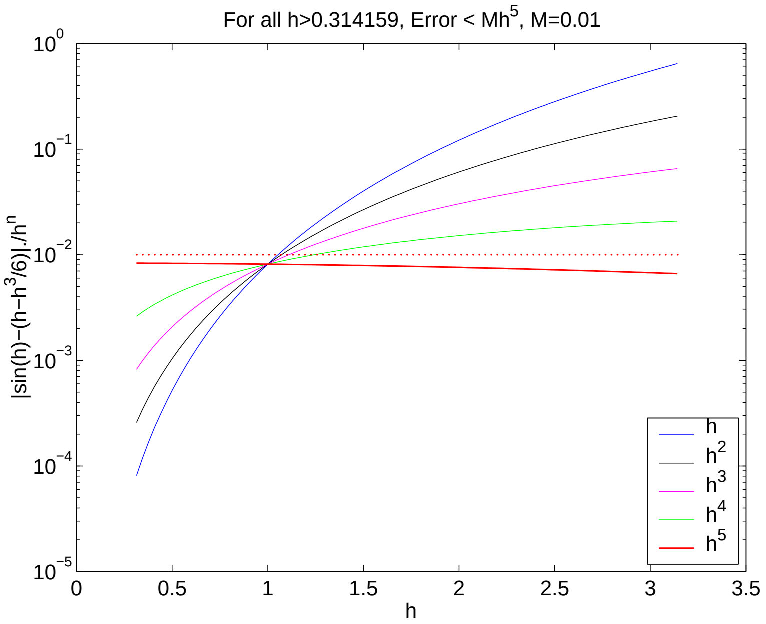


Figure 25.1: Big O analysis of truncation error. Scaling of Eq. (25.1) with h and $m = 1, 2, 3, 4, 5$. Error of this particular approximation of $\sin(h)$ is uniformly smaller that Mh for all $h \geq h$min, where $M = 0.01$ (the dotted line). Note that for small h, the linear approximation of $\sin(h$ is best.

The right-hand side of an equation does *not* give more information than the left-hand side. Here are some simple examples of O-notation operations:

$$\begin{aligned}
O(c \cdot f(n)) &= O(f(n)),\\
O(n + k) &= O(n),\\
O(n^2 + n) &= O(n^2),\\
O(\log n + k) &= O(\log n),\\
O(n \log n + n) &= O(n \log n).
\end{aligned}$$

25.5.2 Analysis of bubble sort

There are three different actions which can be identified. Let P be the number of passes (outer loops `while BOUND>1`), C the number of comparisons $R_j : R_{j+1}$, and E the number of exchanges $R_j \leftrightarrow R_{j+1}$. The running time will be proportional to the number of times that each statement in the program is executed. In our case (check it):

$$T = 1 + 4P + 2C + 4E$$

```
function [R] = bubblesort(R)
% bubblesort.m
% Exchange selection sort of array R
%
% B1.
BOUND = length(R);              % 1
%
% B2.
while (BOUND>1)                 % P1
    t = 0;                      % P2
    for j = 1:BOUND-1           % P3,C1
        %
        % B3.
        if (R(j)>R(j+1))        % C2
            temp = R(j);        % E1
            R(j) = R(j+1);      % E2
            R(j+1) = temp;      % E3
            t = j;              % E4
        end
    end
    %
    % B4.
    BOUND = t;                  % P4
end
% T= 1+4P+2C+4E
```

In the worst case scenario (array in descending order):

1. $P = n$ (`BOUND` decreases by one after each pass)
2. $C = (n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$ (arithmetic series)
3. $E = C$ (all comparisons are true)

We have

$$\max T = 1 + 4n + 2\frac{n(n-1)}{2} + 4\frac{n(n-1)}{2} = 1 + 3n^2,$$

and thus

$$\boxed{\max T = O(n^2).}$$

25.5.3 Analysis of MATLAB sort

The MATLAB sort algorithm is $O[n\log(n)]$, **Figure 25.4**.

25.5.4 Analysis of binary search

There are three different actions: P is the number of passes (sequence of searching intervals), C is the number of comparisons, and A is the number of adjustments of either limit of the interval. The number of operations will be

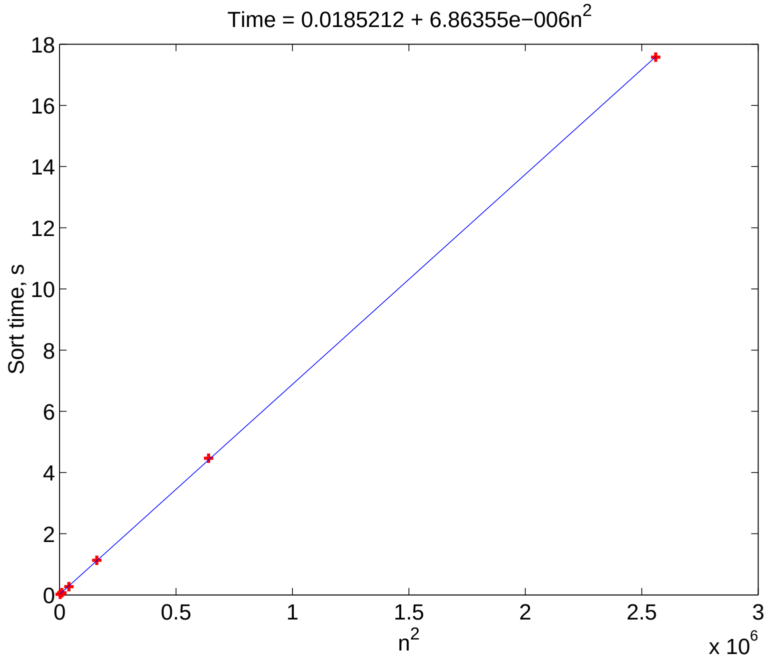


Figure 25.2: Scaling of bubble sort with n^2.

```
function [lower,upper] = binsearch(R,K)
% Binary search of an ordered table
% April 24, 2002
%
% B1. Initialize
lower = 1;                              % 1
upper = length(R);                      % 2
%
% B2. Get midpoint
while (upper >= lower)                  % C1
    mid = floor(0.5*(lower+upper));     % P1,P2,P3
    %
    % B3. Compare
    if (K < R(mid))                     % C2
        %
        % B4. Adjust upper
        upper = mid-1;                  % A1
    else                                % C2
        %
        % B5. Adjust lower
        lower = mid+1;                  % A2
    end
end
```

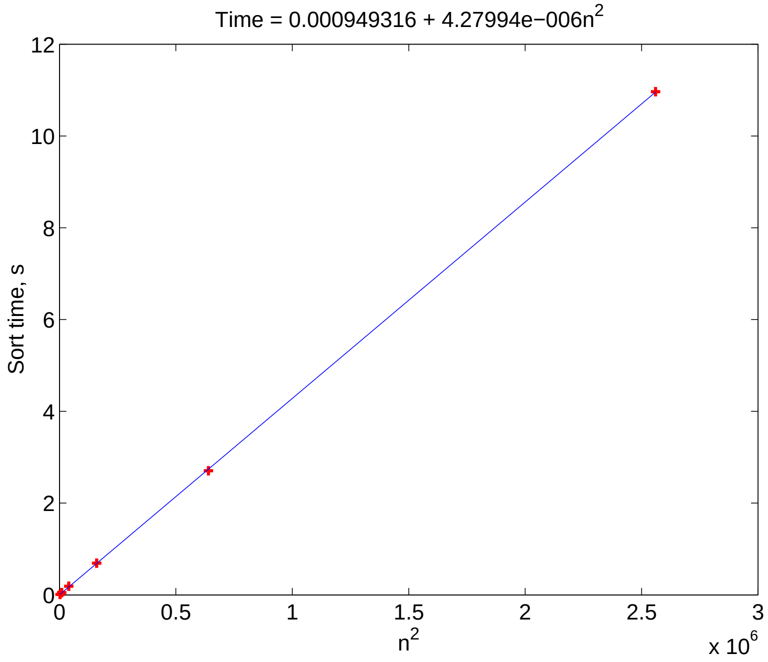


Figure 25.3: Scaling of pick sort with n^2.

```
temp  = lower;                        % 3
lower = upper;                        % 4
upper = temp;                         % 5
% T=5+3P+2C+A=5+3P+2P+P=5+6P
```

$$T = 5 + 3P + 2C + A.$$

Also, the number of comparisons is, in the worst case, $C = P$, and the number of adjustments is $A = P$. So we only have to find the number of passes. This is very easy because we decrease the size of the searching interval by a factor of two in every pass: $n, n/2, n/4, \ldots, 1$. If we take $\log_2$ of these lengths, we get $\log_2(n)$, $\log_2(n) - 1$, $\log_2(n) - 2$,..., 2, 1, 0. The length of this arithmetic sequence is $\log_2(n) + 1$, so $P = \lceil \log_2 n \rceil$. Thus, $T = 5 + 6\lceil \log_2 n \rceil$, and

$$\boxed{T = O(\log n).}$$

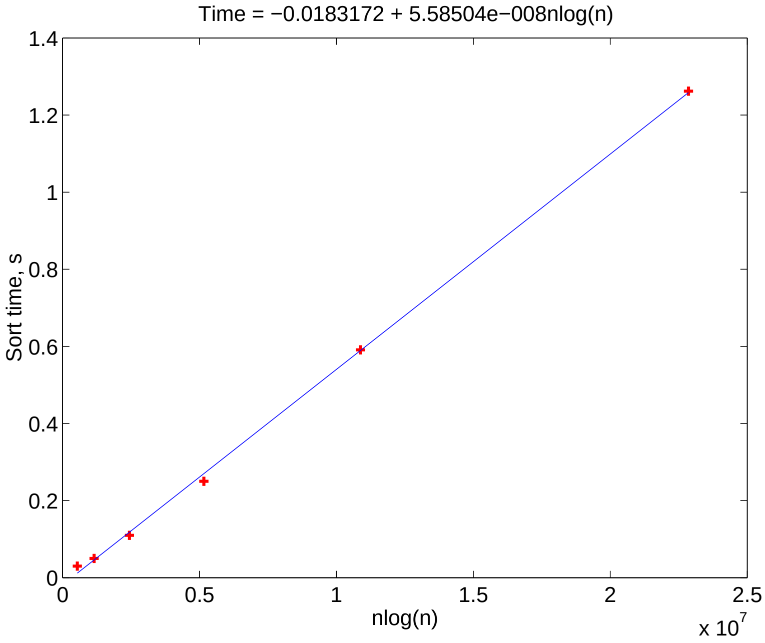


Figure 25.4: Scaling of MATLAB sort with $n \log n$.

In short, the bubble sort seems to have
nothing to recommend it, except a catchy name.

— DONALD E. KNUTH, *The Art of Computer Programming* (1973)

If you know what bubble sort is, wipe it out from your mind;
if you don't know, make a point of never finding out!

— WILLIAM H. PRESS *et al.*, *Numerical Recipes* (1986)

Chapter 26

Data Structures, Part II. Linked Lists, Queues and Stacks

26.1 What Are You Going To Learn?

You will learn about singly and doubly linked lists, stacks and queues. In Chapter 26 folder on fttp:\\petroleum.berkeley.edu, you will find

dsadbx.zip A zipped file, which contains 71 files creating the *Data Structures and Algorithms Toolbox*. The toolbox implements a new `POINTER` object. Unzip these files into a folder in which you will hold your data structure demos. You are now ready to use the other demos listed below.

SingleListDemo.m Demonstrate use of MathWorks' sl_* functions.

changetwp.m A function that replaces a string in a linked list.

QueueDemo.m Demonstrate use of MathWorks' qu_* functions.

StackDemo.m Demonstrate use of MathWorks' st_* functions.

OTHER In addition, the toolbox comes with its own demos.

26.2 Why Is It Important?

Most computer programs operate on tables of information [20]. Usually these tables are not amorphous masses of numbers, but they involve structural relationships among the data elements. In its simplest form, a table may be a linear list of elements. This structure is sufficient when the only questions we ask are: Which element is first? Which is last? Which elements precede and succeed an element of interest? Lists, and their simpler children, queues and stacks, are the fundamental linear data structures of Computer Science. They are essential to the efficient management of computer memory and implementations of computer algorithms.

26.3 MATLAB Shortcuts

Data Structures & Algorithms Toolbox for MATLAB R12, author YARON KEREN, zip file: `dsatbx.zip`.

26.3.1 Summary

The Data Structures & Algorithms Toolbox provides advanced data structures and algorithms for the MATLAB environment. The toolbox implements functions to create and destroy *singly* and *doubly linked lists*, *stacks*, *queues*, *binary trees* and *red-black (balanced) binary trees*:

1. Data structures can be dynamically allocated and destroyed.
2. Data elements can be inserted, deleted or searched for.
3. Contents of lists, queues and stacks can be displayed.
4. A user-defined function can operate on the fields of the lists and binary trees as they are visited.
5. Binary trees can be visited inorder, postorder and preorder.
6. A graphical representation of trees can be shown.

All algorithms are clearly written as m-files, which allows the toolbox to be used as a teaching aid in educational data structures courses. This is the first implementation available of advanced data types, common in other languages, for MATLAB.

26.3.2 Toolbox contents

```
% Data Structures & Algorithms Toolbox
% Version 1.03 8-Aug-2001
%
% Demonstrations:
% avl_demo     - AVL binary tree demo.
% bt_demo      - Binary tree demo.
% rb_demo      - Red-Black binary tree demo.
%                  (compare to bt_demo!)
% sl_demo      - Singly linked list demo.
%
% Binary trees:
% bt_del       - Delete a node.
% bt_find     - Find a node.
% bt_free     - Free allocated tree.
% bt_graph    - Graph the tree.
% bt_inord    - Traverse the tree inorder.
% bt_new      - Allocate a new tree.
% bt_posto    - Traverse the tree postorder.
% bt_preor    - Traverse the tree preorder.
% bt_put      - Add a node.
%
% Red-Black (balanced) binary trees:
% rb_del      - Delete a node.
% rb_find     - Find a node.
% rb_free     - Free allocated tree.
% rb_graph    - Graph the tree.
% rb_inord    - Traverse the tree inorder.
% rb_new      - Allocate a new tree.
% rb_posto    - Traverse the tree postorder.
% rb_preor    - Traverse the tree preorder.
% rb_put      - Add a node.
%
% AVL (better balanced) binary trees:
% avl_del     - Delete a node.
% avl_find    - Find a node.
% avl_free    - Free allocated tree.
% avl_graph   - Graph the tree.
% avl_inord   - Traverse the tree inorder.
% avl_new     - Allocate a new tree.
% avl_posto   - Traverse the tree postorder.
% avl_preor   - Traverse the tree preorder.
% avl_put     - Add a node.
%
% Stacks:
% st_empty    - Is empty?
% st_free     - Free allocated stack.
% st_new      - Allocate a new stack.
% st_pop      - Pop top element.
% st_push     - Push top element.
% st_top      - Return top element without popping it.
%
% Queues:
% qu_dequ     - Dequeue (remove) first element.
% qu_empty    - Is empty?
% qu_enqu     - Enqueue (insert) last element.
% qu_free     - Free allocated queue.
% qu_front    - Return first element without removing it.
% qu_new      - Allocate a new queue.
%
% Singly linked lists:
% sl_appnd    - Append node as tail of list.
% sl_count    - Count number of elements in list.
% sl_del      - Delete a node.
% sl_disp     - Display the list.
% sl_empty    - Is empty?
% sl_free     - Free allocated list.
% sl_get      - Return list head.
% sl_insrt    - Insert node as head of list.
% sl_new      - Allocate a new list.
% sl_puta     - Put after specified node.
% sl_trav     - Traverse the list.
%
% Doubly linked lists:
% dl_del      - Delete a node.
% dl_disp     - Display the list.
% dl_empty    - Is empty?
% dl_free     - Free allocated list.
% dl_get      - Return list head.
% dl_new      - Allocate a new list.
% dl_puta     - Put after specified node.
% dl_trav     - Traverse the list.
%
% Copyright (c) MathWorks Inc. 1998-2001. All rights reserved.
%
% Send suggestions, bug fixes and questions to yaron@mathworks.com
%
% AVL Tree and corrections to Red-Black Tree and Binary Tree
% made by Anthony Gallagher (http://www.cs.cmu.edu/~anthonyg)
% of the Robotics Institute, School of Computer Science,
% Carnegie Mellon University. Nov 2000.
```

26.4 Useful Definitions

The definitions below follow Knuth [20], and the Dictionary of Algorithms and Data Structures maintained on the Web http://www.nist.gov/dads/ by the National Institute of Standards (NIST).

First we must define a couple of terms essential in the definitions below. The information in a linear table consists of a set of *nodes*, also called "records," "entities," or "beads" by some authors. The synonyms of "node" are "item" or "element." More generally a node can be a *container*, i.e., an object used to store other objects, endowed with methods to access these objects. The singly linked list implementation discussed below allows for a user-defined function (a "method") to operate on the contents of each node. From this point of view, the list nodes can also be called containers. Each node consists of one or more consecutive words of computer memory, divided into named parts called *fields.*. The *address* of a node, also called a *link*, *pointer* or *reference* to the node, is the memory location of the node's first word. The contents of any field within a node may represent numbers, characters, links, or anything else we desire. In particular, in MATLAB each field may be a general cell array.

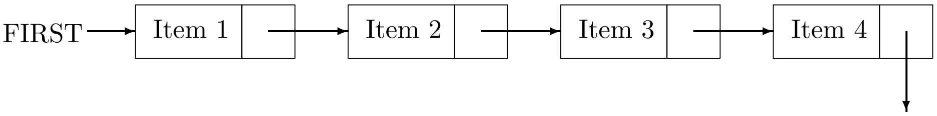


Figure 26.1: A singly linked list. Here FIRST is a link variable pointing to the first node of the list (its head). Node 4 is last (the list tail), and it points to nowhere.

Definition 24 A *linear list* is a sequence of $n \geq 0$ nodes N(1), N(2),. . . , N(n) that are described *only* by the relative positions as they appear in a line. All we care is that if $n > 0$, N(1) is the first node and N(n) is the last; also if $0 < k < n$, then the kth node N(k) is preceded by N($k-1$) and followed by N($k+1$). □

Definition 25 The *singly linked list*, **Figure 26.1**, is a convenient way to store an unbounded array, that is create an array when we don't know in advance how large this array will be. The disadvantage of the singly linked list is that data can only be accessed sequentially. To read the contents of the millionth element (*node*) of a singly linked list, you must access the 999,999 elements that precede it; the singly linked list supports only *forward traversal.* A linked list consists of a *root node*, allocated on the stack (a local variable) and one or more *nodes*, allocated on the heap (by allocating to them permanent storage locations[1]. In the singly linked list, each node has two parts, the link field to the subsequent node and the data field(s) containing the information you want stored in the linked list. The end of the singly linked list, its *tail*, is indicated by setting the link field to zero. The data field in each node can have different length and type; it can be a string, number array or cell array. □

Definition 26 A *doubly linked list* supports both forward and backward traversal, and insertion and removal of elements at the beginning or the end, or in the middle. In a doubly linked list, each node has a link to the previous node as well as the next. These links allow easier access of the list nodes backwards as well as forwards, and inserting or deleting a node in fixed time. □

Because different people mean different things when they say "list", you should always specify that you mean a linear list, singly linked list or a doubly linked list.

A singly linked list can be used to implement two other children data structures, a *stack* and a *queue* :

Definition 27 A *stack*, **Figure 26.2**, is linear list in which all insertions and deletions are made at one end of the list: only the most recently added item may be accessed or deleted. The latest added item is at the stack *top.* The basic operations defined for a stack are add (to the head) or *push*, remove (from the

[1] Equivalent to calling `malloc()` in C; see `malloc.m` in the Lecture 22 folder.

head) or *pop*, and access (the head) or *top*. Stacks are examples of the "last-in, first-out" or LIFO linear lists. Let `S` denote a stack and `v` a value. In the MATLAB toolbox, the operation `new()` is `st_new.m`, `push(v, S)` is `st_push.m`, `pop(S)` is `st_pop.m`, and `top(S)` is `st_top.m`. □

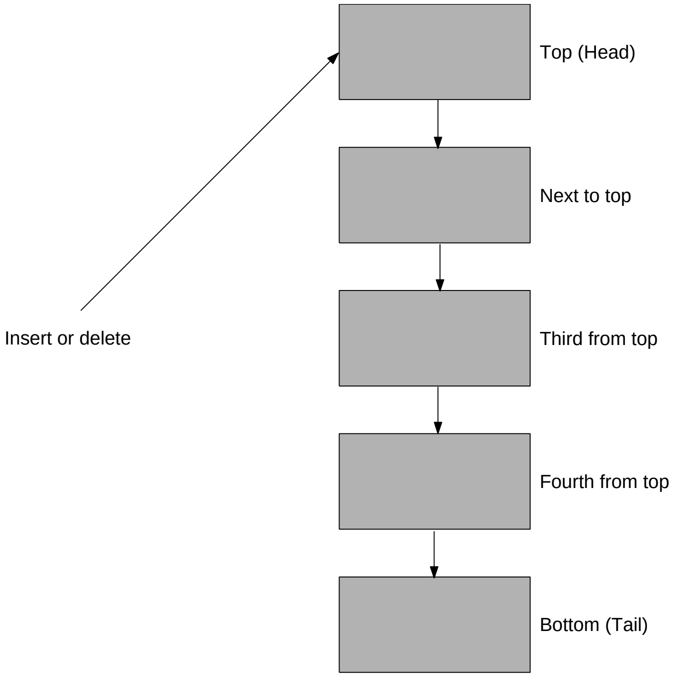


Figure 26.2: Stack

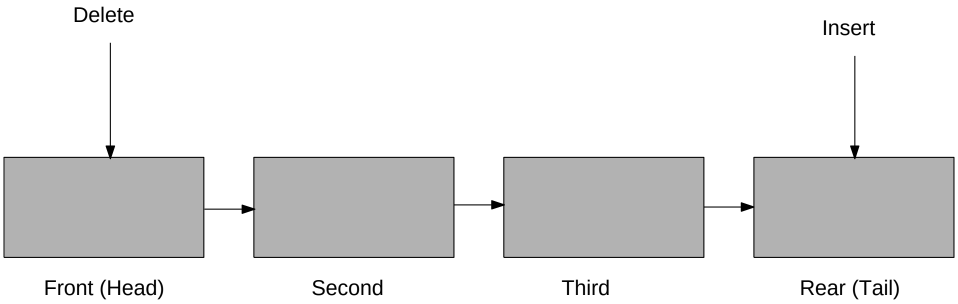


Figure 26.3: Queue

Definition 28 A *queue*, **Figure 26.3**, is a linear list for which all insertions are made at one end of the list and all deletions are made at the other end: only the earliest added item may be accessed or deleted. Basic operations are add (to the tail), or *enqueue*, and delete (from the head) or *dequeue*. The first element is at the queue's *front*. Delete returns the item removed. Queues are examples of the "first-in, first-out" or FIFO linear lists. Let `Q` denote a queue and `v` a value. In the MATLAB toolbox the operation `new()` is `qu_new.m`, `add(v, Q)` is `qu_enque.m`, `front(Q)` is `qu_front.m` , and `remove(Q)` is `qu_de-qu.m`. □

26.5 Singly Linked List, Example

The following demo will create a singly linked list, put cell arrays into each node by expanding the list head-first or tail-first. Then a user-defined function will operate on each node of the list replacing a fragment of text with another one:

```
function SingleListDemo
% SingleListDemo uses the MathWorks sl_* functions,
% copyrighted (c) MathWorks Inc. 1998-2001.
% E77N Class Material, T.W. Patzek, 2002-04-25
%
clc, clear all, close all;
% Create a new singly linked list
h1=sl_new;
% Put something into it:
Records = {'I love learning';
             'about linked lists';
             'because I love E77N too';
             'What will I do with my time';
             'when the semester is over';
             '??!!?'};
%----------------------------------------------------------
% NOTE: Each record can be anything: a cell array, string
% array, number... Records may be of different types
%----------------------------------------------------------
len = length(Records);
% Insert records into the list.
% The last record becomes the list head
for i = 1:len
    h1=sl_puta(h1,h1.head,Records{i,:});
end
disp('--- Contents of a singly linked list ---');
disp('--- Each new record is inserted at head ---');
% Display list contents from head to tail
sl_disp(h1);
disp('---------------------------');
% Get list head
data=sl_get(h1);
fprintf('\nList head is:  %s\n\n',data);
% Delete list
h1=sl_free(h1);
%----------------------------------------------------------
```

```
% Insert records into the list.
% The last record becomes the list tail
h2=sl_new;
for i = 1:len
    h2=sl_puta(h2,h2.tail,Records{i,:});
end
disp('--- Contents of a singly linked list ---');
disp('--- Each new record is inserted at tail ---');
sl_disp(h2);
disp('---------------------------');
% Get list head
data=sl_get(h2);
fprintf('\nList head is:  %s\n\n',data);
% While traversing each node of the list,
% execute a user-supplied function
disp('--- Execute a user function at each node ---');
sl_trav(h2,@changetwp);
% Free the list
h2=sl_free(h2);


function changetwp(s)
% TWP, E77N Class Material, 2002-04-25
[k]=findstr(s,'love');
if ~isempty(k)
    s(k(1):k(1)+3)='hate';
end
fprintf('%s\n',s);
```

26.5.1 What Does the Demo Do?

The SingleListDemo.m initializes two singly linked lists and inserts into them a cell array, one row per node. In the first case, the new node is inserted at the list head (just as in a stack), pushing the prior nodes further away. In the second case, the new item is inserted at the list tail (just like in a queue), preserving the order of rows in the input cell array. Here is the MATLAB output:

```
--- Contents of a singly linked list ---
--- Each new record is inserted at head ---
  ??!!?
  when the semester is over
  What will I do with my time
  because I love E77N too
  about linked lists
  I love learning
---------------------------

List head is:  ??!!?
```

```
--- Contents of a singly linked list ---
--- Each new record is inserted at tail ---
  I love learning
  about linked lists
  because I love E77N too
  What will I do with my time
  when the semester is over
  ??!!?
---------------------------

List head is:  I love learning

--- Execute a user function at each node ---
  I hate learning about linked lists
  because I hate E77N too
  What will I do with
  my time when the semester is over
  ??!!?
```

26.6 Stack Example

```
function StackDemo
% StackDemo uses the MathWorks st_* functions,
% Copyrighted (c) MathWorks Inc. 1998-2001.
% E77N Class Material, T.W. Patzek, 2002-04-26
%
clc, clear all, close all;
% Create a new stack
s1=st_new;
Records = {’Hi, I am a stack’;
           ’When you push in’;
           ’I pop out’};
%---------------------------------------------------------
% NOTE: Each stack record can be anything: a cell array,
% string array, number... Records may be of different types
%---------------------------------------------------------
len = length(Records);
% Insert records into the stack.
% The last record becomes the list head
disp(’--- Contents of a stack ---’);
disp(’--- Push a new record into stack’’s HEAD ---’);
for i = 1:len
    s1=st_push(s1,Records{i,:});
    % Display stack contents
    disp(’---------------------------’);
    sl_disp(s1);
end
```

```
% Get the top element of the stack
data=st_top(s1);
fprintf('\nStack top is:  %s\n\n',data);
disp('--- Contents of a stack ---');
disp('--- Pop a new record from stack''s HEAD ---');
sl_disp(s1);
for i = 1:len
    s1=st_pop(s1);
    % Display stack contents
    disp('---------------------------');
    sl_disp(s1);
end
disp('Stack is LIFO');
% Delete stack
s1=sl_free(s1);
```

26.6.1 What Does the Demo Do?

First we push three cell arrays into the stack:

```
>>StackDemo
--- Contents of a stack ---
--- Push a new record into stack's HEAD ---
---------------------------
  Hi, I am a stack
---------------------------
  When you push in
  Hi, I am a stack
---------------------------
  I pop out
  When you push in
  Hi, I am a stack
---------------------------

Stack top is:  I pop out

--- Contents of a stack ---
--- Pop a new record from stack's HEAD ---
  I pop out
  When you push in
  Hi, I am a stack
---------------------------
  When you push in
  Hi, I am a stack
---------------------------
  Hi, I am a stack
---------------------------
```

```
Stack is LIFO
```

26.7 Queue Example

```
function QueueDemo
% QueueDemo uses the MathWorks qu_* functions,
% Copyrighted (c) MathWorks Inc. 1998-2001.
% E77N Class Material, T.W. Patzek, 2002-04-26
%
clc, clear all, close all;
% Create a new stack
q1=qu_new;
Records = {'Hi, I am a queue';
           'When you push in';
           'I pop out'};
%----------------------------------------------------------
% NOTE: Each queue record can be anything: a cell array,
% string array, number... Records may be of different types
%----------------------------------------------------------
len = length(Records);
% Insert records into the queue.
% The last record becomes the queue tail;
disp('--- Contents of a queue ---');
disp('--- Enqueue a new record into queue''s TAIL ---');
for i = 1:len
    q1=qu_enqu(q1,Records{i,:});
    % Display queue contents
    disp('---------------------------');
    sl_disp(q1);
end
% Get the first element of the queue, without popping it
data=qu_front(q1);
fprintf('\nQueue front is:  %s\n\n',data);
disp('--- Contents of a queue ---');
disp('--- Dequeue a new record from queue''s HEAD ---');
sl_disp(q1);
for i = 1:len
    q1=qu_dequ(q1);
    % Display queue contents
    disp('---------------------------');
    sl_disp(q1);
end
disp('Queue is FIFO');
% Delete queue
q1=qu_free(q1);
```

26.7.1 What Does the Demo Do?

First we push three cell arrays into the queue:

```
>>QueueDemo
--- Contents of a queue ---
--- Enqueue a new record into queue’s TAIL ---
---------------------------
  Hi, I am a queue
---------------------------
  Hi, I am a queue
  When you push in
---------------------------
  Hi, I am a queue
  When you push in
  I pop out
---------------------------

Queue front is:  Hi, I am a queue

--- Contents of a queue ---
--- Dequeue a new record from queue’s HEAD ---
  Hi, I am a queue
  When you push in
  I pop out
---------------------------
  When you push in
  I pop out
---------------------------
  I pop out
---------------------------
Queue is FIFO
```

Chapter 27

Graphs, Adjacency Matrices and Lists, & Depth-First Search

27.1 What Are You Going To Learn?

You will learn a little about *graphs* and how their *vertices* and *edges* can be encoded and displayed in MATLAB. In particular, the graph connectivity will be encoded through the *adjacency matrix* and the *adjacency list*. You will also see an implementation of the *depth-first search* using the adjacency list description of graphs.

In Chapter 27 folder on fttp:\\petroleum.berkeley.edu, you will find the files pertinent to the adjacency matrix description of graphs:

am_graph.m is the main function to generate a directed graph, given a connectivity (adjacency) matrix.

am_graphvertices.m generates n vertices of a graph located on a symmetric n-gon rotated by an arbitrary angle.

am_graphedges.m generates edges linking vertices V of a graph and adjusts their lengths to fit the radius R of each vertex (node) to be plotted.

am_edgearrows.m generates arrows in the middle of graph each edge.

arrowgen.m , plots a a single vector with an arrow.

myrotation.m rotates Cartesian coordinates (x, y) of each row of the vertex array V counterclockwise.

am_plotvertices.m plots a vertices of a polygon and labels them as they appear in the vertex matrix V.

am_plotedges.m .

am_plotarrows.m plots an arrow in the middle of each directed edge of a polygon.

plotcircle.m plots edges of a polygon.

You will also find files pertinent to the adjacency (neighbor) list description of graphs:

nl_graph.m generates an undirected graph using the neighbor list description.

nl_graphedges.m generates edges linking vertices of a graph G and adjusts their lengths to fit the radius R of each vertex (node) to be plotted.

nl_plotedges.m plots edges of a polygon with neighbor list description.

nl_dfs.m A naïve depth-first search of a graph.

nl_plotvertices.m plots a vertices of a polygon and labels them as they appear in structure G.

27.2 Why Is It Important?

Engineers and scientists use mathematics to build models of practical problems. One area of mathematics particulary well-suited to model building is *graph theory.* Graphs are essential to solve problems in

1. Connections: electrical, flow, chemical...
2. Scheduling: classes, computer laboratories, airport takeoff times...
3. Transportation: highway control, street light control, UPS deliveries, FedEX flights...
4. Network analysis: pore space of a natural rock is a disordered network with random connectivity
5. Games and Puzzles.

Graph theory is routinely applied in

1. Critical path analysis
2. Molecular chemistry
3. Analysis of complex chemical reactions
4. Genetics
5. Linear algebra (matrix theory)

6. Set theory
7. Topology
8. Group theory
9. Sort & Search
10. Analysis of computer algorithms
11. ... (just to name a few)

27.3 Background

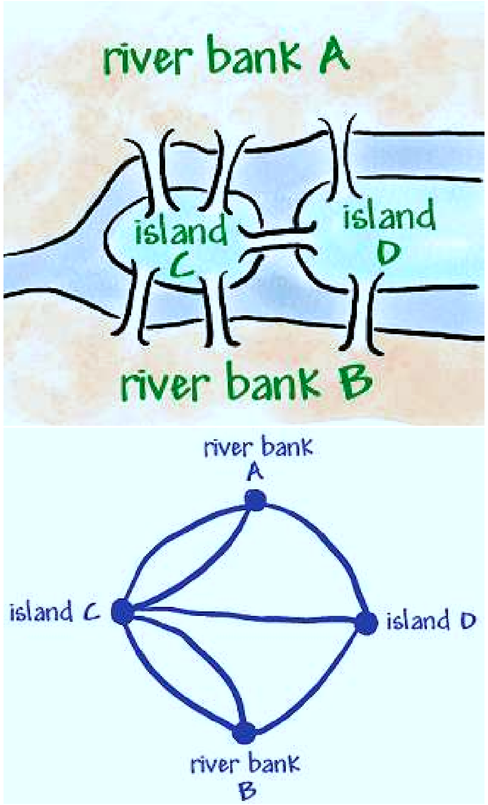


Figure 27.1: It became a tradition to try to walk around Königsberg crossing each bridge only once, but nobody could find the right path. LEONHARD EULER, proved that such a walk was impossible. He proved this by inventing a diagram called *graph* (or network), that was made up of *vertices* (dots where lines meet, also called sites or nodes) and arcs (also called *edges* or links).

Figure 27.2: LEONHARD EULER, probably the greatest mathematician of all times. In 1736 EULER published a paper on the solution of the Königsberg bridge problem entitled *Solutio problematis ad geometriam situs pertinentis* which translates into English as *The solution of a problem relating to the geometry of position*. The title indicates that EULER knew he was dealing with a different type of geometry where distance was *not* relevant, but connectivity was.

The river Pregel divides the town of Königsberg into four separate land masses, A, B, C, and D, **Figure 27.1**. Seven bridges connect the various parts of town, and some of the town's curious citizens wondered if it were possible to take a journey across all seven bridges without having to cross any bridge more than once. All who tried ended up in failure, including the Swiss mathematician, LEONHARD EULER, **Figure 27.2**, probably the best mathematician of all times. Of course, EULER succeeded in explaining why such a journey was impossible, not only for the Königsberg bridges, but whether such a journey was possible or not for any network of bridges anywhere. Euler reasoned that to succeed each land mass should have an even number of bridges connected to it, or if the journey would begin on one land mass and end on another, then exactly those two land masses could have an odd number of connecting bridges, while all other land masses must have an even number of connecting bridges.

To succeed, a traveller in the middle of the journey must enter a land mass via one bridge and leave by another, thus that land mass must have an even number of connecting bridges. Further, if the traveller at the start of the journey leaves one land mass, then a single bridge will suffice and upon completing the journey the traveller may again only require a single bridge to reach the ending point of the journey. The starting and ending points then, are allowed to have an odd number of bridges. But if the starting and ending point are to be the same land mass, then it and all other land masses must have an even number of connecting bridges.

Alas, all the land masses of Königsberg have an odd number of connecting bridges and the journey that would take a traveller across all the bridges, one and only one time during the journey, proves to be impossible!

27.4 Graph Definitions

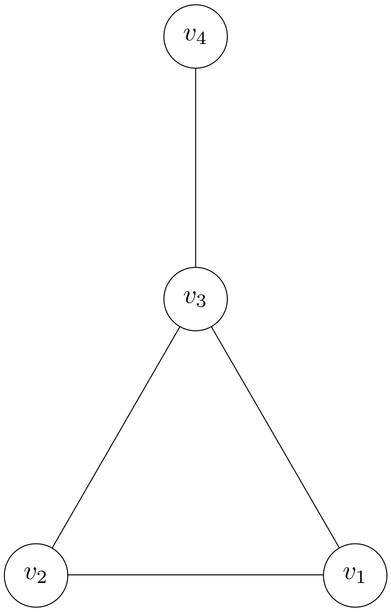


Figure 27.3: An *undirected* graph with 4 vertices and 4 edges.

Definition 29 A *graph* G is a finite nonempty set of *vertices*, V, together with a set of *edges*, E. For example, **Figures 27.3** and **27.4**, a graph, G, may be defined by the vertex set:

$$V = \{v_1, v_2, v_3, v_4\}, \tag{27.1}$$

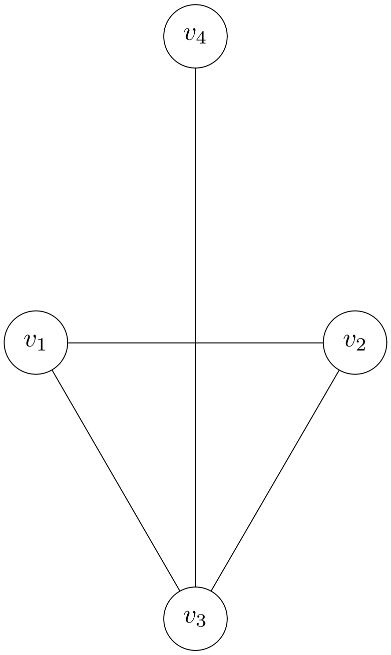


Figure 27.4: This graph is equivalent to the graph in Figure 27.3.

and the edge set:

$$\begin{aligned} E = \{&\{(v_1, v_2), (v_2, v_1)\}, \{(v_1, v_3), (v_3, v_1)\}, \\ &\{(v_2, v_3), (v_3, v_2)\}, \{(v_3, v_4), (v_4, v_3)\}\}. \end{aligned} \tag{27.2}$$

□

Definition 30 A *directed graph* or *digraph*, D, is a finite nonempty set of *vertices*, V, together with a set of *directed edges* or *arcs*, E. For example, **Figure 27.5**, a graph, D, may be defined by the vertex set:

$$V = \{v_1, v_2, v_3, v_4\}, \tag{27.3}$$

and the edge set:

$$E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_4, v_3)\}. \tag{27.4}$$

You can think of undirected graphs as a subset of directed graphs where each edge (v_i, v_j) coexists with the edge (v_j, v_i). For simplicity, we will identify each vertex with its index and write, e.g., 3 instead of v_3. □

Definition 31 The number of vertices in G is called the *order* of G. The number of edges in G is called the *size* of G. Occasionally it is useful to to talk about the *vertex set* or *edge set* of a given graph: $V(G)$ and $E(G)$. Graphs are more general data structures than trees in that they may have *loops*. □

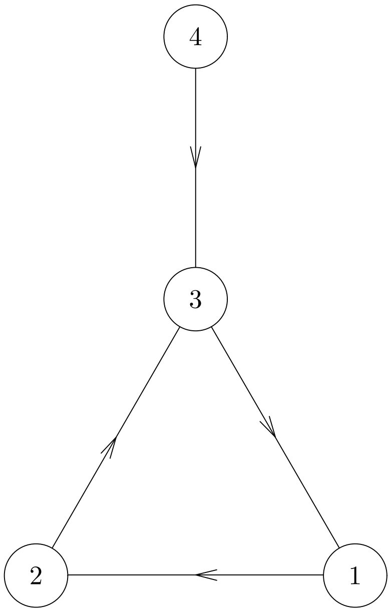


Figure 27.5: The *directed* graph above is *not* the same as that in Figure 27.3.

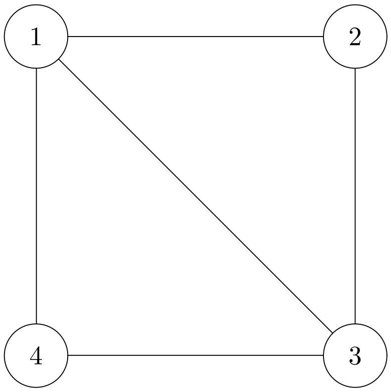


Figure 27.6: A simple undirected graph: $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 1), (2, 3), (3, 2), (3, 4), (4, 3), (4, 1), (1, 4), (1, 3), (3, 1)\}$.

27.5 Adjacency Matrix Representation

Definition 32 A graph with n vertices (nodes) can be represented as a matrix of n rows and n columns. The (i, j)th-entry of the matrix is the number of edges from node i to node j $(0, 1, 2, \dots)$. Note that the the adjacency matrix of an *undirected* graph is symmetric with respect to the diagonal or anti-diagonal,

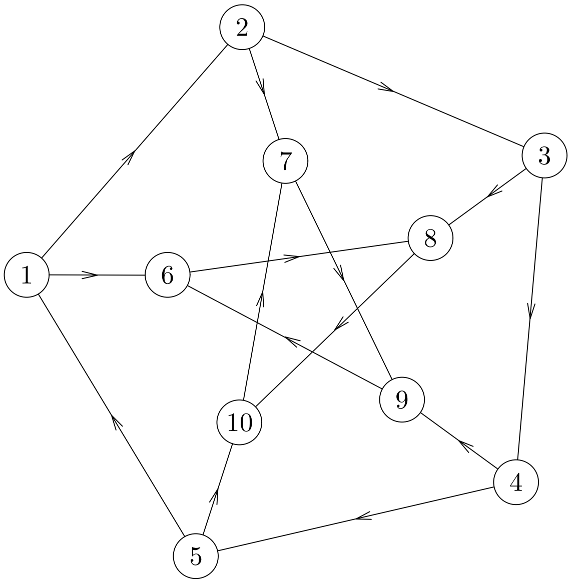


Figure 27.7: A more complicated directed graph: $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $E = \{$(1,2), (2,3), (3,4), (4,5), (5,1), (1,6), (2,7), (3,8), (4,9), (5,10), (6,8), (8,10), (10,7), (7,9), (9,6)$\}$.

depending on the node numbering scheme. This symmetry is enforced by Eq. (27.2): if node i is connected to node j, then both (i, j) and (j, i)th elements of the matrix are equal. The adjacency matrix of a *directed* graph is *not* symmetric in general. □

27.5.1 Examples

Example 24 Adjacency matrix of the graph in **Figure 27.6** is symmetric with respect to the diagonal, because the nodes are numbered clockwise:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \tag{27.5}$$

If node labels 3 and 4 were swapped, the node numbering would be rowwise, the new adjacency matrix would be different:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{27.6}$$

but still symmetric with respect to the diagonal. □

Example 25 Adjacency matrix of the directed graph in **Figure 27.7** is not symmetric:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad (27.7)$$

□

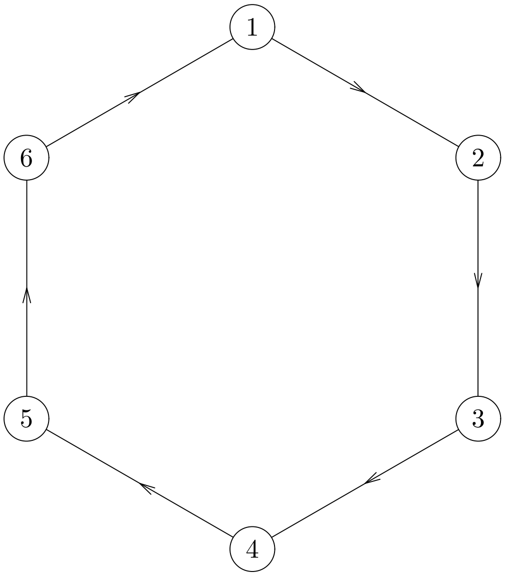


Figure 27.8: The directed graph in **Example 26**.

Example 26 The adjacency matrix representation of graphs allows one to operate on graphs using well-known matrix techniques which are efficiently implemented in MATLAB. For example the operation of squaring a graph can be done by squaring the adjacency matrix. Let us square a simple graph in **Figure 27.8**, $\boldsymbol{A} = \boldsymbol{A}^2$. The result is shown in **Figure 27.9**. Now every second node is connected. This is one of the reasons why "adjacency matrix" has its name. □

27.6 Adjacency List Representation

The *undirected* graph in Figure 27.6 was conveniently represented by the adjacency matrix (27.5). The same graph can be represented alternatively by the

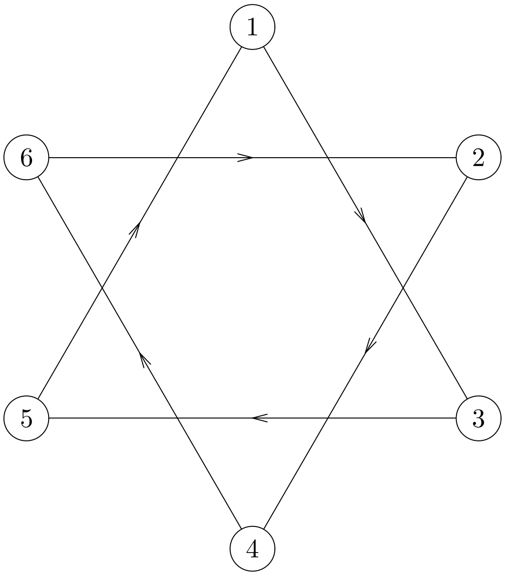


Figure 27.9: The *square* of the graph in Figure 27.8.

adjacency list (the list of neighbors) of each vertex:

$$\begin{array}{rl} 1 : & [2,3,4] \\ 2 : & [1,3] \\ 3 : & [1,2,4] \\ 4 : & [1,3] \end{array} \tag{27.8}$$

For the *directed* graph in Figure 27.7, we only account for the edges going *out* of a given vertex:

$$\begin{array}{rl} 1 : & [2,6] \\ 2 : & [3,7] \\ 3 : & [4,8] \\ 4 : & [5,9] \\ 5 : & [1,10] \\ 6 : & [7] \\ 7 : & [9] \\ 8 : & [10] \\ 9 : & [6] \\ 10 : & [7] \end{array} \tag{27.9}$$

27.7 Depth-First-Search

In Chapter 9, we saw three kinds of binary tree traversals, preorder, inorder and postorder; now we look at graph traversals. There are two types of graph traversals: depth-first search and breadth-first search, often abbreviated DFS and BFS respectively. With the adjacency list implementation of the graph, both types of traversals give $O(|V| + |E|)$ time for a complete traversal. This time is optimal, since in order to completely traverse a graph using any method, you must visit all of the vertices and edges. The key to a fast traversal is avoiding loops: you do not want to get stuck going around and around a cycle endlessly.

Definition 33 Breadth-first search goes through the tree level-by-level, visiting all of the nodes on the top level first, then all the nodes on the second level, and so on. This strategy has the benefit of being complete (if there's a solution, it will be found), and optimal as long as the shallowest solution is the best solution. However, the way breadth-first search achieves this is by keeping all of the leaf nodes in memory, which requires a prohibitive amount of memory when searching anything more than a very small tree. □

Definition 34 Depth-first search goes through the tree branch by branch, going all the way down to the leaf nodes at the bottom of the tree before trying the next branch over. This strategy requires much less memory than breadth-first search, since it only needs to store a single path from the root of the tree down to the leaf node. However, it is potentially incomplete, since it will keep going on down one branch until it finds a dead-end, and it is non-optimal – if there is a solution at the fourth level in the first branch tried, and a solution at the second level in the next one over, the solution at the fourth level will be returned. □

27.8 Exercises

1. What is the maximum number of connections in a directed graph with n vertices?

2. If one integer takes 8 bytes of storage in MATLAB, what would be the space required to store a graph with 1000 vertices?

3. What will be the result of transposing the adjacency matrix in Example 26?

Chapter 28

Object Oriented Programming in MATLAB

28.1 What Are You Going To Learn?

Before you start reading this lecture, please review the definitions of *class* and *object* introduced in Chapter 8, and the properties and operations on polynomials discussed in Chapter 17.

In this lecture you will learn the fundamentals of object oriented programming methods peculiar to MATLAB. We shall proceed as follows:

- Classes and objects
- Designing classes in MATLAB
- Overloading operators and functions
- Example: a polynomial class: polynom

The Chapter 28 folder on fttp:\\petroleum.berkeley.edu has the following files[1]:

polynomdemo.m Demo which demonstrates the polynom object methods.

@polynom\polynom.m polynom object constructor.

@polynom\char.m An overloaded method converting a polynom object into a syntactically correct string.

@polynom\double.m Convert polynom object to a coefficient vector.

@polynom\display.m An overloaded method to display polynom objects.

[1] Most of these files were taken from MATLAB online help.

@polynom\subsref.m An overloaded subscript reference method to evaluate numerical values of a polynom object at a set of values given by vector.

@polynom\plus.m An overloaded polynom addition method.

@polynom\minus.m An overloaded polynom subtraction method.

@polynom\mpower.m A simple implementation of p^q, where p is a polynom, and $q \geq 0$ is an integer.

@polynom\mtimes.m An overloaded polynom multiplication method.

@polynom\roots.m an overloaded polynom root finder.

@polynom\polyval.m An overloaded polynom `polyval` method.

@polynom\diff.m An overloaded method for polynom differentiation.

@polynom\plot.m An overloaded plot method to plot a polynom object.

In order to fully understand this chapter, please download the MATLAB files listed above.

Following MATLAB Help, we shall create a *class* polynom. The *objects* in this class will be polynomials of any order with the coefficients specified by us. We shall implement several *methods* that act upon these polynomials. These methods will allow us to display, add, subtract, multiply and differentiate polynomials, as well as to plot them in a predefined way. Still other methods will allow us to evaluate the polynomials with vector arguments and calculate their roots. You can easily create your own methods to get the polynomial coefficients, integrate the polynomials, etc. Note that all the method files are included in the subdirectory of the Chapter 28 directory named @polynom. The calling function `PolynomDriver.m` resides in the Chapter 28 directory.

28.2 Why Is It Important?

Think about the following. Will the software you develop survive a lifetime of change as requirements change and technology evolves? Or will it become woefully inadequate and obsolete before you are promoted or move to another job? It has always been the case that even the smallest piece of code, once delivered, needs to be modified to fit the ever-changing needs. Today, the cost of software modification dominates the cost of development. Just think about all the buggy software you buy everyday. The reasons for the bugs are many, but the common reason is that the code is never easy to understand and its documentation is never complete and up to date. As a result, we spend way too much time trying to understand old, often badly written code. On the other hand, the code represents an investment which is more cost effective to reuse than to rewrite. The object-oriented methods that we shall barely begin to scratch in this course, make the code reusable and flexible the future. The price we pay up front is more work in algorithm design and code development; in short, more thinking.

28.3 Classes and Objects

There are two ways of constructing a diagram, computer program or another model of the world around us. The *concrete* way is to just draw a single diagram or code the computer description of just the objects we want to model today. The *abstract* way is to consider not individual objects but classes to which they belong. *Class* and *object* are two important concepts. They are *not* the same and the distinction is fundamental. An object will always be an instance of a class. It is therefore possible to refer to all the instances of a particular class.

You can view classes as new data types having specific behaviors defined for the class. For example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. This redefinition is called *overloading*. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as overloading the operator.

28.4 Example: A Polynomial Class

In this example we shall implement a new MATLAB data type for polynomials by defining a new class called polynom. The class definition specifies a structure for data storage and defines a directory (@polynom) of functions (methods) that operate on polynom objects.

28.4.1 Polynom Data Structure

The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in *decreasing* order. Therefore, a polynom object p is a structure with a single field, p.c, containing the coefficients. This field is accessible only within the methods in the @polynom directory.

28.4.2 Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method `polynom.m`
- A polynom to double converter, `double.m`
- A polynom to char converter, `char.m`
- A display method, `display.m`
- A subscripted-reference, *subsref*, method, `subsref.m`
- Overloaded $+$, $-$, and matrix $*$ operators, `plus.m`, `minus.m` and `mtimes.m`
- Overloaded root finder, `roots.m`
- Overloaded polynomial evaluation, `polyval.m`
- Overloaded plotting method, `plot.m`, and
- Overloaded differentiation method, `diff.m`

Polynom constructor method

Here is the simple code:

```
function p = polynom(a)
% POLYNOM: Polynomial class constructor.
%
%   p = POLYNOM(a) creates a polynomial object from the vector a,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

You can call the polynom constructor method with one of three different arguments:

No input argument - If you call the constructor function with no arguments, it returns a polynom object with empty fields.

Input argument is an object - If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The isa function (pronounced "is a") checks for this situation.

Input argument is a coefficient vector - If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the .c field of the object's structure. The class function creates the polynom object, which is then returned by the constructor.

An example use of the polynom constructor is the statement

```
>> p = polynom([1 0 -2 -5])
```

This creates a polynomial with the specified coefficients.

Note that the syntax

```
p.c = a(:).'
```

converts an arbitrary array into a row vector column-by-column, e.g.,

```
>> a = [1,2,3,4;5,6,7,8]

>> a =
      1  2  3  4
      5  6  7  8

>> b = a(:).'
>> b =
      1  5  2  6  3  7  4  8
```

Polynom to double converter

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are double and char. Conversion to double produces MATLAB's traditional matrix, although this may not be appropriate for some classes.

The double converter method for the polynom class is a very simple M-file, @polynom/double.m, which merely retrieves the coefficient vector:

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

Polynom to char converter

Conversion to char is useful for producing printed output.The converter to char is a key method because it produces a character string involving the powers of an independent variable, x. Therefore, once you have specified x, the string returned is a *syntactically correct* MATLAB expression, which you can then evaluate.

Here is @polynom/char.m:

```
function s = char(p,b)
% POLYNOM/CHAR, Modified by TWP, 2002-05-06
% CHAR(p) is the string representation of p.c
% For example, 2*b^2 + 2*b + 3
if nargin < 2
   b='x';
else
   if ~isa(b,'char')
      error('Power base must be a character')
   end
end
if all(p.c == 0)
    s = '0';
else
    d = length(p.c) - 1; % Degree of polynomial
    s = [];
    for a = p.c;         % Loop through all coefficients
        if a ~= 0;
            if ~isempty(s)
                 if a > 0
                     s = [s ' + '];
                 else
                     s = [s ' - '];
                     a = -a;
                 end
            end
            if a ~= 1 | d == 0
                 s = [s num2str(a)];
                 if d > 0
                     s = [s '*'];
                 end
            end
            if d >= 2
                 s = [s b '^' int2str(d)];
            elseif d == 1
                 s = [s b];
            end
        end
        d = d - 1;
    end
end
```

If you create the polynom object p

```
 >> p = polynom([1 0 -2 -5]);
```

and then call the char method on p

char(p)

MATLAB produces the result

```
 ans =
      x^3-2*x-5
```

The value returned by char is a string that you can pass to eval once you have defined a scalar value for x. For example,

```
>> x = 3;
>> eval(char(p))
   ans =
        16
```

Polynom display method

Here is @polynom/display.m. This method relies on the char method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```
function display(p,b)
% POLYNOM/DISPLAY Command window display of a polynom
% Modified by TWP, 2002-05-06
disp(' ');
% Display name of the input variable and equal sign
disp([inputname(1),' = '])
% Blank line
disp(' ');
% Display the character conversion of p
if nargin < 2
   disp(['   ' char(p)])
elseif nargin == 2
   disp(['   ' char(p,b)])
end
disp(' ');
```

Polynom subscripted reference method

Suppose the design of the polynom class specifies that a subscripted reference to a polynom object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object p,

```
 p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at x = 3 and x = 4.

```
  >> p([3 4])
     ans =
          16 51
```

This implementation takes advantage of the char method already defined in the polynom class to produce an expression that can then be evaluated.

```
function b = subsref(a,s)
%B = subsref(A,S) is called with the syntax
%    A(i), A{i}, or A.i when A is an object.
%    S is a structure array with the following fields:
%
%       S.type: A string containing ’()’, ’{}’, or ’.’, where
%               ’()’ specifies numeric subscripts
%               ’{}’ specifies cell array subscripts, and
%               ’.’ specifies subscripted structure fields
%
%       S.subs: A cell array or string containing the actual
%               values of subscripted variable
%
% Function "subscript reference" evaluates polynomial a
% at a set of values given by vector s.  The calling
% sequence could be x=xmin:step;xmax; p(x)
%
%fprintf(’\na=%s\n’,char(a))
%s

switch s.type
case ’()’
   ind = s.subs{:};
   for i = 1:length(ind)
% string search and replace:
% str = strrep(str1,str2,str3) replaces all occurrences
% of string >>str2<< within string >>str1<< with
% string >>str3<<
%
% eval executes a string containing a MATLAB expression
%
%                                str1    str2       str3
      b(i) = eval(strrep( char(a), ’x’,  num2str(ind(i)) ));
      %strrep( char(a), ’x’,  num2str(ind(i)) )
   end
otherwise
   error(’Specify value for x as p(x)’)
end
```

Once the polynomial expression has been generated by the char method, the strrep function is used to swap the passed in value for the character x. The eval function then evaluates the expression and returns the value in the output argument.

polynom $+$, $-$, $*$ operators

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, the plus, minus, and mtimes methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

If either p or q is a polynom, the expression

$$\mathsf{p} + \mathsf{q}$$

generates a call to a function @polynom/plus.m, if it exists (unless p or q is an object of a higher precedence,see below.)

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.
p = polynom(p); % Convert object p to a polynom
q = polynom(q); % Convert object q to a polynom
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

$$\mathsf{p} + 1$$

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the polynom constructor a third time to create the properly typed result.

You can implement the overloaded minus operator (-) using the same approach as the plus (+) operator. MATLAB calls @polynom/minus.m to compute $p - q$.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynoms.
p = polynom(p); % Convert object p to a polynom
q = polynom(q); % Convert object q to a polynom
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

MATLAB calls the method @polynom/mtimes.m to compute the product $p * q$. The letter m at the beginning of the function name comes from the fact that it is overloading MATLAB's *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.
p = polynom(p); % Convert object p to a polynom
q = polynom(q); % Convert object q to a polynom
r = polynom(conv(p.c,q.c));
```

Polynom root finder

MATLAB already has several functions for working with polynomials represented by coefficient vectors. They should be overloaded to also work with the new polynom object. In many cases, the overloading methods can simply apply the original function to the coefficient field. The method @polynom/roots.m finds the roots of polynom objects:

```
function r = roots(p)
% POLYNOM/ROOTS.  ROOTS(p) is a vector containing the roots of p.
r = roots(p.c);
```

Polynom evaluation method

The function polyval evaluates a polynomial at a given set of points. @polynom/polyval.m uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
   y = y.*x + a;
end
```

Polynom plot method

Here is the overloaded @polynom/plot.m function

```
function plot(p)
% POLYNOM/PLOT  PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Polynom differentiation method

The method @polynom/diff.m differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
p = polynom(p); % Convert object p to a polynom
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Polynom integration method

The method @polynom/integral.m integrates a polynomial by reducing the degree by 1 and dividing each coefficient by its original degree+1. It is your task to construct this method.

Listing class methods

The function call

methods('class_name')

or its command form

methods class_name

shows all the methods available for a particular class.

28.5 Object Precedence

Object precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

```
objectA + objectB
```

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

1. User-defined classes have *precedence* over MATLAB built-in classes.

2. User-defined classes can specify their relative precedence with respect to other user-defined classes using the inferiorto and superiorto functions.

For example, in section 28.4.2, the polynom class defines a plus method that enables addition of polynom objects. Given the polynom object p

```
 p = polynom([1 0 -2 -5])
 p =
    x^3-2*x-5
```

The expression,

```
1 + p
ans =
    x^3-2*x-4
```

calls the polynom plus method (which converts the double, 1, to a polynom object, and then adds it to p). The user-defined polynom class has precedence over the MATLAB `double` class.

28.5.1 Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferiorto` or `superiorto` function in the class constructor.

The `inferiorto` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferiorto` function is

```
inferiorto('class1','class2',...)
```

You can specify multiple classes in the argument list, placing the class below many other classes in the hierarchy.

Similarly, the `superiorto` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superiorto` function is

```
superiorto('class1','class2',...)
```

Location in the Hierarchy

If objectA is above objectB in the precedence hierarchy, then the expression

```
objectA + objectB
```

calls @classA/plus.m. Conversely, if objectB is above objectA in the precedence hierarchy, then MATLAB calls @classB/plus.m.

28.6 Followup

Go to http://www.geocities.com/tablizer/oopbad.htm and read about some of the pitfalls of Object-Oriented Programming (OOP).

Chapter 29

The Greatest Minds Behind the Modern Computer

29.1 What Are You Going To Learn?

In the Introduction, I told you that computers were incredibly stupid, but lightening fast *abaci*. After several months of struggle, you probably discovered that, yes, your computer is stupid and inflexible, but when properly instructed it can do pretty incredible things. It can draw precise geometrical objects, rotate them in three dimensions, and zoom down to the finest details. It can solve thousands of linear equations, large systems of nonlinear ordinary differential equations, sort millions of numbers and characters in a blink of an eye, and find a number among millions of other numbers faster than you can read this sentence. So how is it that a computer is at the same time so "stupid" and so "smart"? The secret lies in the computer's past... The purpose of this chapter is to tell you a short story about the giant men of the past (yes, they were all men) whose unrelenting pursuit of perfection and truth in science made the present day computer such an amazing device.

For more information please go to an absolutely lovely and clever book, *The Universal Computer: The Road from Leibnitz to Turing* [7] by MARTIN DAVIS, a professor of logic retired from Berkeley. You should also visit the *History of Mathematics* website `http://www-groups.dcs.st-and.ac.uk/~history/index.html`.

29.2 Why Is It Important?

You cannot appreciate how computers came to be the way they are without some knowledge of their distinguished past.

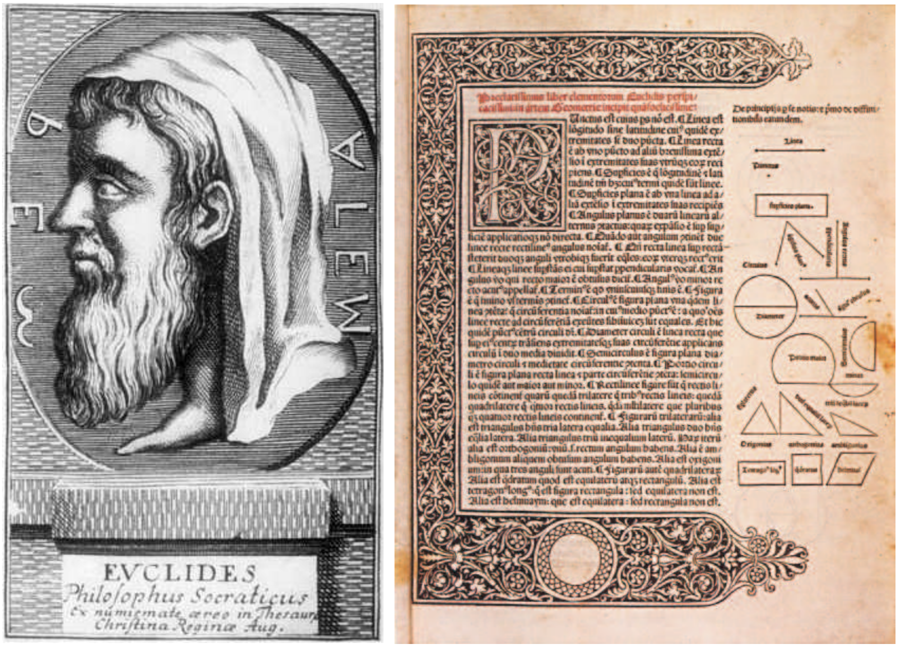


Figure 29.1: EUCLID was born *ca* 325 B.C., and died *ca* 265 in Alexandria, Egypt. **Right**: A page from the first of more than a thousand editions of EUCLID's *Elements* (Venice, 1482). Printing revolutionized the study of mathematics since it could mass-produce accurate calculations and diagrams.

Figure 29.2: René Descartes was born on 31 March 1596 in La Haye (now Descartes), Touraine, France, and died on 11 February 1650 in Stockholm, Sweden. In one of three appendices to his treatise *Discours de la méthod pour bien conduire sa raison et chercher la vérité dans les sciences*, entitled *La géométrie*, Descartes applied algebra to geometry and created what we call now *Cartesian geometry*.

Figure 29.3: Gottfried Wilhelm von Leibniz was born on 1 July 1646 in Leipzig, Saxony (now Germany), and died on 14 November 1716 in Hanover, Hanover (now Germany). Leibnitz was a great builder of clear notation in mathematics, second perhaps only to Euler. He conceived of a language whose words would be ideas not symbols. Another of Leibniz's great achievements in mathematics was his development of the binary system of arithmetic.

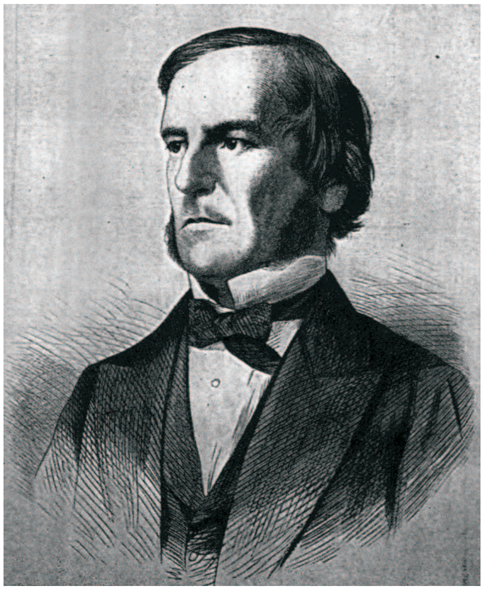


Figure 29.4: George Boole was born on 2 November 1815 in Lincoln, Lincolnshire, England, and died on 8 December 1864 in Ballintemple, County Cork, Ireland. In 1854 he published *An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities*. Boole approached logic in a new way reducing it to simple algebra. He pointed out the analogy between algebraic symbols and those that represent logical forms. Thus began the algebra of logic called *Boolean algebra* which now finds application in computer architecture and languages.

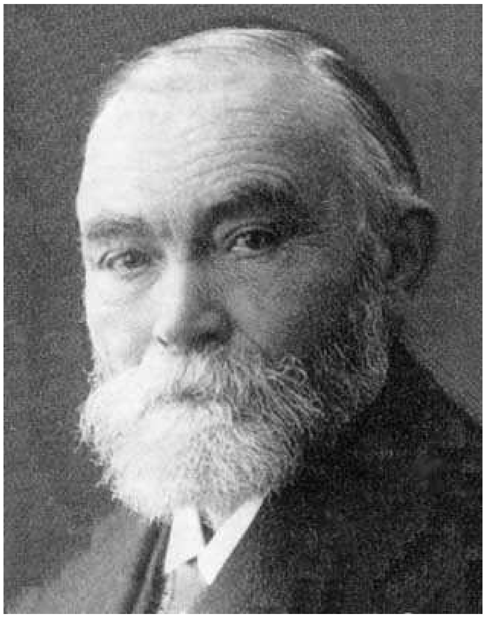


Figure 29.5: GOTTLOB FREGE was born on 8 November 1848 in Wismar, Mecklenburg-Schwerin (now Germany), and died on 26 July 1925 in Bad Kleinen, Germany. He was the first to attempt to show that all mathematics is *reducible to logic.* His works, *The Foundations of Arithmetic* (1884), and *The Basic Laws of Arithmetic,Volume 1* (1893) were devoted to this subject. When reading FREGE's work, BERTRAND RUSSELL formulated his famous paradox of *extraordinary sets* and caused FREGE a lot of grief. In Europe, FREGE is considered the father of PROLOG, a logic predicate-based computer language.

Figure 29.6: Georg Cantor was born on 3 March 1845 in St. Petersburg, Russia, and died on 6 January 1918 in Halle, Germany. Cantor was the first mathematician to quantify what happens when one is not merely passing to infinity, but *is* at infinity. Are all infinities the same, or are some more infinite than others? In 1873 Cantor proved the rational numbers to be countable, i.e., they may be placed in one-one correspondence with the natural numbers. He also showed that the algebraic numbers, i.e., the numbers which are roots of polynomial equations with integer coefficients, were countable. A transcendental ("real") number is an irrational number that is not a root of any polynomial equation with integer coefficients. In his 1874 work, Cantor showed that in a certain sense *almost all* numbers are transcendental by proving that the real numbers were not countable while he had proved that the algebraic numbers were countable. His work had a huge impact on the future computer science.

Figure 29.7: DAVID HILBERT was born on 23 January 1862 in Königsberg, Prussia (now Kaliningrad, Russia), and died on 14 February 1943 in Göttingen, Germany. HILBERT's work in geometry had the greatest influence in that area after EUCLID. A systematic study of the axioms of Euclidean geometry led HILBERT to propose 21 such axioms and analyze their significance. He published *Grundlagen der Geometrie* in 1899 putting geometry in a formal axiomatic setting. By proving that all geometry can be reduced to algebra, HILBERT greatly influenced the future computer science.

Figure 29.8: KURT GÖDEL was born on 28 April 1906 in Brünn, Austria-Hungary (now Brno, Czech Republic), and died on 14 January 1978 in Princeton, New Jersey, USA. GÖDEL is best known for his proof of *Incompleteness Theorems*. In 1931 he published these results in *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*. He showed that in any axiomatic mathematical system there are propositions that *cannot* be proved or disproved within the axioms of the system. In particular the consistency of the axioms cannot be proved. GÖDEL showed that a (future) computer can never be programmed to answer all mathematical questions.

Figure 29.9: ALAN MATHISON TURING was born on 23 June 1912 in London, England, and died on 7 June 1954 in Wilmslow, Cheshire, England. TURING studied GÖDEL's incompleteness results and HILBERT's question on *decidability*: given a mathematical proposition could one find an algorithm which would decide if the proposition was true or false. For many propositions it was easy to find such an algorithm. The real difficulty arose in proving that for certain propositions no such algorithm existed. When given an algorithm to solve a problem it was clear that it was indeed an algorithm, yet there was no definition of an algorithm which was rigorous enough to allow one to prove that none existed. In 1936 he published *On Computable Numbers*, with an application to the *Entscheidungsproblem*. In this paper TURING introduced an abstract machine, now called a *Turing machine*, which moved from one state to another using a precise finite set of rules (given by a finite table) and depending on a single symbol it read from a tape. This machine was an abstract prototype of the future digital computer.

Figure 29.10: JOHN (JÁNOS) VON NEUMANN was born on 28 December 1903 in Budapest, Hungary, and died on 8 February 1957 in Washington D.C., USA. He became one of the original six mathematics professors (J. W. Alexander, A. Einstein, M. Morse, O. Veblen, J. von Neumann and H. Weyl) in 1933 at the newly founded *Institute for Advanced Study* in Princeton, a position he kept for the remainder of his life. VON NEUMANN was one of the pioneers of computer science making significant contributions to the development of logical design. He advanced the theory of *cellular automata*, advocated the adoption of the *bit* as a measurement of computer memory, and solved problems in obtaining reliable answers from unreliable computer components. Today's digital computer architecture was conceived in great part by VON NEUMANN.

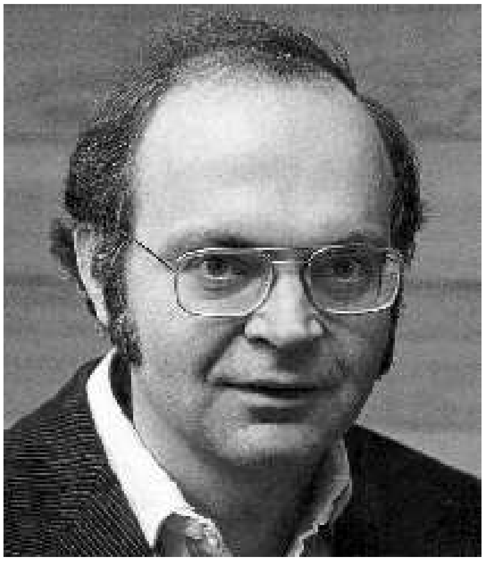


Figure 29.11: DONALD KNUTH was born 10 January 1938 in Milwaukee, Wisconsin, USA. KNUTH's seminal three-volume monograph on the ART OF COMPUTING has made a large impact on computer science in general, and this book in particular. His major contribution, which has totally changed the whole way that mathematics is printed and communicated is the invention of TEX, a language for typesetting mathematical and scientific articles. TEX has changed the technology of mathematics and science publishing since it enables mathematicians and scientists to produce the highest quality of printing of mathematical articles, yet this can be achieved simply by using a home computer. However, TEX has not only changed how mathematical and scientific articles are published but also how mathematics is communicated. In the 17th century, a mathematician would have written a letter to another mathematician, and they would discuss their everyday lives in English, French or German, say, but whenever they came to explain a piece of mathematics they would use Latin. Now mathematicians do not speak Latin, communicate by e-mail, and whenever they want to explain a piece of mathematics they require mathematical symbols, which almost always they communicate using TEX. This book was typeset using a TEX derivative LATEX.

Bibliography

[1] G. J. Borse, editor. *Numerical Methods in MATLAB, A Resource for Scientists and Engineers.* PWS Publishing company, Boston, 1997.

[2] C. A. Boyer and U. C. Merzbach. *A History of Mathematics.* John Wiley & Sons, Inc., New York, Second edition, 1991.

[3] F. W. Byron and R. W. Fuller. *Mathematics of Classical and Quantum Physics*, volume 1. Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[4] A. Cayley. The Newton-Fourier imaginary problem. *American Journal of Mathematics*, 2:97, 1879.

[5] K. K. Chakrabarti. *Definition and Induction - A Historical and Comparative Study.* University of Hawai'i Press, Honolulu, 1995.

[6] P. Coulette and C. Tresser. Iteration d'endomorphismes et groupe de renormalisation. *J. De Physique Colloque C*, 539:C5–25, 1978.

[7] M. Davis. *The Universal Computer: The Road from Leibnitz to Turing.* W.W. Norton & Company, New York, 1st edition, 2000.

[8] J. W. Demmel. *Applied Numerical Analysis.* SIAM, Philadelphia, 1 edition, 1997.

[9] N. R. Draper and H. Smith. *Applied Regression Analysis.* Wiley Series on Probability and Statistics. John Wiley & Sons, Inc., New York, Third edition, 1998.

[10] J. L. Esposito, editor. *The Oxford History of Islam.* Oxford University Press, Oxford, 1999.

[11] M. Feigenbaum. Quantitative universality for a class of nonlinear transformations. *J. Stat. Phys.*, 19:25–52, 1978.

[12] R. P. Feynman. *Feynman lectures on computation, Edited by T. Hay and R. W. Allen.* The Advanced Book Program. Perseus Publishing, Cambridge, Massachusetts, 1996.

[13] Booch. G. *Object-Oriented Analysis and Design With Applications.* Addison-Wesley, Redding, Massachusetts, 2 edition, 1994.

[14] T. H. Garland. *Fascinating Fibonaccis - Mystery and Magic in Numbers.* Dale Seymour Publications, 1987.

[15] L. L. Gaston. *The unity of knowledge.* The Columbia University Bicentennial Conference Series. Doubleday, Garden City, NY, 1955.

[16] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, 1996.

[17] A. S. Householder. *The theory of matrices in numerical analysis.* Dover, New York, 1975.

[18] H. E. Huntley. *The Divine Proportion - A Study of Mathematical Beauty.* Dover Publications, Inc., New York, 1970.

[19] G. Ifrah. *The universal history of numbers: From prehistory to the invention of the computer.* John wiley & Sons, Inc., New York, 2000.

[20] D. E. Knuth. *The Art of Computer Programming*, volume 1 / Fundamantal Algorithms. Addison-Wesley, Reading, Massachusetts, 3 edition, 1997.

[21] D. E. Knuth. *The Art of Computer Programming*, volume 3 / Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 3 edition, 1997.

[22] J. Mandel. *The Statistical Analysis of Experimental Data.* Dover, 1964.

[23] W. Mendenhall and T. Sincich. *A Second Course in Business Statistics: Regression Analysis.* Prentice Hall, New York, Fourth edition, 1992.

[24] D. S. Moore and McCabe G. P. *Introduction to the Practice of Statistics.* W.H. Freeman and Co., New York, Second edition, 1993.

[25] J. W. III Palm. *Introduction to Matlab 7 for engineers.* McGraw-Hill, Boston, 2004.

[26] D. Peak and M. Frame. *Chaos Under Control: The Art and Science of Complexity.* W. H. Freeman Company, New York, 1994.

[27] H.-O. Peitgen, H. Jürgens, and Saupe D. *Fractals for the Classroom, Part I: Introduction to Fractals and Chaos.* Springer-Verlag, New York, 1992.

[28] W. H. Press, Teukolsky S. A., W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The art of scientific computing.* Cambridge University Press, Cambridge, Second edition, 1992.

[29] Th. Roszak. *The cult of information, a neo-Luddite treatise on high-tech, artificial intelligence, and the true art of thinking.* University of California Press, Berkeley, Second edition, 1994.

[30] B. C. van Fraassen. *Formal Semantics and Logic.* Macmillan, New York, 1971.

[31] P. F. Verhulst. Notice sur la loi que la population suit dans son accroissement. *Corr. Math. et Phys. publ. par A. Quetelet*, T. X. (also numbered T. II of the third series):113–121, 1838.

[32] Webster. *Webster's Third New International Dictionary of the English Language - Unabridged.* Encyclopædia Britannica, Inc., Chicago, 1993.