

# Numerical Analysis in MATLAB

Radu T. Trîmbițăș



---

## Preface

---

This work is an introductory course for undergraduate students. It is intended to achieve a balance between theory, algorithms and practical implementation aspects.

Lloyd N. Trefethen [95] proposed the following definition of Numerical Analysis:

*Numerical Analysis is the study of algorithms for the problems of continuous mathematics.*

The keyword here is that of *algorithms*, and the main purpose of Numerical Analysis is devising and analyzing numerical algorithms to solve a certain class of problems.

These are the problems of *continuous mathematics*. “Continuous” means here that input data are real and complex variables; its opposite is discrete. Shortly, one could say that Numerical analysis is continuous algorithmics, as opposed to classical algorithmics, that is discrete algorithmics.

Since real and complex numbers cannot be represented exactly on computers, they must be approximated using a finite representation (floating-point representation and arithmetic). Moreover, most problems of continuous mathematics cannot be solved by the so-called finite algorithms, even we assume an infinite precision arithmetic. A first example is the solution of nonlinear algebraic equations. This becomes more clear in eigenvalue and eigenvector problems. The same conclusion extends to virtually any problem with a nonlinear term or a derivative in it – zero finding, numerical quadrature, differential equations, integral equations, optimization, and so on. Thus we introduce various type of *errors*; their study is an important task of numerical analysis. Chapter 3, *Errors and Floating Point Arithmetic* is devoted to this topic.

As we mention previously, Numerical Analysis mean to find and analyze algorithms for constructive mathematical problems. Rapid convergence of approximations is the aim, and the pride of our field is that, for many problems, we have invented algorithms that converge exceedingly fast. The development of computer algebra software like Maple or Mathematica diminished the importance of rounding errors without diminishing the importance of algo-

rithms speed of convergence. Chapters 4 to 10 study various class of numerical algorithms, as follows:

- Chapter 4 – *Numerical Solution of Linear Algebraic Systems*;
- Chapter 5 – *Function Approximation*;
- Chapter 6 – *Linear Functional Approximation*;
- Chapter 7 – *Numerical Solution of Nonlinear Equations*;
- Chapter 8 – *Eigenvalues and Eigenvectors*;
- Chapter 9 – *Numerical Solution of Ordinary Differential Equations*;
- Chapter 10 – *Multivariate Approximation*.

There are other important aspects: that numerical algorithms are implemented on computers, whose architecture may be an important part of the problem; that reliability and efficiency are paramount goals; and most important, all this work is *applied*, applied daily and successfully to thousands of applications on millions of computer around the world. Without numerical methods, science and engineering as practiced today cease to exist. Numerical analysts proudly consider themselves the heirs to the great traditions of Newton, Euler, Lagrange, Gauss and other great mathematicians.

Also, in [94], Trefethen wrote:

*A computational study is unlikely to lead to real scientific progress unless the software environment is convenient enough to encourage one to vary parameters, modify the problem, play around.*

In this context, MATLAB is an excellent choice as software support. It allows us to focuss on essence of numerical algorithms, to rapidly prototype them, to obtain compact code and high accuracy, and to vary the problems and their parameters. The first two chapters are an introduction to MATLAB.

We gave MATLAB implementations for algorithms in the book and tested them carefully. Each chapter, excepting the first, contains fully implemented applications of the notions presented within them. The sources in this book and solutions to problems can be downloaded by following the links from the author's web page: <http://www.math.ubbcluj.ro/~tradu>.

Radu Tiberiu Trîmbițăș  
Cluj-Napoca, July 2008

---

## Contents

---

<b>1</b>	<b>Introduction to MATLAB</b>	<b>1</b>
1.1	Starting MATLAB and MATLAB Help . . . . .	2
1.2	Calculator Mode . . . . .	3
1.3	Matrices . . . . .	6
1.3.1	Matrix generation . . . . .	7
1.3.2	Indexing and colon operator . . . . .	11
1.3.3	Matrix and array operations . . . . .	13
1.3.4	Data analysis . . . . .	16
1.3.5	Relational and Logical Operators . . . . .	18
1.3.6	Sparse matrices . . . . .	21
1.4	MATLAB Programming . . . . .	23
1.4.1	Control flow . . . . .	23
1.5	M files . . . . .	25
1.5.1	Scripts and functions . . . . .	25
1.5.2	Subfunctions, nested and anonymous functions . . . . .	28
1.5.3	Passing a Function as an Argument . . . . .	30
1.5.4	Advanced data structures . . . . .	32
1.5.5	Variable Number of Arguments . . . . .	37
1.5.6	Global variables . . . . .	38
1.5.7	Recursive functions . . . . .	39
1.5.8	Error control . . . . .	40
1.6	Symbolic Math Toolboxes . . . . .	41
	Problems . . . . .	48
<b>2</b>	<b>MATLAB Graphics</b>	<b>51</b>
2.1	Two-Dimensional Graphics . . . . .	51
2.1.1	Basic Plots . . . . .	51

2.1.2	Axes and Annotation . . . . .	57
2.1.3	Multiple plots in a figure . . . . .	60
2.2	Three-Dimensional Graphics . . . . .	61
2.3	Handles and properties . . . . .	68
2.4	Saving and Exporting Graphs . . . . .	69
2.5	Application - Snail and Shell Surfaces . . . . .	70
	Problems . . . . .	71
<b>3</b>	<b>Errors and Floating Point Arithmetic</b>	<b>73</b>
3.1	Numerical Problems . . . . .	74
3.2	Error Measuring . . . . .	74
3.3	Propagated error . . . . .	75
3.4	Floating-Point Representation . . . . .	76
3.4.1	Parameters . . . . .	76
3.4.2	Cancelation . . . . .	78
3.5	IEEE Standard . . . . .	80
3.5.1	Special Quantities . . . . .	80
3.6	MATLAB Floating-Point Arithmetic . . . . .	81
3.7	The Condition of a Problem . . . . .	85
3.8	The Condition of an algorithm . . . . .	87
3.9	Overall error . . . . .	88
3.10	Ill-Conditioned Problems and Ill-Posed Problems . . . . .	89
3.11	Stability . . . . .	90
3.11.1	Asymptotical notations . . . . .	90
3.11.2	Accuracy and stability . . . . .	91
3.11.3	Backward Error Analysis . . . . .	93
3.12	Applications . . . . .	93
3.12.1	Inverting the hyperbolic cosine . . . . .	93
3.12.2	Conditioning of a root of a polynomial equation . . . . .	95
	Problems . . . . .	98
<b>4</b>	<b>Numerical Solution of Linear Algebraic Systems</b>	<b>101</b>
4.1	Notions of Matrix Analysis . . . . .	101
4.2	Condition of a linear system . . . . .	107
4.3	Gaussian Elimination . . . . .	112
4.4	Factorization based methods . . . . .	116
4.4.1	LU decomposition . . . . .	116
4.4.2	LUP decomposition . . . . .	117
4.4.3	Cholesky factorization . . . . .	119
4.4.4	QR decomposition . . . . .	121
4.5	Strassen's algorithm for matrix multiplication . . . . .	124
4.6	Solution of Algebraic Linear Systems in MATLAB . . . . .	126
4.6.1	Square systems . . . . .	126
4.6.2	Overdetermined systems . . . . .	127
4.6.3	Underdetermined systems . . . . .	128

4.6.4	LU and Cholesky factorizations . . . . .	129
4.6.5	QR factorization . . . . .	131
4.6.6	The <code>linsolve</code> function . . . . .	133
4.7	Iterative refinement . . . . .	134
4.8	Iterative solution of Linear Algebraic Systems . . . . .	135
4.9	Applications . . . . .	141
4.9.1	The finite difference method for linear two-points boundary value problem	141
4.9.2	Computing a plane truss . . . . .	145
	Problems . . . . .	147
<b>5</b>	<b>Function Approximation</b>	<b>151</b>
5.1	Least Squares approximation . . . . .	154
5.1.1	Inner products . . . . .	154
5.1.2	The normal equations . . . . .	155
5.1.3	Least squares error; convergence . . . . .	158
5.2	Examples of orthogonal systems . . . . .	160
5.3	Examples of orthogonal polynomials . . . . .	163
5.3.1	Legendre polynomials . . . . .	163
5.3.2	First kind Chebyshev polynomials . . . . .	167
5.3.3	Second kind Chebyshev polynomials . . . . .	173
5.3.4	Laguerre polynomials . . . . .	174
5.3.5	Hermite polynomials . . . . .	174
5.3.6	Jacobi polynomials . . . . .	174
5.3.7	A MATLAB example . . . . .	175
5.4	Polynomials and data fitting in MATLAB . . . . .	177
5.4.1	An application — Census Data . . . . .	183
5.5	The Space $H^n[a, b]$ . . . . .	184
5.6	Polynomial Interpolation . . . . .	187
5.6.1	Lagrange interpolation . . . . .	188
5.6.2	Hermite Interpolation . . . . .	192
5.6.3	Interpolation error . . . . .	195
5.7	Efficient Computation of Interpolation Polynomials . . . . .	199
5.7.1	Aitken-type methods . . . . .	199
5.7.2	Divided difference method . . . . .	201
5.7.3	Barycentric Lagrange Interpolation . . . . .	205
5.7.4	Multiple nodes divided differences . . . . .	210
5.8	Convergence of polynomial interpolation . . . . .	212
5.9	Spline Interpolation . . . . .	215
5.9.1	Interpolation by cubic splines . . . . .	217
5.9.2	Minimality properties of cubic spline interpolants . . . . .	222
5.10	Interpolation in MATLAB . . . . .	223
5.11	Applications . . . . .	227
5.11.1	Spline and sewing machines . . . . .	227
5.11.2	A membrane deflection problem . . . . .	229
	Problems . . . . .	231

<b>6 Linear Functional Approximation</b>	<b>235</b>
6.1 Introduction . . . . .	235
6.1.1 Method of interpolation . . . . .	237
6.1.2 Method of undetermined coefficients . . . . .	238
6.2 Numerical Differentiation . . . . .	239
6.3 Numerical Integration . . . . .	240
6.3.1 The composite trapezoidal and Simpson's rule . . . . .	241
6.3.2 Weighted Newton-Cotes and Gauss formulae . . . . .	245
6.3.3 Properties of Gaussian quadrature rules . . . . .	248
6.4 Adaptive Quadratures . . . . .	253
6.5 Iterated Quadratures. Romberg Method . . . . .	255
6.6 Adaptive Quadratures II . . . . .	259
6.7 Numerical Integration in MATLAB . . . . .	260
6.8 Applications . . . . .	264
6.8.1 Computation of an ellipsoid surface . . . . .	264
6.8.2 Computation of the wind action on a sailboat mast . . . . .	266
Problems . . . . .	268
<b>7 Numerical Solution of Nonlinear Equations</b>	<b>271</b>
7.1 Nonlinear Equations . . . . .	271
7.2 Iterations, Convergence, and Efficiency . . . . .	271
7.3 Sturm Sequences Method . . . . .	273
7.4 Method of False Position . . . . .	275
7.5 Secant Method . . . . .	278
7.6 Newton's method . . . . .	280
7.7 Fixed Point Iteration . . . . .	286
7.8 Newton's Method for Multiple zeros . . . . .	287
7.9 Algebraic Equations . . . . .	288
7.10 Newton's method for systems of nonlinear equations . . . . .	288
7.11 Quasi-Newton Methods . . . . .	291
7.11.1 Linear Interpolation . . . . .	292
7.11.2 Modification Method . . . . .	292
7.12 Nonlinear Equations in MATLAB . . . . .	295
7.13 Applications . . . . .	298
7.13.1 Analysis of the state equation of a real gas . . . . .	298
7.13.2 Nonlinear heat transfer in a wire . . . . .	300
Problems . . . . .	301
<b>8 Eigenvalues and Eigenvectors</b>	<b>305</b>
8.1 Eigenvalues and Polynomial Roots . . . . .	305
8.2 Basic Terminology and Schur Decomposition . . . . .	306
8.3 Vector Iteration . . . . .	309
8.4 QR Method – the Theory . . . . .	311
8.5 QR Method – the Practice . . . . .	315
8.5.1 Classical QR method . . . . .	315

8.5.2	Spectral shift . . . . .	320
8.5.3	Double shift QR method . . . . .	322
8.6	Eigenvalues and Eigenvectors in MATLAB . . . . .	326
8.7	Applications . . . . .	330
8.7.1	Solving mass-spring systems . . . . .	330
8.7.2	Computing natural frequencies of a rectangular membrane . . . . .	333
	Problems . . . . .	336
<b>9</b>	<b>Numerical Solution of Ordinary Differential Equations</b>	<b>339</b>
9.1	Differential Equations . . . . .	339
9.2	Numerical Methods . . . . .	340
9.3	Local Description of One-Step Methods . . . . .	341
9.4	Examples of One-Step Methods . . . . .	342
9.4.1	Euler's method . . . . .	342
9.4.2	Method of Taylor expansion . . . . .	344
9.4.3	Improved Euler methods . . . . .	345
9.5	Runge-Kutta Methods . . . . .	346
9.6	Global Description of One-Step Methods . . . . .	350
9.6.1	Stability . . . . .	353
9.6.2	Convergence . . . . .	355
9.6.3	Asymptotics of global error . . . . .	356
9.7	Error Monitoring and Step Control . . . . .	358
9.7.1	Estimation of global error . . . . .	358
9.7.2	Truncation error estimates . . . . .	360
9.7.3	Step control . . . . .	362
9.8	ODEs in MATLAB . . . . .	372
9.8.1	Solvers . . . . .	372
9.8.2	Nonstiff examples . . . . .	373
9.8.3	Options . . . . .	375
9.8.4	Stiff equations . . . . .	378
9.8.5	Event handling . . . . .	385
9.8.6	<code>deval</code> and <code>odextend</code> . . . . .	391
9.8.7	Implicit equations . . . . .	392
9.9	Applications . . . . .	392
9.9.1	The restricted three-body problem . . . . .	392
9.9.2	Motion of a projectile . . . . .	396
	Problems . . . . .	399
<b>10</b>	<b>Multivariate Approximation</b>	<b>407</b>
10.1	Interpolation in Higher Dimensions . . . . .	407
10.1.1	Interpolation problem . . . . .	407
10.1.2	Cartesian product and grid . . . . .	407
10.1.3	Boolean sum and tensor product . . . . .	409
10.1.4	Geometry . . . . .	414
10.1.5	A Newtonian scheme . . . . .	416

10.1.6 Shepard interpolation . . . . .	417
10.1.7 Triangulation . . . . .	422
10.1.8 Moving least squares . . . . .	424
10.1.9 Interpolation by radial basis functions . . . . .	426
10.2 Multivariate Numerical Integration . . . . .	428
10.3 Multivariate Approximations in MATLAB . . . . .	433
10.3.1 Multivariate interpolation in MATLAB . . . . .	433
10.3.2 Computing double integrals in MATLAB . . . . .	436
Problems . . . . .	439
<b>Bibliography</b>	<b>441</b>
<b>Index</b>	<b>446</b>

---

## List of MATLAB sources

---

1.1	The first MATLAB script - playingcards . . . . .	26
1.2	The <code>stat</code> function . . . . .	27
1.3	Function <code>mysqrt</code> . . . . .	29
1.4	Function <code>fd_deriv</code> . . . . .	31
1.5	Function <code>companb</code> . . . . .	37
1.6	Function <code>moments</code> . . . . .	38
1.7	Recursive <code>gcd</code> . . . . .	40
2.1	Epicycloid . . . . .	58
2.2	Snail/Shell surface . . . . .	70
3.1	Computation of <code>eps</code> - 1st variant . . . . .	82
3.2	Computation of <code>eps</code> - 2nd variant . . . . .	82
3.3	Conditioning of the roots of an algebraic equation . . . . .	96
4.1	Solve the system $Ax = b$ by Gaussian elimination with scaled column pivoting.	115
4.2	LUP Decomposition . . . . .	118
4.3	Forward substitution . . . . .	118
4.4	Back substitution . . . . .	119
4.5	Solution of a linear system by LUP decomposition . . . . .	119
4.6	Cholesky Decomposition . . . . .	121
4.7	QR decomposition using Householder reflections . . . . .	123
4.8	Solution of $Ax = b$ using QR method . . . . .	124
4.9	Strassen's algorithm for matrix multiplication . . . . .	126
4.10	Jacobi method for linear systems . . . . .	138
4.11	Successive Overrelaxation method (SOR) . . . . .	140
4.12	Finding optimal value of relaxation parameter . . . . .	141
4.13	Two point boundary value problem – finite difference method . . . . .	143
4.14	Two point boundary value problem – finite difference method and symmetric matrix	144
5.1	Compute Legendre polynomials using recurrence relation . . . . .	165
5.2	Compute Legendre coefficients . . . . .	166

5.3	Least square approximation via Legendre polynomials . . . . .	166
5.4	Leat squares continuous approximation with Chebyshev # 1 polynomials . . . . .	171
5.5	Compute Chebyshev # 1 polynomials by mean of recurrence relation . . . . .	172
5.6	Least squares approximation with Chebyshev # 1 polynomials – continuation: computing Fourier coefficients . . . . .	172
5.7	Discrete Chebyshev least squares approximation . . . . .	172
5.8	The coefficients of discrete Chebyshev least squares approximation . . . . .	173
5.9	Test for least squares approximations . . . . .	176
5.10	An example of least squares approximation . . . . .	185
5.11	Lagrange Interpolation . . . . .	189
5.12	Find basic Lagrange interpolation polynomials using MATLAB facilities. . . . .	190
5.13	Generate divided difference table . . . . .	203
5.14	Compute Newton's form of Lagrange interpolation polynomial . . . . .	204
5.15	Barycentric Lagrange interpolation . . . . .	208
5.16	Barycentric weights . . . . .	208
5.17	Barycentric Lagrange interpolation with Chebyshev nodes of first kind . . . . .	209
5.18	Barycentric Lagrange interpolation with Chebyshev nodes of second kind . . . . .	209
5.19	Generates a divided difference table with double nodes . . . . .	213
5.20	Runge's Counterexample . . . . .	214
6.1	Composite trapezoidal rule . . . . .	242
6.2	Composite Simpson formula . . . . .	243
6.3	Compute nodes and coefficients of a Gaussian quadrature rule . . . . .	250
6.4	Approximation of an integral using a Gaussian formula . . . . .	251
6.5	Generate a Gauss-Legendre formula . . . . .	251
6.6	Generate a first kind Gauss-Chebyshev formula . . . . .	251
6.7	Generate a second kind Gauss-Chebyshev formula . . . . .	251
6.8	Generate a Gauss-Hermite formula . . . . .	252
6.9	Generate a Gauss-Laguerre formula . . . . .	252
6.10	Generate a Gauss-Jacobi formula . . . . .	252
6.11	Adaptive quadrature . . . . .	254
6.12	Romberg method . . . . .	258
6.13	Adaptive quadrature, variant . . . . .	261
6.14	Computation of the wind action on a sailboat mast . . . . .	267
7.1	False position method for nonlinear equation in $\mathbb{R}$ . . . . .	276
7.2	Secant method for nonlinear equations in $\mathbb{R}$ . . . . .	281
7.3	Newton method for nonlinear equations in $\mathbb{R}$ . . . . .	284
7.4	Newton method in $\mathbb{R}$ and $\mathbb{R}^n$ . . . . .	290
7.5	Broyden method for nonlinear systems . . . . .	294
8.1	The RQ transform of a Hessenberg matrix . . . . .	316
8.2	Reduction to upper Hessenberg form . . . . .	318
8.3	Pure (simple) QR Method . . . . .	318
8.4	<b>QRSplit1a</b> – QR method with partition and treatment of $2 \times 2$ matrices . . . . .	320
8.5	Compute eigenvalues of a $2 \times 2$ matrix . . . . .	321
8.6	QR iterations on a Hessenberg matrix . . . . .	321
8.7	Spectral shift QR method, partition and treatment of complex eigenvalues . . . . .	323
8.8	QR iteration and partition . . . . .	324

8.9	Double shift QR method with partition and treating $2 \times 2$ matrices . . . . .	325
8.10	Double shift QR iterations and Hessenberg transformation . . . . .	326
9.1	Classical 4th order Runge-Kutta method . . . . .	349
9.2	Implementation of a Runge-Kutta method with constant step given Butcher table	351
9.3	Initialize Butcher table for RK4 . . . . .	351
9.4	Rössler system . . . . .	377
9.5	Stiff problem with information about Jacobian . . . . .	386
9.6	Two body problem . . . . .	389
10.1	Bivariate tensor product Lagrange interpolant . . . . .	412
10.2	Bivariate Boolean sum Lagrange interpolant . . . . .	412
10.3	Shepard interpolation . . . . .	419
10.4	Local Shepard interpolation . . . . .	421
10.5	Interpolation with radial basis function . . . . .	428
10.6	Double integral approximation on a rectangle . . . . .	434



# CHAPTER 1

---

## Introduction to MATLAB

---

MATLAB<sup>1</sup> is a high-performance interactive software system for technical computing and data visualization. It integrates programming, computation, and visualization in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.

The name MATLAB stands for MATrix LABoratory. MATLAB was originally written by Cleve Moler to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB engines incorporate the LAPACK and BLAS libraries, embedding the state of the art in software for matrix computation.

In many university environments, MATLAB became the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry and research establishments, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB has several advantages over classical programming languages:

- It allows quick and easy coding in a very high-level language.
- Its interactive interface allows rapid experimentation and easy debugging.
- It is a versatile tool for modeling, simulation, and prototyping and also for analysis, exploration, and visualization of data.
- High-quality graphics and visualization facilities are available. They allow you to plot the results of computations with a few statements.

---

<sup>1</sup>MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

- Application development, including implementation of GUIs (Graphical User's Interfaces) can be easily done.
- MATLAB M-files are completely portable across a wide range of platforms.
- A wide range of user-contributed M-files is freely available on the Internet.

The basic data structure of MATLAB language is an array that does not require dimensioning. This allows flexibility and easiness during the solution of problems, especially those with matrix and vector formulations. MATLAB capabilities save the time of developing, testing and prototyping algorithms. But, the language has sophisticated data structures and object-oriented facilities, allowing development of large, complex applications. Having and interpreted languages, MATLAB inevitably suffers some loss of efficiency compared with compiled languages. This can be partially fixed by using the MATLAB Compiler or by linking to compiled Fortran or C code using MEX files.

Besides the language and the development environment, MATLAB provides a set of add-on application-specific solutions called toolboxes. Toolboxes allow the users to learn and apply specialized technology. They are collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, statistics and many others.

For a gentle and detailed introduction see [58, 57, 44]. An excellent source for both MATLAB and numerical computing is Moler's book "Numerical Computing in MATLAB" [66]. A very good short to medium introduction to MATLAB is [27].

## 1.1 Starting MATLAB and MATLAB Help

To start the MATLAB program on a Microsoft Windows platform, select and click the MATLAB entry from start menu, or double-click the shortcut icon for MATLAB on your Windows desktop. To start the MATLAB program on UNIX platforms, type `matlab` at the operating system prompt. When you start MATLAB you get a multipaneled *desktop* (Figure 1.1). The layout and the behavior of the desktop and its components are highly customizable. The main component is the *Command Window*. Here you can give MATLAB commands typed at the prompt, `>>`.

At the top left you can see the *Current Directory*. In general MATLAB is aware only of files in the current directory (folder) and on its path, which can be customized. Commands for work with directory and path include `cd`, `what`, `addpath`, and `editpath` (or you can choose "File/Set Path..." from the menus).

At top right is the *Workspace* window. The workspace shows you what variable names are currently defined and some information about their contents. (At start-up it is, naturally, empty.) This represents another break from compiled environments: variables created in the workspace persist for you to examine and modify, even after code execution stops. Another component is the *Command History* window. As you enter commands, they are recorded here. This record persists across different MATLAB sessions, and commands or blocks of commands can be copied from here or saved to files.

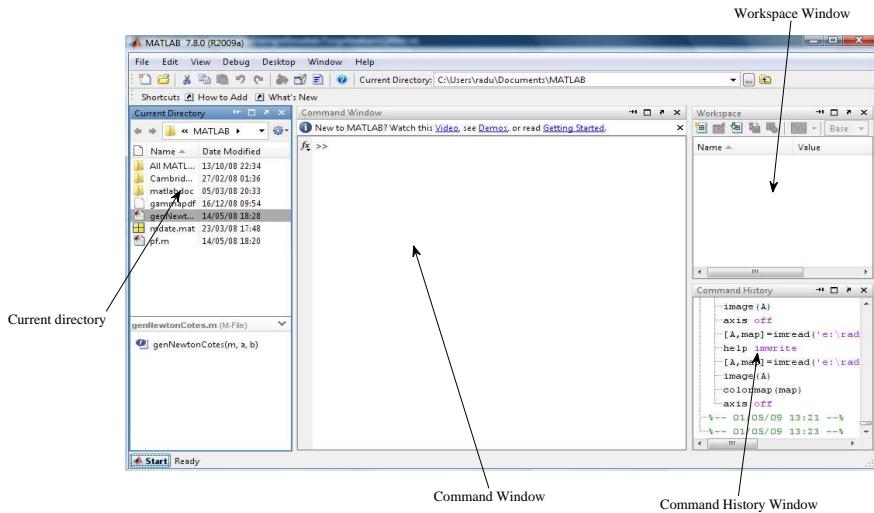


Figure 1.1: MATLAB desktop

MATLAB is a huge software. Nobody can know everything about it. To have success, it is essential that you become familiar with the online help. There are two levels of help:

- If you need quick help on the syntax of a command, use `help` command. For example, `help plot` shows right in the Command Window all the ways in which you can use the `plot` command. Typing `help` by itself gives you a list of categories that themselves yield lists of commands.
  - Typing `doc` followed by a command name brings up more extensive help in a separate window. MATLAB help browser is a fully functional web browser. For example, `doc plot` is better formatted and more informative than `help plot`. In the left panel one sees a hierarchical, browsable display of all the online documentation. Typing `doc` alone or selecting Help from the menu brings up the window at a “root” page.

A lot of useful documentation is available in printable form, by following the links from the MathWorks Inc. web page ([www.mathworks.com](http://www.mathworks.com)).

## 1.2 Calculator Mode

If you type in a valid expression and press Enter, MATLAB will immediately execute it and return the result, just like a calculator.

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

```
>> 3^2
ans =
    9
>> sin(pi/2)
ans =
    1
```

The basic arithmetic operations are  $+$   $-$   $*$   $/$  and  $^$  (power). You can alter the default precedence with parentheses.

MATLAB recognizes several number types:

- Integers, like 1362 or -217897;
- Reals, for example, 1.234, -10.76;
- Complex numbers, like  $3.21 - 4.3i$ , where  $i = \sqrt{-1}$ ;
- Inf, denotes infinity;
- NaN, Not a Number, result of an operation that is not mathematically defined or is illegal, like  $0/0$ ,  $\infty/\infty$ ,  $\infty - \infty$ , etc. Once generated, a NaN propagates through all subsequent computations.

The scientific notation is also used:

$$\begin{aligned}-1.3412e + 03 &= -1.3412 \times 10^3 = -1341.2 \\ -1.3412e - 01 &= -1.3412 \times 10^{-1} = -0.13412\end{aligned}$$

MATLAB carries out all its arithmetic computations in double precision floating point arithmetic, conforming to the IEEE standard. The `format` command controls the way in which numbers are displayed. Type `help format` for a complete list. The next table gives some examples.

Command	Output examples
<code>format short</code>	31.4162
<code>format short e</code>	31.416e+01
<code>format long e</code>	3.141592653589793e+000
<code>format short g</code>	31.4162
<code>format bank</code>	31.42

The `format compact` command suppresses blank lines and allows to display more information. All MATLAB session examples in this work were generated using this command.

MATLAB variable names are case sensitive and consists of a letter followed by any combination of letters, digits and underscores. Examples: `x`, `y`, `z525`, `GeneralTotal`. Variables must have values before they can be used. There exist some special names:

- `eps` =  $2.2204e-16 = 2^{-54}$  is machine-epsilon (see Chapter 3); it is the largest floating point number with the property that  $1+eps==1$ ;

- `pi = π.`

If we perform computations with complex numbers, the usage of variable names `i` and `j` is not recommended, since they denote the imaginary unit. It is safer to use `1i` or `1j` instead. We give some examples:

```
>> 5+2i-1+i
ans =
    4.0000 + 3.0000i
>>eps
ans =
2.2204e-016
```

The special variable `ans` store the value of last evaluated expression. It can be used in expressions, like any other variable.

```
>>3-2^4
ans =
-13
>>ans*5
ans =
-65
```

To suppress the display of last evaluated expression, end the expression by a semicolon (`;`). We can type several expression in a single line. We can separate them by comma, when its value is displayed, or by a `“;”`, when its value is not displayed.

```
>> x=-13; y = 5*x, z = x^2+y, z2 = x^2-y;
y =
-65
z =
104
```

MATLAB has a rich set of elementary and mathematical functions. Type `help elfun` and `help matfun` to obtain the full list. A selection is listed in Table 1.1.

Once variables have been defined, they exist in the workspace. You can see the list of all variables used in current session by typing `whos`:

```
>>whos
  Name      Size            Bytes  Class
  ans       1x1                  8  double array
  i         1x1                  8  double array
  v         1x3                 24  double array
  x         1x1                  8  double array
  y         1x1                  8  double array
  z         1x1                  8  double array
  z2        1x1                  8  double array
Grand total is 7 elements using 72 bytes
```

An existing variable `var` can be removed from the workspace by typing `clear var`, while `clear` clears all existing variables.

To print the value of a variable or expression without the name of the variable or `ans` being displayed, you can use `disp`:

<code>cos, sin, tan, csc, sec, cot acos, asin, atan, atan2, asec, acsc, acot cosh, sinh, tanh, sech, csch, coth acosh, asinh, atanh, asech,acsch, acoth log, log2, log10, exp, pow2, nextpow2 ceil, fix, floor, round abs, angle, conj, imag, real mod, rem, sign</code>	Trigonometric Inverse trigonometric  Hyperbolic Inverse hyperbolic  Exponential Rounding Complex Remainder, sign
<code>airy, bessel*, beta*, erf*, expint, gamma*, legendre</code>	Mathematical
<code>factor, gcd, isprime, lcm, primes, nchoosek, perms, rat, rats</code>	Number theoretic
<code>cart2sph, cart2pol, pol2cart, sph2cart</code>	Coordinate transforms

Table 1.1: Elementary and special mathematical functions (“fun\*” indicates that more than one function name begins “fun”).

```
>> X=ones(2); disp(X)
      1      1
      1      1
```

If we want to save our work, we can do it with

```
>>save file-name variable-list
```

where variables din variable-list are separated by spaces. We can use wildcards, like `a*` in variable-list. It save variables in variable-list in `file-name.mat`. If variable-list is missing, all variables in the workspace are saved to `file-name.mat`. A `.mat` file is a binary file, peculiar to MATLAB. The command `load`

```
>>load file-name
```

loads variables from a `.mat` file. We can do savings and loadings in ascii, in ascii and double precision, or by adding to an existing file. For details help `save` and help `load` (or doc `save` and doc `load`).

Often you need to capture MATLAB output for incorporation into a report. The command

```
>>diary file-name
```

copies all commands and their results (excepting graphs) to file `file-name`; `diary off` end the diary process. You can later type `diary on` to cause subsequent output to be appended to the same diary file.

## 1.3 Matrices

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. Matrix is a fundamental data type in MATLAB.

An *array* is a collection of elements or entries, referenced by one or more indices running over different index sets. In MATLAB, the index sets are always sequential integers starting with 1. The dimension of the array is the number of indices needed to specify an element. The size of an array is a list of the sizes of the index sets. A *matrix* is a two-dimensional array with special rules for specific operations. With few exceptions, its entries are double-precision floating point numbers. A *vector* is a matrix for which one dimension has only the index 1. A row vector has only one row and a column vector has only one column.

The terms matrix and array are often used interchangeable.

### 1.3.1 Matrix generation

The simplest way to construct a matrix is by enclosing its elements in square brackets. Elements are separated by spaces or commas, and rows by semicolons and new lines.

```
>> A = [5 7 9
1 -3 -7]
A =
      5      7      9
      1     -3     -7
>> B = [-1 2 5; 9 0 5]
B =
     -1      2      5
      9      0      5
>> C = [0, 1; 3, -2; 4, 2]
C =
      0      1
      3     -2
      4      2
>> x=[7, 1, 3, 2]
x =
      7      1      3      2
>>
```

Information about size and dimension is stored with the array. For this reason, array sizes are not usually passed explicitly to functions as they are in C, Pascal or FORTRAN. You can use `size` to obtain information about the size of a matrix. Other useful commands are `length` that returns the size of longest dimension and `ndims` that gives the number of dimensions.

```
>> v = size(A)
v =
      2      3
>> [r, c] = size(A)
r =
      2
c =
      3
>> ndims(A)
ans =
      2
```

<code>zeros</code>	Zeros array
<code>ones</code>	Ones array
<code>eye</code>	Identity matrix
<code>repmat</code>	Replicate and tile array
<code>rand</code>	Uniformly distributed random numbers
<code>randn</code>	Normally distributed random numbers
<code>linspace</code>	Vector of equally spaced elements
<code>logspace</code>	Logarithmically spaced vector

Table 1.2: Matrix generation functions

```
>> length(C), length(x)
ans =
    3
ans =
    4
>> ndims(x)
ans =
    2
```

One special array is the empty matrix entered as [ ].

Bracket constructions are not suitable for larger matrices. Many elementary matrices can be generated with MATLAB functions; see Table 1.2. Examples:

```
>> zeros(2) %like zero(2,2)
ans =
    0     0
    0     0
>> ones(2,3)
ans =
    1     1     1
    1     1     1
>> eye(3,2)
ans =
    1     0
    0     1
    0     0
```

If we wish to generate a zeros, ones or identity matrix, having the same size as a given matrix A, you can use the size function as argument, like in `eye(size(A))`.

The `rand` and `randn` functions generates matrices of (pseudo) random numbers from the uniform distribution on [0,1] and standard normal distribution, respectively. Examples:

```
>> rand
ans =
    0.4057
>> rand(3)
ans =
    0.9355    0.8936    0.8132
```

```
0.9169    0.0579    0.0099
0.4103    0.3529    0.1389
```

In simulations or in experiments with random number it is important to reproduce the random numbers on a subsequent occasion. The generated random numbers depends on the status of the generator. The method and the status can be set with `rand(method, s)`. It causes `rand` to use the generator determined by `method`, and initializes the state of that generator using the value of `s`. We give some examples.

Set `rand` to its default initial state:

```
rand('twister', 5489);
```

Initialize `rand` to a different state each time:

```
rand('twister', sum(100*clock));
```

Save the current state, generate 100 values, reset the state, and repeat the sequence:

```
s = rand('twister');
u1 = rand(100);
rand('twister',s);
u2 = rand(100); % contains exactly the same values as u1
```

Matrices can be built out of other arrays (in block form) as long as their sizes are compatible. With `B`, defined by `B=[1 2; 3 4]`, we may create

```
>> C=[B, zeros(2); ones(2), eye(2)]
C =
 1     2     0     0
 3     4     0     0
 1     1     1     0
 1     1     0     1
```

An example with incompatible sizes:

```
>> [A;B]
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

Several commands are available for manipulating matrices; see Table 1.3.

`diag`, `tril`, and `triu` functions consider that the diagonal of the matrix they act on is numbered as follows: main diagonal has number 0, the  $k$ th diagonal above the main diagonal has number  $k$ , and the  $k$ th diagonal below the main diagonal has number  $-k$ ,  $k = 0, \dots, n$ . If `A` is a matrix, `diag(A, k)` extracts the diagonal numbered by  $k$  in a vector. If `A` is a vector, `diag(A, k)` constructs a diagonal matrix, whose  $k$ th diagonal consists of elements of `A`.

```
>> diag([1,2],1)
ans =
 0     1     0
 0     0     2
 0     0     0
>> diag([3 4],-2)
```

<code>reshape</code>	Change size
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>blkdiag</code>	Block diagonal matrix
<code>tril</code>	Extract lower triangular part
<code>triu</code>	Extract upper triangular part
<code>fliplr</code>	Flip matrix in left/right direction
<code>flipud</code>	Flip matrix in up/down direction
<code>rot90</code>	Rotate matrix 90 degrees

Table 1.3: Matrix manipulation functions

```
ans =
    0     0     0     0
    0     0     0     0
    3     0     0     0
    0     4     0     0
```

If

```
A =
    2     3     5
    7    11    13
   17    19    23
```

then

```
>> diag(A)
ans =
    2
    11
    23
>> diag(A,-1)
ans =
    7
    19
```

The commands `tril(A, k)` and `triu(A, k)` extracts the lower and the upper triangular part, respectively, of their argument, starting from diagonal `k`. For `A` given above,

```
>>tril(A)
ans =
    2     0     0
    7    11     0
   17    19    23
>>triu(A,1)
ans =
    0     3     5
    0     0    13
    0     0     0
```

compan	Companion matrix
gallery	Large collection of test matrices
hadamard	Hadamard matrix
hankel	Hankel matrix
hilb	Hilbert matrix
invhilb	Inverse Hilbert matrix
magic	Magic square
pascal	Pascal matrix (binomial coefficients)
rosser	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz Matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

Table 1.4: Special matrices

```
>> triu(A, -1)
ans =
    2     3     5
    7    11    13
    0    19    23
```

MATLAB provides a number of special matrices; see Table 1.4. These matrices have interesting properties that make them useful for constructing examples and for testing algorithms. We shall give examples with `hilb` and `vander` functions in Section 4.2. The `gallery` function provides access to a large collection of test matrices created by Nicholas J. Higham [46]. For details, see `help gallery`.

### 1.3.2 Indexing and colon operator

To enable access and assignment to *submatrices* MATLAB has a powerful notation based on the colon character, “`:`”. This is an important array constructor. The format is `first : step:last`, and the result is a row vector (which may be empty). Examples:

```
>> 1:5
ans =
    1     2     3     4     5
>> 4:-1:-2
ans =
    4     3     2     1     0    -1    -2
>> 0:.75:3
ans =
    0    0.7500    1.5000    2.2500    3.0000
```

Single elements of a matrix are accessed as `A(i, j)`, where  $i \geq 1$  and  $j \geq 1$  (zero or negative subscripts are not supported in MATLAB). The submatrix comprising the intersection of rows  $p$  to  $q$  and columns  $r$  to  $s$  is denoted by `A(p:q, r:s)`. As a special case, a lone colon as the row or column specifier covers all entries in that row or column; thus `A(:, j)` is the

$j$ th column of  $A$  and  $A(i, :)$  the  $i$ th row. The keyword `end` used in this context denotes the last index in the specified dimension; thus  $A(end, :)$  picks out the last row of  $A$ . Finally, an arbitrary submatrix can be selected by specifying the individual row and column indices. For example,  $A([i \ j \ k], [p \ q])$  produces the submatrix given by the intersection of rows  $i, j$  and columns  $p$  and  $q$ .

Here are some example working on the matrix

```
A =
    1      2      3
    7      5      9
   17     15     21

>> A(2,1)
ans =
    7
>> A(2:3,2:3)
ans =
    5      9
   15     21
>> A(:,1)
ans =
    1
    7
   17
>> A(2,:)
ans =
    7      5      9
>> A([1 3],[2 3])
>> A([1 3],[2 3])
ans =
    2      3
   15     21
```

The special case  $A(:)$  denotes a vector comprising all the elements of  $A$  taken down the columns from first to last:

```
>> B=A(:)
B =
    1
    7
   17
    2
    5
   15
    3
    9
   21
```

Any array can be accessed by a single script. Multidimensional arrays are actually stored linearly in memory, varying over the first dimension, then the second, and so on. (Think of

the columns of a table being stacked on top of each other.) In this sense the array is equivalent to a vector, and a single subscript will be interpreted in this context.

```
>> A(5)
ans =
    5
```

Subscript referencing can be used on either side of assignments.

```
>> B=ones(2, 3)
B =
    1     1     1
    1     1     1
>> B(1,:)=A(1,:)
B =
    1     2     3
    1     1     1
>> C=rand(2, 5)
C =
    0.1576    0.9572    0.8003    0.4218    0.7922
    0.9706    0.4854    0.1419    0.9157    0.9595
>> C(2,:)=-1      %expand scalar into a submatrix
C =
    0.1576    0.9572    0.8003    0.4218    0.7922
   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
>> C(3,1) = 3      %create a new row to make space
C =
    0.1576    0.9572    0.8003    0.4218    0.7922
   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
    3.0000        0        0        0        0
```

The empty matrix [ ] is useful to delete lines or columns

```
>> C(:,4)=[]
C =
    0.1576    0.9572    0.8003    0.7922
   -1.0000   -1.0000   -1.0000   -1.0000
    3.0000        0        0        0
```

The empty matrix is also useful in a function call to indicate a missing argument, as we shall see in §1.3.4.

An array is resized automatically if you delete elements or make assignments outside the current size. (Any new undefined elements are made zero.) But attention, this might be a source of hard-to-find errors.

### 1.3.3 Matrix and array operations

The arithmetic operators +, -, \*, /, are interpreted in a matrix sense. When appropriate, scalars are “expanded” to match a matrix. In addition to the usual division operator /, with the meaning  $A/B = AB^{-1}$ , MATLAB has a left division operator, (backslash \), with the meaning  $A\backslash B = A^{-1}B$ . The backslash and the forward slash define solutions of linear systems:  $A\backslash B$  is a solution  $X$  of  $A*X = B$ , while  $A/B$  is a solution  $X$  of  $X*B = A$ . Examples:

Operation	Matrix sense	Array sense
Addition	+	+
Subtraction	-	-
Multiplication	*	.*
Left division	\	.\ .
Right division	/	./
Exponentiation	^	.^

Table 1.5: Elementary matrix and array operations

```

>> A=[1 2; 3 4], B=ones(2)
A =
    1     2
    3     4
B =
    1     1
    1     1
>> A+B
ans =
    2     3
    4     5
>> A*B
ans =
    3     3
    7     7
>> A\B
ans =
    -1    -1
    1     1
>> A+2
ans =
    3     4
    5     6
>> A^3
ans =
    37    54
    81   118

```

Array operations simply act identically on each element of an array. Multiplication, division and exponentiation in array or elementwise sense are specified by preceding the operator with a period. If  $A$  and  $B$  are matrices of the same dimensions then  $C = A.*B$  sets  $C(i, j) = A(i, j) * B(i, j)$  and  $C = B.\A$  sets  $C(i, j) = B(i, j) / A(i, j)$ . With the same  $A$  and  $B$  as in the previous example:

```

>>A.*B
ans =
    1     2
    3     4

```

```
>>B./A
ans =
    1.0000    0.5000
    0.3333    0.2500
>> A.^2
ans =
    1      4
    9     16
```

The dot form of exponentiation allows the power to be an array when the dimension of the base and the power agree, or when the base is a scalar:

```
>>x=[1 2 3]; y=[2 3 4]; z=[1 2; 3 4];

>>x.^y
ans =
    1      8      81

>>2.^x
ans =
    2      4      8

>>2.^z
ans =
    2      4
    8     16
```

To invert a nonsingular matrix one uses `inv`, and for the determinant of a square matrix `det`.

Matrix exponentiation is defined for all powers, not just for positive integers. If  $n < 0$  is an integer then  $A^n$  is defined as `inv(A)^( -n )`. For noninteger  $p$ ,  $A^p$  is evaluated using the eigensystem of  $A$ ; results can be incorrect or inaccurate when  $A$  is not diagonalizable or when  $A$  has an ill-conditioned eigensystem.

The conjugate transpose of the matrix  $A$  is obtained with  $A'$ . If  $A$  is real this is simply the transpose. The transpose without conjugation is obtained with  $A.'$ . The functional alternatives `ctranspose(A)` and `transpose(A)` are sometimes more convenient. Table 1.5 gives a list of matrix and array operations.

For the special case of column vectors  $x$  and  $y$ ,  $x' * y$  is the inner or dot product, which can also be obtained using the `dot` function as `dot(x, y)`. The vector or cross product of two 3-by-1 or 1-by-3 vectors (as used in mechanics) is produced by `cross`. Example:

```
>>x=[-1 0 1]'; y=[3 4 5]';
>>x'*y
ans =
    2
>>dot(x,y)
```

max	Largest component
min	Smallest component
mean	Average or mean value
median	Median value
std	Standard deviation
var	Variance
sort	Sort
sum	Sum of elements
prod	Product of elements
cumsum	Cumulative sum of elements
cumprod	Cumulative product of elements
diff	Difference of elements

Table 1.6: Basic data analysis functions

```
ans =
2

>>cross(x,y)
ans =
-4
8
-4
```

### 1.3.4 Data analysis

Table 1.6 lists functions for basic data analysis computation. The simplest usage is to apply these functions to vectors. For example

```
>>x=[ 4 -8 -2 1 0 ]
x =
    4      -8      -2       1       0

>>[min(x) max(x) ]
ans =
-8      4

>>sort(x)
ans =
-8      -2       0       1       4

>>sum(x)
ans =
-5
```

The `sort` function sorts into ascending order. For complex vectors, `sort` sorts by absolute value and so descending order must be obtained by explicitly reordering the output:

```
>>x=[1+i -3-4i 2i 1];
>>y=sort(x)
>>y=y(end:-1:1)
y =
-3.0000 - 4.0000i 0 + 2.0000i 1.0000 + 1.0000i 1.0000
```

For matrices the functions are defined columnwise. Thus `max` and `min` return a vector containing the maximum and the minimum element, respectively, in each column, `sum` returns a vector containing the column sums, and `sort` sorts the elements in each column of the matrix in ascending order. The function `min` and `max` can return a second argument that specifies in which components the minimum and the maximum elements are located. For example, if

```
A =
0      -1      2
1       2     -4
5      -3     -4
```

then

```
>>max(A)
ans =
5      2      2

>>[m, i]=min(A)
m =
0      -3     -4
i =
1      3      2
```

As this example shows if there are two or more minimal elements in a column, then the index of the first is returned. The smallest element in the matrix can be found by applying `min` twice in succession:

```
>>min(min(A))
ans =
-4
```

An alternative way is typing

```
>>min(A(:))
ans =
-4
```

The `diff` function computes differences. Applied to a vector  $x$  of length  $n$  it produces the vector  $[x(2)-x(1) \ [x(3)-x(2) \ \dots \ x(n)-x(n-1)]$  of length  $n-1$ . Applied to a matrix, it acts columnwise. Example

```
>>x=(1:8).^2
x =
```

```

1      4      9      16      25      36      49      64

>>y=diff(x)
y =
3      5      7      9      11      13      15

```

For details like sorting in descending order, acting rowwise, how the function not described here work, see the corresponding helps or docs.

### 1.3.5 Relational and Logical Operators

MATLAB relational operators are: == (equal), ~= (not equal), < (less than), > (greater than), <= (less than or equal) and >= (greater than or equal). Note that a single = denotes assignment, not an equality test.

Comparisons between scalars produce 1 if the relation is true and 0 if it is false. Comparisons are defined between matrices of the same dimensions and between a matrix and a scalar, the result being a matrix of 0s and 1s in both cases. For matrix-matrix comparisons corresponding pairs of elements are compared, while for matrix-scalar comparisons the scalar is compared with each matrix element. For example:

```

>> A=[1 2; 3 4]; B = 2*ones(2);

>> A == B
ans =
0      1
0      0

>> A > 2
ans =
0      0
1      1

```

To test whether matrices A and B are identical, the expression `isequal(A, B)` can be used:

```

>> isequal(A, B)
ans =
0

```

There are many useful logical function whose names begin with `is`; for a full list type `doc is`.

MATLAB's logical operators are &(and), |(or), ~(not), xor(exclusive or), all(true if all elements of vector is nonzero), any(true if any element of vector is nonzero). They produce matrices of 0s and 1s when one of the arguments is a matrix. Examples

```

>> x = [-1 1 1]; y = [1 2 -3];
>> x>0 & y>0
ans =
0      1      0
>> x>0 | y>0

```

Precedence level	Operator
1 (highest)	Transpose ( $\cdot'$ ), power( $\cdot^{\wedge}$ ), complex conjugate transpose ( $\cdot'$ ), matrix power( $\wedge$ )
2	Unary plus (+), unary minus (-), logical negation ( $\sim$ )
3	Multiplication ( $\cdot \ast$ ), right division ( $\cdot /$ ), left division ( $\cdot \backslash$ ), matrix multiplication (*), matrix right division (/), matrix left division (\)
4	Addition (+), subtraction (-)
5	Colon operator (:)
6	Less than (<), less than or equal to ( $\leq$ ), greater than (>), greater than or equal to ( $\geq$ ), equal to ( $\equiv$ ), not equal to ( $\neq$ )
7	Logical and ( $\&$ )
8 (lowest)	Logical or ( $\mid$ )

Table 1.7: Operator precedence

```

ans =
    1      1      1
>> xor(x>0,y>0)
ans =
    1      0      1
>> any(x>0)
ans =
    1
>> all(x>0)
ans =
    0

```

The precedence of arithmetic, relational and logical operators is summarized in table 1.7 (which is based on the information provided by `help precedence`). For operators of equal precedence MATLAB evaluates from left to right. Precedence can be overridden by using parentheses.

The `find` command returns the indices corresponding to the nonzero elements of a vector. For example,

```

>> x = [-3 1 0 -inf 0 NaN];
>> f = find(x)
f =
    1      2      4      6

```

The result of `find` can then be used to extract just those elements of the vector:

```

>> x(f)
ans =
    -3      1     -Inf     NaN

```

With `x` as above, we can use `find` to obtain the finite and nonNaN elements of `x`,

```
>> x(find(isfinite(x) & ~isnan(x)))
ans =
-3     1     0     0
```

and to replace the negative components of  $x$  by zero:

```
>> x(find(x<0))=0
x =
0     1     0     0     0
```

When `find` is applied to a matrix  $A$ , the index vector corresponds to  $A(:)$ , and this vector can be used to index into  $A$ . In the following example we use `find` to set to zero those elements of  $A$  that are less than the corresponding elements of  $B$ :

```
>> A = [4 2 16; 12 4 3], B = [12 3 1; 10 -1 7]
A =
    4      2      16
   12      4      3
B =
    12      3      1
    10     -1      7
>> f = find(A<B)
f =
    1
    3
    6
>> A(f) = 0
A =
    0      0      16
   12      4      0
```

An alternative usage of `find` for matrices is `[i, j] = find(A)`, which returns vectors  $i$  and  $j$  containing the row and column indices of nonzero elements.

```
>> [i, j]=find(B<3); [i, j]
ans =
    2      2
    1      3
```

The results of MATLAB's logical operators and logical functions are array of 0s and 1s that are examples of logical arrays. Logical arrays can also be created by applying the function `logical` to a numeric array. Logical arrays can be used for subscripting. The expression  $A(M)$ , where  $M$  is a logical array of the same dimension as  $A$ , extracts the elements of  $A$  corresponding to the elements of  $M$  with nonzero real part.

```
>> B = [-1 2 5; 9 0 5]
B =
   -1      2      5
    9      0      5
>> la=B>3
la =
```

```

0      0      1
1      0      1
>> B(1a)
ans =
9
5
5
>> b=B(2,:)
b =
9      0      5
>> x=[1, 1, 1]; xl=logical(x);
>> b(x)
ans =
9      9      9
>> b(xl)
ans =
9      0      5
>> whos x*
  Name      Size            Bytes  Class       Attributes
  x            1x3             24  double
  xl           1x3              3  logical

```

### 1.3.6 Sparse matrices

You can create two-dimensional double and logical matrices using one of two storage formats: full or sparse. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems. Many real world matrices are both extremely large and very sparse, meaning that most entries are zero. In such cases it is wasteful or downright impossible to store every entry. Instead one can keep a list of nonzero entries and their locations. MATLAB has a sparse data type for this purpose. The `sparse` and `full` commands convert back and forth and lay bare the storage difference.

```

>> sparse(A)
ans =
(1,1)      1
(2,1)      4
(3,1)      9
(1,2)      1
(2,2)      2
(3,2)      3
(1,3)      1
(2,3)      1
(3,3)      1
>> clear
>> A = vander(1:3);
>> S = sparse(A);
>> full(S)

```

```
ans =
    1     1     1
    4     2     1
    9     3     1
```

Sparsifying a standard full matrix is usually not the right way to create a sparse matrix you should avoid creating very large full matrices, even temporarily. One alternative is to give `sparse` the raw data required by the format.

```
sparse(1:4, 8:-2:2, [2 3 5 7])
ans =
    (4,2)      7
    (3,4)      5
    (2,6)      3
    (1,8)      2
```

Alternatively, you can create an empty sparse matrix with space to hold a specified number of nonzeros, and then fill in using standard subscript assignments. Another useful sparse building command is `spdiags`, which builds along the diagonals of the matrix.

```
>> M = ones(6,1)*[-20 Inf 10]
M =
    -20    Inf    10
    -20    Inf    10
>> S = spdiags(M, [-2 0 1], 6, 6);
>> full(S)
ans =
    Inf    10    0    0    0    0
    0    Inf    10    0    0    0
   -20    0    Inf    10    0    0
    0   -20    0    Inf    10    0
    0    0   -20    0    Inf    10
    0    0    0   -20    0    Inf
```

The `nnz` function tells how many nonzeros are in a given sparse matrix. Since it is impractical to view directly all the entries (even just the nonzeros) of a realistically sized sparse matrix, the `spy` command helps by producing a plot in which the locations of nonzeros are shown. For instance, `spy(bucky)` shows the pattern of bonds among the 60 carbon atoms in a buckyball.

MATLAB has a lot of ability to work intelligently with sparse matrices. The arithmetic operators `+`, `-`, `*`, and `^` use sparse-aware algorithms and produce sparse results when applied to sparse inputs. The backslash `\` uses sparse-appropriate linear system algorithms automatically as well. There are also functions for the iterative solution of linear equations, eigenvalues, and singular values.

## 1.4 MATLAB Programming

### 1.4.1 Control flow

MATLAB has four control flow structures: the `if` statement, the `for` loop, the `while` loop and the `switch` statement.

Here is an example illustrating most of the features of `if`.

```
if isnan(x) | ~isreal(x)
    disp('Bad input!')
    y = NaN;
elseif (x == round(x)) && (x > 0)
    y = prod(1:x-1);
else
    y = gamma(x);
end
```

Compound logic in `if` statements can be short-circuited. As a condition is evaluated from left to right, it may become obvious before the `end` that truth or falsity is assured. At that point, evaluation of the condition is halted. This makes it convenient to write things like

```
if (length(x) > 2) & (x(3)==1) ...
```

that otherwise could create errors or be awkward to write. If you want short-circuit behavior for logical operations outside `if` and `while` statements, you must use the special operators `||` and `&&`. The `if/elseif` construct is fine when only a few options are present. When a large number of options are possible, it is customary to use `switch` instead. For instance:

```
switch units
    case 'length'
        disp('meters')
    case 'volume'
        disp('liters')
    case 'time'
        disp('seconds')
    otherwise
        disp('I give up')
end
```

The `switch` expression can be a string or a number. The first matching `case` has its commands executed. Execution does not “fall through” as in C. If `otherwise` is present, it gives a default option if no case matches.

The `for` loop is one of the most useful MATLAB constructs although, code is more compact without it. The syntax is

```
for variable = expression
    statements
end
```

Usually, `expression` is a vector of the form `i : s : j`. The statements are executed with `variable` equal to each element of the `expression` in turn. For example, the first 10 terms of Fibonacci sequence are computed by

```
f = [1 1];
for n = 3:10
    f(n) = f(n-1) + f(n-2);
end
```

Another way to define *expression* is using the square bracket notation:

```
for x = [pi/6 pi/4 pi/3], disp([x, sin(x)]), end
0.5236    0.5000
0.7854    0.7071
1.0472    0.8660
```

Multiple `for` loops can of course be nested, in which case indentation helps to improve the readability. MATLAB editor debugger can perform automatic indentation. The following code forms the 5-by-5 symmetric matrix `A` with  $(i, j)$  element  $i/j$  for  $j \geq i$ :

```
n = 5; A = eye(n);
for j=2:n
    for i = 1:j-1
        A(i,j)=i/j;
        A(j,i)=i/j;
    end
end
```

The expression in the `for` loop can be a matrix, in which case *variable* is assigned the columns of *expression* from first to last. For example, to set `x` to each of the unit vector in turn, we can write `for x=eye(n), ..., end`.

The `while` loop has the form

```
while expression
    statements
end
```

The *statements* are executed as long as *expression* is true. The following example approximates the smallest nonzero floating point number:

```
x = 1; while x>0, xmin = x; x = x/2; end, xmin
xmin =
4.9407e-324
```

A `while` loop can be terminated with a `break` statement, which passes control to the first statement after the corresponding `end`. An infinite loop can be constructed using `while 1, ..., end`, which is useful when it is not convenient to put the exit test at the top of the loop. (Note that, unlike some other languages, MATLAB does not have a “repeat-until” loop.) We can rewrite the previous example less concisely as

```
x = 1; while 1
    xmin = x;
    x = x/2;
    if x == 0, break, end
end
xmin
```

The `break` statement can also be used to exit a `for` loop. In a nested loop a `break` exits to the loop at the next higher level.

The `continue` statement causes the execution of a `for` or a `while` loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop. As a trivial example

```
for i=1:10
    if i < 5, continue, end
    disp(i)
end
```

displays the integers 5 to 10. In more complicated loops the `continue` statement can be useful to avoid long-bodied `if` statements.

## 1.5 M files

### 1.5.1 Scripts and functions

MATLAB M-files are equivalents of programs, functions, subroutines and procedures in other programming languages. Collecting together a sequence of commands into an M-file opens up many possibilities including

- experimenting with an algorithm by editing a file, rather than retyping a long list of commands,
- making a permanent record of a numerical experiment,
- building up utilities that can be reused at a later date,
- exchanging M-files with colleagues.

An M-file is a text file that has a `.m` filename extension and contains MATLAB commands. There are two types:

**Script M-files** (or command files) have no input or output arguments and operate on variables in the workspace.

**Function M-files** contain a function definition line and can accept input arguments and return output arguments, and their internal variables are local to the function (unless declared `global`).

A script enables you to store a sequence of commands that are used repeatedly or will be needed at some future time.

The script given in MATLAB Source 1.1 uses random numbers to simulate a game. Consider 13 spade playing cards which are well shuffled. The probability to choose any card from the deck is  $1/13$ . The action of extracting a card is implemented by generating a random number. The game proceeds by putting the card back to the deck and reshuffling until the user press a key different to `x` or the number of repetitions is 20.

The line

```
rand('twister',sum(100*clock));
```

**MATLAB Source 1.1** The first MATLAB script - playingcards

---

```
%PLAYINGCARDS
%Simulating a card game

rand('Twister',sum(100*clock));
for k=1:20
    n=ceil(13*rand);
    fprintf('Selected card: %d\n',n)
    disp(' ')
    disp('Press r and Return to continue')
    r=input('or any other key to finish: ','s');
    if r=='r', break, end
end
```

---

resets the generator to a different state each time.

The first two lines of this script begin with the % symbol and hence are the comment lines. Whenever MATLAB encounters a % it ignores the remainder of the line. This allows you to insert text that makes the code easier for humans to understand. Starting with version 7 MATLAB allows block comments, that is, comments with more than one line length. To comment a contiguous group of lines, type %{ before the first line and %} after the last line you want to comment. Example:

```
%{
Block comment
two lines
%}
```

If our script is stored into the file `playingcards.m`, typing `playingcards` we obtain:

```
>> playingcards
Selected card: 6

Press r and Return to continue
or any other key to finish: r
Selected card: 9

Press r and Return to continue
or any other key to finish: x
>>
```

The first two lines serves as documentation for the file and will be typed out in the command window if `help` is used on the file.

```
>> help playingcards
PLAYINGCARDS
Simulating a card game
```

Function M-files enable you to extend the MATLAB language by writing your own functions that accept and returns arguments. They can be used in exactly the same way as existing MATLAB functions as `sin`, `eye`, `size`, etc.

---

### **MATLAB Source 1.2 The stat function**

---

```
function [med, sd] = stat(x)
%STAT Mean and standard deviation of a sample
%      [MEAN, SD] = STAT(X) computes mean and standard
%      deviation of sample X

n = length(x);
mean = sum(x)/n;
sd = sqrt(sum((x-mean).^2)/n);
```

---

MATLAB Source 1.2 shows a simple function that computes the mean and standard deviation of a sample (vector). This example illustrates a number of features. The first line begins with the keyword `function` followed by the list of output arguments, `[mean, sd]`, and the `=` symbol. On the right of the `=` comes the function name, `stat`, followed by the input arguments, `x`, within parentheses. (In general, there can be any number of input and output arguments.) The function name must be the same as the name of the `.m` file in which the function is stored – in this case the file must be named `stat.m`.

The second line of a function file is called the H1 (help 1) line. It should be a comment line of a special form: a line beginning with a `%` character, followed without any space by the function name in capital letter, followed by one or more space and then a brief description. The description should begin with a capital letter, end with a period, and omit the words “the” and “a”. All the comment lines from the first comment line up to the first noncomment line (usually a blank line, for readability of the source code) are displayed when `help function_name` is typed. Therefore these lines should describe the function and its arguments. It is conventional to capitalize function and argument names in these comment lines. When we type `help function_name`, all lines, from the first comment line to the first noncomment line, are displayed on screen. For `stat.m` example we have

```
>> help stat
STAT Mean and standard deviation of a sample
      [MEAN, SD] = STAT(X) computes mean and standard
      deviation of sample X
```

We strongly recommend documenting *all* your function files in this way, however short they may be. It is often useful to record in comment lines the date when the function was first written and to note all subsequent changes that have been made. The `help` command works in a similar manner on script files, displaying the initial sequence of comment lines.

The function `stat` is called just like any other MATLAB function:

```
>> [m, s]=stat(1:10)
m =
    5.5000
a =
    2.8723
```

```
>> x=rand(1,10);
[m,s]=stat(x)
m =
    0.5025
a =
    0.1466
```

A more complicated example is `mysqrt`, shown in MATLAB Source 1.3. Given  $a > 0$ , it computes  $\sqrt{a}$  by Newton's method

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{a}{x_k} \right), \quad x_1 = a.$$

Here are some examples of usage:

```
>> [x,it] = mysqrt(2)
x =
    1.414213562373095
it =
    6
>> [x,it]=mysqrt(2,1e-4)
x =
    1.414213562374690
it =
    4
```

This M-file illustrates the use of optional input arguments. The function `nargin` returns the number of input arguments supplied when the function was called and enables default values to be assigned to arguments that have not been specified. In this case, if the call to `sqrtn` does not specify a value for `tol`, then `eps` is assigned to `tol`. Analogously, `nargout` returns the number of the output arguments requested.

### 1.5.2 Subfunctions, nested and anonymous functions

A single M-file may hold more than one function definition. The first function is called primary function. Two other types of functions can be in the same file: *subfunctions* and *nested functions*.

The subfunctions begin after the end of primary function. Every subfunction in the file is available to be called by the primary function and the other subfunctions, but they have private workspaces and otherwise behave like functions in separate files. The difference is that only the primary function, not the subfunctions, can be called from sources outside the file. The next example computes the roots of a quadratic equation. It can solve several equations simultaneously, if the coefficients are given as vectors.

```
function x = quadeq(a,b,c)
% QUADEQ Find roots of a quadratic equation.
%     X = QUADRATIC(A,B,C) returns the two roots of
%     y = A*x^2 + B*x + C.
%     The roots are contained in X = [X1 X2].
```

---

**MATLAB Source 1.3 Function mysqrt**

---

```

function [x,iter] = mysqrt(a,tol)
%MYSQRT    Square root by Newton's method
%           X = SQRTN(A,TOL) computes the square root of
%           A by Newton's (Heron's) method
%           assume A >= 0.
%           TOL is the tolerance (default EPS).
%           [X,ITER] = SQRTN(A,TOL) returns also the number
%           of iterations ITER required

if nargin < 2, tol = eps; end

x = a;

for k=1:50
    xold = x;
    x = (x + a/x)/2;
    if abs(x-xold) <= tol*abs(x)
        iter=k; return;
    end
end
error('Not converged after 50 iterations')

```

---

```

denom = 2*a(:);
delta = descr(a,b,c); % Root of the discriminant
x1 = (-b + delta)./denom;
x2 = (-b - delta)./denom;
x = [x1(:), x2(:)];
end %quadeq
function d = descr(a,b,c)
d = sqrt(b().^2-4*a(:).*c(:));
end %descr

```

The end line is optional for single-function files, but it is a good idea when subfunctions are involved and mandatory when using nested functions.

For further examples on subfunctions, see MATLAB Sources 5.2 and 5.6 in Chapter 5.

Nested functions are defined within the scope of another function, and they share access to the containing functions workspace. For example, we can recast our quadratic formula yet again

```

function x = quadeqn(a,b,c)
    function descr(a,b,c)
        d = sqrt(b().^2-4*a(:).*c(:));
    end %descr()
denom = 2*a(:);
descr(a,b,c); % Root of the discriminant

```

---

```
x1 = (-b + d) ./denom;
x2 = (-b - d) ./denom;
x = [x1(:), x2(:)];
end %quadecn()
```

Sometimes you may need a quick, short function definition that does not seem to merit a named file on disk, or even a named subfunction. The “old-fashioned” way to do this is by using an `inline` function. Example:

```
f = inline('x*sin(x)', 'x')
```

Starting with MATLAB 7, a better alternative became available: the *anonymous function*. A simple example of an anonymous function is

```
sincos = @(x) sin(x) + cos(x);
```

We can define multivariate functions:

```
w = @(x,y,z) cos(x-y*z);
```

More interestingly, anonymous functions can define functions depending on parameters available at the time of anonymous function’s creation. As an illustration, consider the function given by  $f(x) = x^2 + a$ , where  $a$  is a given parameter. The definition could be:

```
a = 2;
f = @(x) x.^2+a
```

Nested functions and anonymous functions have similarities but are not quite identical. Nested functions always share scope with their enclosing parent; their behavior can be changed by and create changes within that scope. An anonymous function can be influenced by variables that were in scope at its creation time, but thereafter it acts autonomously.

### 1.5.3 Passing a Function as an Argument

Many problems in scientific computation, like finding roots, approximating integrals (quadrature), and solving differential equations need to pass a function as an argument to another function. MATLAB calls the functions operating on other functions `function functions`. (See the help topic `funfun` for a complete list.) There are several ways to pass functions as parameters but the most important is the `function handle`. A `function handle` is a MATLAB datatype that contains all the information necessary to evaluate a function. A `function handle` can be created by putting the `@` character before the function name. Example:

```
ezplot(@fun)
```

The parameter `fun` can be the name of an M-file, or a built-in function.

```
ezplot(@sin)
```

The definitions of anonymous functions and `inline` objects provide `function handles` as result, so they do not require a `@` character before their name.

Another way is to transmit the parameter as a string:

```
ezplot('exp')
```

but a `function handle` is more efficient and versatile.

Consider the function `fd_deriv` in MATLAB Source 1.4. This function evaluates the finite difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

---

**MATLAB Source 1.4 Function fd\_deriv**

---

```
function y = fd_deriv(f,x,h)
%FD_DERIV    Aproximates derivative by divided difference
%              FD_DERIV(F,X,H) is the divided difference of F
%              with nodes X and X+H.
%              default H:  SQRT(EPS).

if nargin < 3, h = sqrt(eps); end
f=fcnchk(f);
y = (f(x+h) - f(x))/h;
```

---

to the function passed as its first argument. Our first example uses a built-in function

```
>> fd_deriv(@sqrt,0.1)
ans =
    1.5811
```

We can use `mysqrt` function (source MATLAB 1.3) instead of predefined `sqrt`:

```
fd_deriv(@mysqrt,0.1)
ans =
    1.5811
```

In this example `fd_deriv` uses the default tolerance, since it accepts a univariate function as argument. If we want to force the use of another tolerance, use an anonymous function as argument

```
>> format long
>> fd_deriv(@(x) mysqrt(x, 1e-3), 0.1)
ans =
    1.581138722598553
```

We can pass an inline object or an anonymous function to `fd_deriv`:

```
>> fd_deriv(f,pi)
ans =
    -0.0063
>> g = @(x) x^2*sin(x);
>> fd_deriv(g,pi)
ans =
    -9.8696
```

The role of line

```
f=fcnchk(f);
```

in `fd_deriv` is to make this function to accept a string expression as input parameter. (This is how `ezplot` and other MATLAB function work with string expression, see [66] for examples). The command `fcnchk(f, 'vectorized')` vectorizes the string `f`. The function `vectorize` vectorizes an inline object.

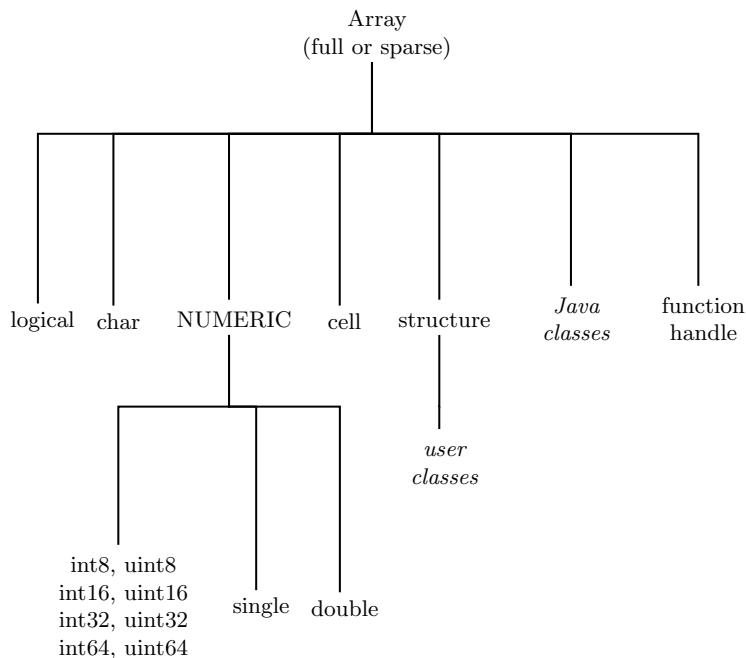


Figure 1.2: MATLAB class hierarchy

### 1.5.4 Advanced data structures

#### Other data types

There are 15 fundamental data types in MATLAB. Each of these data types is in the form of a matrix or array. This matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size. All of the fundamental data types are shown in lowercase, plain nonitalic text in Figure 1.2. The two data types shown in italic text are user-defined, object-oriented user classes and Java classes.

The default data type is `double`. There exists also a `single` data type corresponding to IEEE single precision arithmetic. The reasons for working with single precision are: to save storage (a scalar single requires only 32 bits to be represented) and to explore the accuracy of numerical algorithms.

The `single` type will be described in Section 3.6.

The types `int*` and `uint*`, where \* is replaced by 8, 16, 32 or 64 are array of signed (`int`) and unsigned (`uint`) integers. Some integer types require less storage space than `single` or `double`. All integer types except for `int64` and `uint64` can be used in mathematical operations. We can construct an object of previously mentioned types by typing its type name as a function with its value as argument. The command `c = class(obj)` returns the class of the object `obj`. Examples:

```

>> s=single(pi)
s =
    3.1416
>> x=pi
x =
    3.1416
>> format long, x=pi, format short
x =
    3.141592653589793
>> a=uint8(3)
a =
    3
>> b=int32(2^20+1)
b =
    1048577
>> whos
  Name      Size            Bytes  Class       Attributes
  a         1x1              1  uint8
  b         1x1              4  int32
  s         1x1              4  single
  x         1x1              8  double
>> class(s)
ans =
single

```

For details on nondouble types see [60, 61].

### String and formatted output

A string in MATLAB is enclosed in single forward quotes. In fact a string is really just a row vector of character ASCII codes. Because of this, strings can be concatenated using matrix concatenation.

```

>> str = 'Hello world';
>> str(1:5)
ans =
Hello
>> s1 = double(str)
s1 =
    72    101    108    108    111     32    119    111    114    108    100
>> s2 = char(s1)
s2 =
Hello world
>> ['Hello', ' ', 'world']
ans =
Hello world
>> whos
  Name      Size            Bytes  Class       Attributes
  
```

```

ans      1x11          22   char
s1       1x11          88   double
s2       1x11          22   char
str      1x11          22   char

```

You can convert a string such as 3.14 into its numerical meaning (not its character codes) by using `eval` or `str2num` on it. Conversely, you can convert a number to string representation using `num2str` or the much more powerful `sprintf` (see below). If you want a quote character within a string, use two quotes, as in `It''s Donald''s birthday`. Multiple strings can be stored as rows in an array using `str2mat`, which pads strings with blanks so that they all have the same length. However, a better way to collect strings is to use cell arrays. There are lots of string handling functions. See the help on `strfun`. Here are a few:

```

>> strcat('Hello',' world')
ans =
Hello world
>> upper(str)
ans =
HELLO WORLD
>> strcmp(str,'Hello world')
ans =
1
>> findstr('world',str)
ans =
7

```

For the conversion of internal data to string, in memory, or for output, use `sprintf` or `fprintf`. These are closely based on the C function `printf`, with the important vectorization enhancement that format specifiers are reused through all the elements of a vector or matrix (in the usual rowfirst order).

```

x=0.25; n=1:6;
c=1./cumprod([1 n]);
for k=1:7, T(k)=polyval(c(k:-1:1),x); end
X=[(0:6)',T', abs(T-exp(x))'];
fprintf('\n n |      T_n(x)      | |T_n(x)-exp(x)|\n');
fprintf('-----\n');
fprintf(' %d | %15.12f | %8.3e\n', x' )
fprintf('-----\n');

n |      T_n(x)      | |T_n(x)-exp(x)|
-----
0 | 1.000000000000 | 2.840e-001
1 | 1.250000000000 | 3.403e-002
2 | 1.281250000000 | 2.775e-003
3 | 1.283854166667 | 1.713e-004
4 | 1.284016927083 | 8.490e-006
5 | 1.284025065104 | 3.516e-007

```

```
6 | 1.284025404188 | 1.250e-008
-----
```

Use `sprintf` if you want to save the result as a string rather than have it output immediately.

## Cell arrays

Cell arrays are used to gather dissimilar objects into one variable. They are like regular numeric arrays, but their elements can be absolutely anything. A cell array is created or using curly braces rather than parentheses or `cell`. Cell array contents are indexed using curly braces, and the colon notation can be used in the same way as for other arrays.

```
>> vvv={'Humpty' 'Dumpty' 'sat' 'on' 'a' 'wall'}
vvv =
    'Humpty'    'Dumpty'    'sat'    'on'    'a'    'wall'
>> vvv{3}
ans =
sat
```

The next example tabulates the coefficients of Chebyshev polynomials. In MATLAB one expresses a polynomial as a vector (highest degree first) of its coefficients. The number of coefficients needed grows with the degree of the polynomial. Although you can put all the Chebyshev coefficients into a triangular array, this is an inconvenient complication. The recurrence relation is

$$\begin{aligned} T_0(x) &= 1; \quad T_1(x) = x; \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x), \quad n = 1, 2, \dots \end{aligned}$$

Here is the code

```
function T=ChebyshevLC(deg)
%CHEBYSHEVLC - Chebyshev polynomials
%   tabulate coefficients of Chebyshev polynomials
%   T = CHEBYSHEVLC(DEG)
T = cell(1,deg);
T(1:2) = { [1], [1 0] };
for n = 2:deg
    T{n+1} = [2*T{n} 0] - [0 0 T{n-1}];
end
```

and a call

```
>> T=ChebyshevLC(6)
T =
    [1]      [1x2 double]      [1x3 double]      [1x4 double]
                  [1x5 double]      [1x6 double]      [1x7 double]
>> for k=1:5, disp(T{k}), end
    1
    1      0
    2      0      -1
    4      0      -3      0
    8      0      -8      0      1
```

Cell arrays are used by the `varargin` and `varargout` functions to work with a variable number of arguments.

## Structures

Structures are much like cell arrays, but they are indexed by names rather than by numbers. Suppose we want to build a structure with probability distributions, containing the name, the type and the probability density (or mass) function.

```
>> pdist(1).name='binomial';
>> pdist(1).type='discrete';
>> pdist(1).pdf=@binopdf;
>> pdist(2).name='normal';
>> pdist(2).type='continuous';
>> pdist(2).pdf=@normpdf;
```

Displaying the structure gives the field names but not the contents:

```
>> pdist
pdist =
1x2 struct array with fields:
  name
  type
  pdf
```

We can access individual fields using a period:

```
>> pdist(2).name
ans =
normal
```

Another way to set up the `pdist` structure is using the `struct` command:

```
>> pdist = struct('name',{'binomial','normal'},...
' type',{'discrete','continuous'}, ...
' pdf',{@binopdf,@normpdf})
```

The arguments to the `struct` function are the field names, with each field name followed by the field contents listed within curly braces (that is, the field contents are cell arrays, which are described next). If the entire structure cannot be assigned with one `struct` statement then it can be created with fields initialized to a particular value using `repmat`. For example, we can set up a `pdist` structure for six distributions with empty fields

```
>> pdist = repmat(struct('name','',''), ' type','','', ...
' pdf',''),6,1)
pdist =
6x1 struct array with fields:
  name
  type
  pdf
```

Struct arrays make it easy to extract data into cells:

---

**MATLAB Source 1.5 Functioncompanb**

---

```

function C = companb(varargin)
%COMPANB    Block companion matrix.
%
%          C = COMPANB(A_1,A_2,...,A_m) is the block companion
%          matrix corresponding to the n-by-n matrices
%
%          A_1,A_2,...,A_m.

m = nargin;
n = length(varargin{1});

C = diag(ones(n*(m-1),1),-n);
for j = 1:m
    Aj = varargin{j};
    C(1:n, (j-1)*n+1:j*n) = -Aj;
end

>> [rlp{1:2}] = deal(pdist.name)
rlp =
    'binomial'    'normal'

```

---

In fact, `deal` is a very handy function for converting between cells and structs. See online help for many examples.

Structures are used by `spline`, by `solve`, and to set options for the nonlinear equation and optimization solvers and the differential equation solvers. Structures also play an important role in object-oriented programming in MATLAB (which is not discussed here).

### 1.5.5 Variable Number of Arguments

In certain situations a function must accept or returns a variable, possibly unlimited, number of input or output arguments. This can be achieved using the `varargin` and `varargout` functions. Suppose we wish to write a function `companb` to form the  $mn \times mn$  block companion matrix corresponding to the  $n \times n$  matrices  $A_1, A_2, \dots, A_m$ :

$$C = \begin{bmatrix} -A_1 & -A_2 & \dots & \dots & -A_m \\ I & 0 & & & 0 \\ & I & \ddots & & \vdots \\ & & \ddots & & \vdots \\ & & & I & 0 \end{bmatrix}.$$

The solution is to use `varargin` as shown in MATLAB Source 1.5. (This example is given in [44].) When `varargin` is specified as the input argument list, the input arguments supplied are copied into a cell array called `varargin`. Consider the call

```
>> X = ones(2); C = companb(X, 2*X, 3*X)
C =
```

```

-1    -1    -2    -2    -3    -3
-1    -1    -2    -2    -3    -3
1     0     0     0     0     0
0     1     0     0     0     0
0     0     1     0     0     0
0     0     0     1     0     0

```

If we insert the line

```
varargin
```

at the beginning of `companb` then the above call produces

```
varargin =
[2x2 double] [2x2 double] [2x2 double]
```

Thus `varargin` is a 1-by-3 cell array whose elements are 2-by-2 matrices passed as arguments to `companb`, and `varargin{j}` is the  $j$ -th input matrix,  $A_j$ .

It is not necessary for `varargin` to be the only input argument but it must be the last one, appearing after any named input arguments.

An example using the analogous statement `varargout` for output arguments is shown in MATLAB Source 1.6. Here we use `nargout` to determine how many output arguments have been requested and then create a `varargout` cell array containing the required output. To illustrate:

```

>> m1 = moments(1:4)
m1 =
2.5000

>> [m1,m2,m3] = moments(1:4)
m1 =
2.5000
m2 =
7.5000
m3 =
25

```

---

#### **MATLAB Source 1.6 Function `moments`**

---

```

function varargout = moments(x)
%MOMENTS Moments of a vector.
%           [m1,m2,...,m_k] = MOMENTS(X) returns the first, second, ...
%           k'th moments of the vector X, where the j'th moment
%           is SUM(X.^j)/LENGTH(X).

for j=1:nargout, varargout(j) = sum(x.^j)/length(x); end

```

---

### **1.5.6 Global variables**

Variables within a function are local to that function workspace. Occasionally it is convenient to create variables that exist in more than workspace including, possibly, the main workspace.

This can be done using the `global`. statement.

As example, we give the code for `tic` and `toc` MATLAB functions (with shorter comments). These functions measure the elapsed time. The global variable `TICTOC` is visible for both function, but is invisible at base workspace (command line or script level) or in any other function that does not declare it with `global`.

```
function tic
%    TIC Start a stopwatch timer.
%    TIC; any stuff; TOC
%    prints the time required.
%    See also: TOC, CLOCK.
global TICTOC
TICTOC = clock;
function t = toc
%    TOC Read the stopwatch timer.
%    TOC prints the elapsed time since TIC was used.
%    t = TOC; saves elapsed time in t, does not print.
%    See also: TIC, ETIME.
global TICTOC
if nargout < 1
    elapsed_time = etime(clock,TICTOC)
else
    t = etime(clock,TICTOC);
end
```

Within a function, the `global` declaration should appear before the first occurrence of the relevant variable, ideally at the top of the file. By convention, the names of global variables are comprised of capital letters, and ideally the names are long in order to reduce the chance of clashes with other variables.

### 1.5.7 Recursive functions

Functions can be recursive, that is, they can call themselves. Recursion is a powerful tool, but not all computations that are described recursively are best programmed this way.

Function `mygcd` in MATLAB Source refolis10.5 use recursion to compute the greatest common divisor, that is the property

$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b).$$

It accepts two numbers as input arguments and returns their greatest common divisor. Example:

```
>> d=mygcd(5376, 98784)
d =
   672
```

For other examples on recursion see `quad` and `quadl` MATLAB functions and sources in Sections 6.4 and 6.6, used for numerical integration.

---

**MATLAB Source 1.7 Recursive gcd**

---

```

function d=mygcd(a,b)
%MYGCD - recursive greatest common divisor
%   D = MYGCD(A,B) computes the greatest
%       common divisor of A and B

if a==0 & b==0, then d = NaN;
elseif b==0
    d = a;
else
    d = mygcd(b, mod(a,b));
end

```

---

### 1.5.8 Error control

MATLAB functions may encounter statements that are impossible to execute (for example, multiplication of incompatible matrices). In that case an error is thrown: execution of the function halts, a message is displayed, and the output arguments of the function are ignored. You can throw errors in your own functions with the `error` statement, called with a string that is displayed as the message. Similar to an error is a warning, which displays a message but allows execution to continue. You can create these using `warning`.

Sometimes you would like the ability to recover from an error in a subroutine and continue with a contingency plan. This can be done using the `try-catch` construct. For example, the following will continue asking for a statement until you give it one that executes successfully.

```

done = false;
while ~done
    state = input('Enter a valid statement: ','s');
    try
        eval(state);
        done = true;
    catch
        err=lasterror;
        disp('That was not a valid statement! Look:')
        disp(err.identifier)
        disp(err.message)
    end
end

```

Within the catch block you can find the most recent error message using `lasterr`, or detailed information using `lasterror`. Also, `error` and `warning` allow a more sophisticated form of call, with formatted string and message identifiers.

For details, see [61] and the corresponding helps.

## 1.6 Symbolic Math Toolboxes

Symbolic Math Toolboxes incorporate symbolic computer facilities into MATLAB numeric environment. The toolbox is based upon the Maple® kernel, which performs all the symbolic and variable precision computations. Starting from Maple 2008b there exist a MuPAD based Symbolic Toolbox. You can select among Maple and MuPAD as support engine. There are two toolboxes:

- Symbolic Math Toolbox, that provides access to Maple kernel and to Maple linear algebra package using a style and a syntax which are natural extensions of MATLAB language.
- Extended Symbolic Math Toolbox extends the above mentioned features to provide access to nongraphical Maple packages, and to Maple programming capabilities and user's defined procedures.

The Symbolic Math Toolbox defines a new datatype: a symbolic object, denoted by `sym`. Internally, a symbolic object is a data structure that stores a string representation of the symbol. Symbolic Math Toolbox uses symbolic object to represent symbolic variables, expressions and matrices. Its default arithmetic for symbolic object is the rational arithmetic.

Symbolic objects can be created with the `sym` function. For example, the statement

```
x = sym('x');
```

produces a new symbolic variable, `x`. We can combine several such declarations using `syms`:

```
syms a b c x y f g
```

Examples in this section assume that we have already executed this command.

**Differentiation.** We can differentiate a symbolic expression with `diff`. Let us create a symbolic expression:

```
>> f=exp(a*x)*sin(x);
```

Its derivative with respect to `x` can be computed using

```
>> diff_f=diff(f,x)
diff_f =
a*exp(a*x)*sin(x)+exp(a*x)*cos(x)
```

If `n` is a nonzero natural number, `diff(f, x, n)` computes the `n`th order derivative of `f`. The next example computes the second derivative

```
>> diff(f,x,2)
ans = a^2*exp(a*x)*sin(x)+2*a*exp(a*x)*cos(x)-exp(a*x)*sin(x)
```

For details see `help sym/diff` or `doc sym/diff`.

**Integration.** To calculate the indefinite integral of a symbolic expression `f` we can use `int(f, x)`. For example, to find the indefinite integral of  $g(x) = e^{-ax} \sin cx$ , we do:

```
>> g = exp(-a*x)*sin(c*x);
>> int_g=int(g,x)
int_g =
-c/(a^2+c^2)*exp(-a*x)*cos(c*x)-a/(a^2+c^2)*exp(-a*x)*sin(c*x)
```

If we execute the command `diff(int_g, x)`, we do not obtain  $g$ , but rather an equivalent expression. After the execution of the command `simple(diff(int_g, x))`, we obtain a sequence of messages that informs about the used rules, and finally

```
ans = exp(-a*x)*sin(c*x)
```

To compute the definite integral  $\int_{-\pi}^{\pi} g \, dx$ , we may try `int(g, x, -pi, pi)`. The result is not too elegant. Another example is the computation of  $\int_0^{\pi} x \sin x \, dx$ :

```
>> int('x*sin(x)', x, 0, pi)
ans = pi
```

When MATLAB is unable to find an analytical (symbolic) integral, such as with

```
>> int(exp(sin(x)), x, 0, 1)
Warning: Explicit integral could not be found.
ans =
int(exp(sin(x)), x = 0 .. 1)
```

we can try to find a numerical approximation, like:

```
>> quad('exp(sin(x))', 0, 1)
ans =
1.6319
```

Functions `int` and `diff` can both be applied to matrices, in which case they operate elementwise.

For details see `help sym/int` or `doc sym/int`.

**Substitutions and simplifications.** The substitution of one value or parameter for another in an expression is done with `subs`. (See `help sym/subs` or `doc sym/subs`).

For example, let us compute the previous definite integral of  $g$  for  $a=2$  and  $c=4$ :

```
>> int_sub=subs(int_def_g, {a, c}, {2, 4})
int_sub =
107.0980
```

`simplify` is a powerful function that applies various type of identities and simplification rules to bring an expression to a “simpler” form. Example:

```
>> syms h
>> h=(1-x^2)/(1-x);
>> simplify(h)
ans = x+1
```

`simple` is a “non-orthodox” function whose aim is to obtain an equivalent expression with the smallest number of characters. We have already given an example, but we consider another one:

```
>> [jj, how]=simple (cos(x)^2+sin(x)^2)
jj =
1
how =
simplify
```

The rôle of the second output parameter is to avoid long messages during the simplification process.

See `help sym/simplify` and `help sym/simple` or `doc sym/simplify` and `doc sym/simple`.

**Taylor series.** The `taylor` command is useful to generate symbolic Taylor expansions about a given symbolic value of the argument. For example, to compute the 5th order expansion of  $e^x$  about  $x = 0$ , we may use:

```
>> clear, syms x, Tay_expx=taylor(exp(x),5,x,0)
Tay_expx =
1+x+1/2*x^2+1/6*x^3+1/24*x^4
```

The `pretty` command, display a `sym` object in a 2D form, close to the mathematical format:

```
>> pretty(Tay_expx)
```

$$1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4$$

Now, we shall compare the approximation for  $x = 2$  to the exact value:

```
>> approx=subs(Tay_expx,x,2), exact=exp(2)
approx =
7
exact =
7.3891
>> frac_err=abs(1-approx/exact)
frac_err =
0.0527
```

We can compare the two approximations graphically with `ezplot` (see Chapter 2 for MATLAB graphics capabilities):

```
>> ezplot(Tay_expx,[-3,3]), hold on
>> ezplot(exp(x),[-3,3]), hold off
>> legend('Taylor','exp','Location','Best')
```

The graph is given in Figure 1.3. `ezplot` function is a convenient way to plot symbolic expressions.

For additional information, see `help sym/taylor` or `doc sym/taylor`.

**Limits.** For the syntax and usage of `limit` command see `help sym/limit` or `doc sym/limit`. We give only two simple examples. The first computes  $\lim_{x \rightarrow 0} \frac{\sin x}{x}$ :

```
>> L=limit(sin(x)/x,x,0)
L =
1
```

The second example computes one sided limit of the `tan` function when  $x$  approaches to  $\frac{\pi}{2}$ .

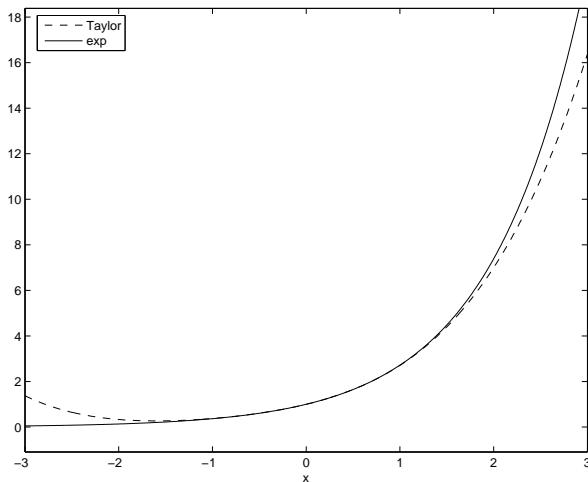


Figure 1.3: The exponential and its Taylor expansion

```
>> LS=limit(tan(x),x,pi/2,'left')
LS =
Inf
>> LD=limit(tan(x),x,pi/2,'right')
LD =
-Inf
```

**Solving equations.** Symbolic Math Toolbox is able to solve equations and linear and nonlinear systems of equations. See `help sym/solve` or `doc sym/solve`. We shall give a few examples. Before the solution we shall clear the memory and define the symbolic variables and equations. (It is a good practice to clear the memory before the solution of a new problem to avoid the side-effects of previous values of certain variables.)

We start with a generic quadratic equation,  $ax^2 + bx + c = 0$ .

```
>> clear, syms x a b c
>> eq='a*x^2+b*x+c=0';
>> x=solve(eq,x)
x =
1/2/a*(-b+(b^2-4*a*c)^(1/2))
1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

If we have several solution we can select one of them using indexing, for example `x(1)`.

Let us solve now the linear system  $2x - 3y + 4z = 5$ ,  $y + 4z + x = 10$ ,  $-2z + 3x + 4y = 0$ .

```
>> clear, syms x y z
>> eq1='2*x-3*y+4*z=5';
>> eq2='y+4*z+x=10';
>> eq3=' -2*z+3*x+4*y=0';
>> [x,y,z]=solve(eq1,eq2,eq3,x,y,z)
```

```
x =
-5/37
y =
45/37
z =
165/74
```

Notice that the order of input variables is not important, while the order of output variables is important.

The next example solves the nonlinear system

$$\begin{aligned}y &= 2e^x \\y &= 3 - x^2.\end{aligned}$$

```
>> clear, syms x y
>> eq1='y=2*exp(x)'; eq2='y=3-x^2';
>> [x,y]=solve(eq1,eq2,x,y)
x =
.36104234240225080888501262630700
y =
2.8696484269926958876157155521484
```

Now, consider the solution of the trigonometric equation  $\sin x = \frac{1}{2}$ . This has an infinite number of solutions. The sequence of commands

```
>> clear, syms x, eq='sin(x)=1/2';
>> x=solve(eq,x)
```

finds only the solution

```
x =
1/6*pi
```

To find the solutions within a given interval, say [2,3], we can use `fsolve` command:

```
>> clear, x=maple('fsolve(sin(x)=1/2,x,2..3)')
x =
2.6179938779914943653855361527329
```

The result is a character string, that can be converted into `double` with `str2double`:

```
>> z=str2double(x), whos
z =
2.6180
Name      Size          Bytes  Class
ans      1x33            264  double array
x       1x33             66   char array
y       1x1              8    double array
z       1x1              8    double array

Grand total is 68 elements using 346 bytes
```

Symbolic Math toolbox can solve differential equation with `dsolve` command. We give two examples. The first solve a scalar second order ODE:  $y'' = 2y + 1$ .

```
>> dsolve('D2y = 2*y+1')
ans =

$$\exp(2t) \_C2 + \exp(-2t) \_C1 - 1/2$$

```

The second solve the previous equation with initial condition  $y(0) = 1$ ,  $y'(0) = 0$ .

```
>> dsolve('D2y = 2*y+1', 'y(0)=1', 'Dy(0)=0')
ans =

$$\frac{3}{4} \exp(2t) + \frac{3}{4} \exp(-2t) - 1/2$$

```

The `maple` command pass command sequences to Maple kernel. See the appropriate help and the documentation.

**Variable precision arithmetic (vpa).** There are three different kinds of arithmetic operations in this toolbox:

- Numeric – MATLAB floating-point arithmetic.
- Rational – Maple exact symbolic arithmetic;
- VPA – Maple's variable-precision arithmetic.

Variable-precision arithmetic is done by calling the `vpa` function. The number of digits is controlled by `Digits` Maple variable. `digits` function display the `Digits` value, and `digits(n)`, where `n` is an integer sets `Digits` to `n` decimal digits. The function `vpa(a, d)` returns the expression `a` evaluated with `d` digits of floating-point precision. If `d` is not given, then `d` is equal to the default precision, `digits`. The result is of type `sym`.

For example, the MATLAB statements

```
>> clear
>> format long
1/2+1/3
```

use numeric computation to produce

```
ans =
0.833333333333333
```

With the Symbolic Math Toolbox, the statement

```
>> sym(1/2)+1/3
```

uses symbolic computation to yield

```
ans =
5/6
```

And, also with the toolbox, the statements

```
>> digits(25)
>> vpa('1/2+1/3')
use variable-precision arithmetic to return
```

```
ans =
.8333333333333333333333333333333
```

To convert a variable precision number to `double` one can use the `double` function.  
The next example compute the golden section

```
>> digits(32)
>> clear, phi1=vpa((1+sqrt(5))/2)
phi1 =
1.6180339887498949025257388711907
>> phi2=vpa('(1+sqrt(5))/2'), diff=phi1-phi2
phi2 =
1.6180339887498948482045868343656
diff =
.543211520368251e-16
```

The discrepancy between `phi1` and `phi2` is due to the fact that the first assignment perform his computation in double precision and convert the result to `vpa`, while the second use a string and perform all computation in `vpa`.

For additional information about Symbolic Math Toolbox, we recommend [62].

**Using MuPAD from MATLAB.** Version 5 of Symbolic Math Toolbox is powered by the MuPAD symbolic engine.

- Nearly all Symbolic Math Toolbox functions work the same way as in previous versions.
- MuPAD notebooks provide a new interface for performing symbolic and variable-precision calculations, plotting, and animations.
- Symbolic Math Toolbox functions allow you to copy variables and expressions between the MATLAB workspace and MuPAD notebooks.
- You can call MuPAD functions and procedures from the MATLAB environment.

The next example computes the mean and the variance of a random variable that obeys an exponential distribution with parameter  $b > 0$ . The probability density function (pdf) is

$$f(x; b) = \begin{cases} \frac{e^{-x/b}}{b}, & \text{if } x > 0; \\ 0, & \text{otherwise.} \end{cases}$$

To open a new MuPAD notebook, at MATLAB prompt type

```
mupad
```

The input area is demarcated by a left bracket (`[]`). To set parameters and to input the pdf, type

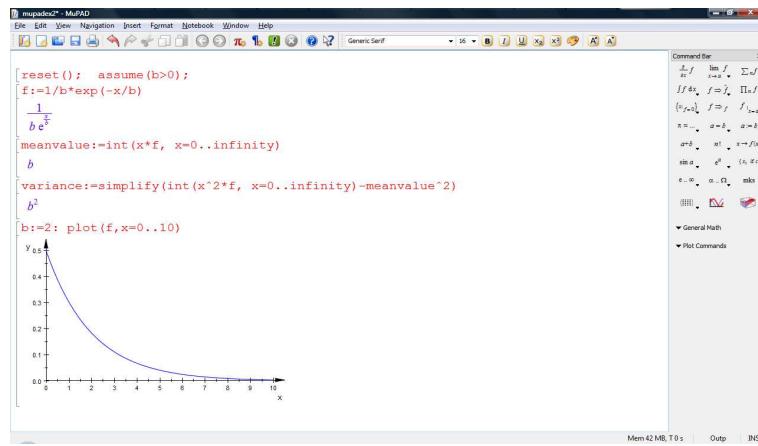


Figure 1.4: A MuPAD notebook

```
reset();
assume(b>0);
f:=1/b*exp(-x/b)
```

Now, the sequence for the computation of mean and variance

```
meanvalue:=int(x*f, x=0..infinity)
variance:=simplify(int(x^2*f, x=0..infinity)-meanvalue^2)
```

Finally, the commands for plotting the pdf for  $b=2$

```
b:=2: plot(f, x=0..10)
```

Figure 1.4 shows the MuPAD notebook for this example.

## Problems

**Problem 1.1.** Compute the sum

$$S_n = \sum_{k=1}^n \frac{1}{k^2},$$

efficiently, for  $n = \overline{20, 200}$ . How well approximates  $S_n$  the sum of series

$$S = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}?$$

**Problem 1.2.** Write a function M file to evaluate MacLaurin expansion of  $\ln(x + 1)$ :

$$\ln(x + 1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n+1} \frac{x^n}{n} + \dots$$

The convergence holds for  $x \in [-1, 1]$ . Test your function for  $x \in [-0.5, 0.5]$  and check what is happened when  $x$  approaches -1 or 1.

**Problem 1.3.** Write a MATLAB script that reads an integer and convert it into the roman numeral.

**Problem 1.4.** Implement an iterative variant of Euclid's algorithm for gcd in MATLAB.

**Problem 1.5.** Implement the binary search in an ordered array in MATLAB.

**Problem 1.6.** Write a MATLAB code that generates, for a given  $n$ , the tridiagonal matrix

$$B_n = \begin{bmatrix} 1 & n & & & & \\ -2 & 2 & n-1 & & & \\ & -3 & 3 & n-2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -n+1 & n-1 & 2 \\ & & & & -n & n \end{bmatrix}.$$

**Problem 1.7.** What is the largest value of  $n$  such that

$$S_n = \sum_{k=1}^n k^2 < L,$$

where  $L$  is given? Solve by summation and using a classical formula for  $S_n$ .

**Problem 1.8.** Generate the matrix  $H_n = (h_{ij})$ , where

$$h_{ij} = \frac{1}{i+j-1}, \quad i, j = \overline{1, n},$$

using Symbolic Math Toolbox.

**Problem 1.9.** Build the triangular matrix of binomial coefficients, for powers from 1 to a given  $n \in \mathbb{N}$ .

**Problem 1.10.** Develop a MATLAB function that accepts the coordinates of a triangle vertices and a level as input parameters and subdivides the triangle recursively into four triangles after the midpoints of edges. The subdivision proceed until the given level is reached.

**Problem 1.11.** Write a MATLAB function that extracts from a given matrix  $A$  a block diagonal part, where the size of each block is given, and the upper left corner of each block lays on the main diagonal.

**Problem 1.12.** One way to compute the exponential function `exp` is to use its Taylor series expansion around  $x = 0$ . Unfortunately, many terms are required if  $|x|$  is large. But a special property of the exponential is that  $e^{2x} = (e^x)^2$ . This leads to a scaling and squaring method: Divide  $x$  by 2 repeatedly until  $|x| < 1/2$ , use a Taylor series (16 terms should be more than enough), and square the result repeatedly. Write a function `expss(x)` that does this. (The function `polyval` can help with evaluating the Taylor expansion.) Test your function on  $x$  values -30, -3, 3, 30.

**Problem 1.13.** Let  $\mathbf{x}$  and  $\mathbf{y}$  be column vectors describing the vertices of a polygon (given in order). Write functions `polyperim(x, y)` and `polyarea(x, y)` that compute the perimeter and area of the polygon. For the area, use a formula based on Green's theorem:

$$A = \frac{1}{2} \left| \sum_{i=1}^n (x_k y_{k+1} - x_{k+1} y_k) \right|.$$

Here  $n$  is the number of vertices and it is understood that  $x_{n+1} = x_1$  and  $y_{n+1} = y_1$ . Test your function on a square and an equilateral triangle.

# CHAPTER 2

---

## MATLAB Graphics

---

MATLAB has powerful and versatile graphics capabilities. You can generate easily a large variety of highly customizable graphs and figures. We do not intend to be exhaustive, rather to introduce the reader to those capabilities of MATLAB which will be necessary in the sequel. The figures and graphs are not only easy to generate, but they can be easily modify and annotate interactively and with specialized functions (see help plotedit). For a deeper insight of MATLAB graphics we recommend [44, 56, 59, 68, 55].

### 2.1 Two-Dimensional Graphics

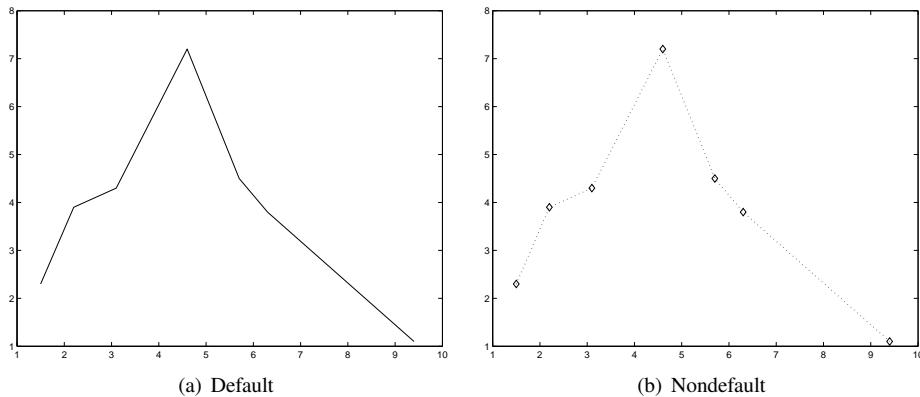
#### 2.1.1 Basic Plots

MATLAB's `plot` function can be used for simple “join-the-dots”  $x$ - $y$  plots. Typing

```
>>x=[1.5,2.2,3.1,4.6,5.7,6.3,9.4];  
>>y=[2.3,3.9,4.3,7.2,4.5,3.8,1.1];  
>>plot(x,y)
```

produces the left-hand picture in Figure 2.1(a), where the points  $x(i)$ ,  $y(i)$  are joined in sequence. MATLAB opens a figure window (unless one has already opened as a result of a previous command) in which to draw the picture. In this example, default values are used for a number of features, including the ranges of the  $x$ - and  $y$ -axes, the spacing of the axis tick marks, and the color and type of the line used for the plot.

More generally we could replace `plot(x,y)` with `plot(x,y,string)`, where *string* combines up to three elements that control the color, marker and line style. For example, `plot(x,y,'r*--')` specifies that a red asterisk is to be placed at each point  $x(i)$ ,  $y(i)$  and the points are to be joined by a red dashed line, whereas `plot(x,y,'+y')` specifies a yellow cross marker with no line joining the points. Table 2.1 lists the options available.

Figure 2.1: Simple  $x$ - $y$  plots

The right-hand picture in Figure 2.1 was produced with `plot(x, y, 'kd:')`, which gives a black dotted line with diamond marker. The three elements in `string` may appear in any order, so, for example, `plot(x, y, 'ms--')` and `plot(x, y, 's--m')` are equivalent. You can exert further control by supplying more arguments to `plot`. The properties called `LineWidth` (default 0.5 points) and `MarkerSize` (default 6 points) can be specified in points, where a point is 1/72 inch. For example, the commands

```
>>plot(x, y, 'm--^', 'LineWidth', 3, 'MarkerSize', 5)
```

produce a plot with a 2-point line width and 10-point marker size, respectively. For markers that have a well-defined interior, the `MarkerEdgeColor` and `MarkerFaceColor` can be set to one of the colors in Table 2.1. So, for example

```
plot(x, y, 'o', 'MarkerEdgeColor', 'm')
```

gives magenta edges to the circles. The left-hand plot in Figure 2.2 was produced with

```
plot(x, y, 'm--^', 'LineWidth', 3, 'MarkerSize', 5)
```

and the right-hand plot width

```
plot(x, y, '--rs', 'MarkerSize', 20, 'MarkerFaceColor', 'g')
```

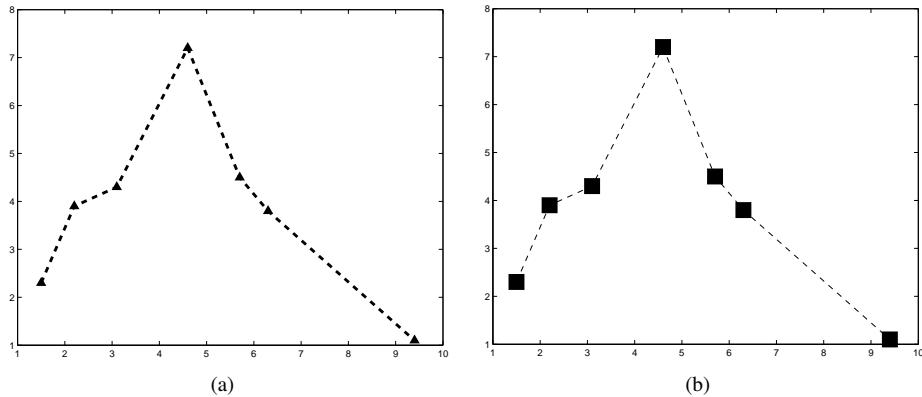
Note that more than one set of data can be passed to `plot`. For example,

```
plot(x, y, 'g-', b, c, 'r--')
```

superimposes plots of  $x(i)$ ,  $y(i)$  and  $b(i)$ ,  $c(i)$  with solid green and dashed red line styles, respectively.

The `plot` command accepts matrices as input arguments. If  $x$  is an  $m$  vector and  $Y$  is an  $m \times n$  matrix, then `plot(x, Y)` plots the graph obtained from  $x$  and each column of  $Y$ . The next example plots sin and cos on the same graph. The output is shown in Figure 2.3.

```
x=(-pi:pi/50:2*pi)';
Y=[sin(x),cos(x)];
plot(x,Y)
```

Figure 2.2: Two nondefault  $x$ - $y$  graphs

Marker	
o	Circle
*	Asterisk
.	Point
+	Plus
x	Cross
s	Square
d	Diamond
^	Upward triangle
v	Downward triangle
>	Right triangle
<	Left triangle
p	Five-point star
h	Six-point star

Line style	
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Table 2.1: Options for the `plot` command

```
xlabel('x');
ylabel('y');
title('Sine and cosine')
box off
```

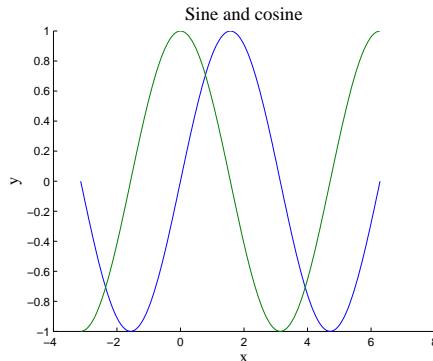


Figure 2.3: Two plots on same graph

In this example we used `title`, `xlabel` and `ylabel`. These functions reproduce their input string above the plot and besides the  $x$ - and  $y$ -axes, respectively. We also used the command `box off`, which removes the box from the current plot, leaving just the  $x$ - and  $y$ -axes.

Similarly, if  $X$  and  $Y$  are matrices of the same size, `plot(X, Y)` plots graphs obtained from corresponding columns of  $X$  and  $Y$ . The example below generates a sawtooth graph (see Figure 2.4 for output):

```
>> x = [ 0:3; 1:4 ] ; y = [ zeros(1,4); ones(1,4) ];
>> plot(x,y,'b'), axis equal
```

If the arguments of `plot` are not real, their imaginary parts are ignored. There is, nevertheless an exception, when `plot` has a single argument. If `plot(Y)` is complex, `plot(Y)` is equivalent with `plot(real(Y), imag(Y))`. If  $Y$  is real, `plot(Y)` takes the subscripts of  $Y$  as abscissas and  $Y$  itself as ordinates.

If one plotting command is later followed by another then the new picture will either replace or be superimposed on the old picture, depending on the current `hold` state. Typing `hold on` causes subsequent plots to be superimposed on the current one, whereas `hold off` specifies that each new plot should start afresh. The default status correspond to `hold off`.

The `fplot` is a more elaborate version of `plot`, useful for plotting mathematical functions. It adaptively samples a function at enough points to produce a representative graph. The general syntax of this function is

```
fplot('fun',lims,tol,N,'LineSpec',p1,p2,...)
```

The argument list works as follows.

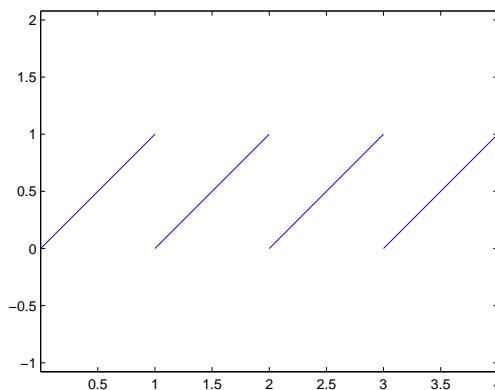


Figure 2.4: Plot of matrices

- `fun` is the function to be plotted.
- The  $x$  and/or  $y$  limits are given by `lims`.
- `tol` is a relative error tolerance, the default value of  $2 \times 10^{-3}$  corresponding to the 0.2% accuracy.
- At least  $N+1$  points will be used to produce the plot.
- `LineSpec` determines the line type.
- `p1, p2, ...` are parameters that are passed to `fun`, which must have input arguments  $x, p1, p2, \dots$

Here is an example:

```
fplot(exp(sqrt(x)*sin(12*x)), [0 2*pi])
```

We can represent polar curves using `polar(t, r)` command, where  $t$  is the polar angle, and  $r$  is the polar radius. An additional string parameter `s`, if used, has the same meaning as in `plot`. The graph of a polar curve, called cardioid, whose equation is

$$r = a(1 + \cos t), \quad t \in [0, 2\pi],$$

where  $a$  is a given constant, is given in Figure 2.5 and is generated by

```
t=0:pi/50:2*pi;
a=2; r=a*(1+cos(t));
polar(t,r)
title('cardioid')
```

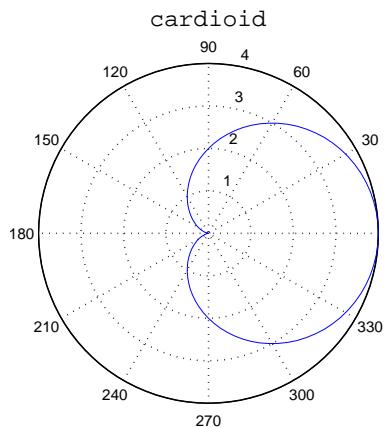


Figure 2.5: Polar graph — cardioid

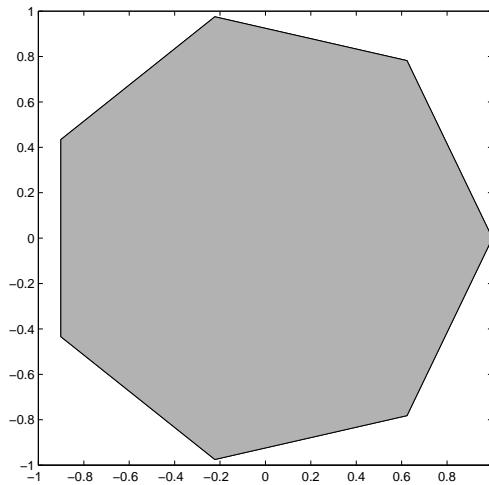
The `fill` function works analogously to `plot`. By typing `fill(x,y,c)` shades a polygon whose vertices are specified by the points  $x(i)$ ,  $y(i)$ . The points are taken in order, and the last vertex is joined to the first. The color of the shading can be given explicitly, or as an  $[r\ g\ b]$  triple. The elements  $r$ ,  $g$  and  $b$ , which must be scalars in the range  $[0,1]$ , determine the level of red, green and blue, respectively in the shading. So, `fill(x,y,[0 1 0])` uses pure green and `fill(x,y,[1 0 1])` uses magenta. Specifying equal amounts of red, green and blue gives a grey shading that can be varied between black ( $[0\ 0\ 0]$ ) and white ( $[1\ 1\ 1]$ ). The next example plots a gray regular heptagon:

```
n=7;
t=2*(0:n-1)*pi/n;
fill(cos(t),sin(t),[0.7,0.7,0.7])
axis square
```

Figure 2.6 gives the resulting picture.

The command `clf` clears the current figure window, while `close` closes it. It is possible to have several figure windows on the screen. The simplest way to create a new figure window is to type `figure`. The  $n$ th figure window (where  $n$  is displayed in the title bar) can be made current by typing `figure(n)`. The command `close all` causes all the figure windows to be closed.

Note that many aspects of a figure can be changed interactively, after the figure has been displayed, by using the items on the toolbar of the figure window or on the Tools pull-down menu. In particular, it is possible to zoom on a particular region of the plot using mouse clicks (see `help zoom`).

Figure 2.6: Example with `fill`

### 2.1.2 Axes and Annotation

Various aspects of the axes of a plot can be controlled with the `axis` command. The axes are removed from a plot with `axis off`. The aspect ratio can be set to unity, so that, for example, a circle appears circular rather than elliptical, by typing `axis equal`. The axis box can be made square with `axis square`.

Setting `axis([xmin xmax ymin ymax])` causes the *x*-axis to run from `xmin` to `xmax` and the *y*-axis from `ymin` to `ymax`. To return to the default axis scaling, which MATLAB chooses automatically based on the data being plotted, type `axis auto`. If you want one of the limits to be chosen automatically by MATLAB, set it to `-inf` or `inf`; for example, `axis([-1 1 -inf 0])`. The *x*-axis and *y*-axis limits can be set individually with `xlim([xmin xmax])` and `ylim([ymin ymax])`.

The next example plots the function  $1/(x-1)^2 + 3/(x-2)^2$  over the interval  $[0,3]$ :

```
x = linspace(0,3,500);
plot(x,1./(x-1).^2+3./(x-2).^2)
grid on
```

The `grid on` produces a light grid of horizontal and vertical grid of dashed lines that extends from axis ticks. You can see the result in Figure 2.7(a). Because of the singularities at  $x = 1, 2$  the plot is not too informative. However, by executing the additional command

```
ylim([0,50])
```

the Figure 2.7(b) is produced, which focuses on the interesting part of the first plot.

In the following example we plot the epicycloid

$$\left. \begin{aligned} x(t) &= (a+b)\cos(t) - b\cos((a/b+1)t) \\ y(t) &= (a+b)\sin(t) - b\sin((a/b+1)t) \end{aligned} \right\} \quad 0 \leq t \leq 10\pi,$$

for  $a = 12$  and  $b = 5$ .

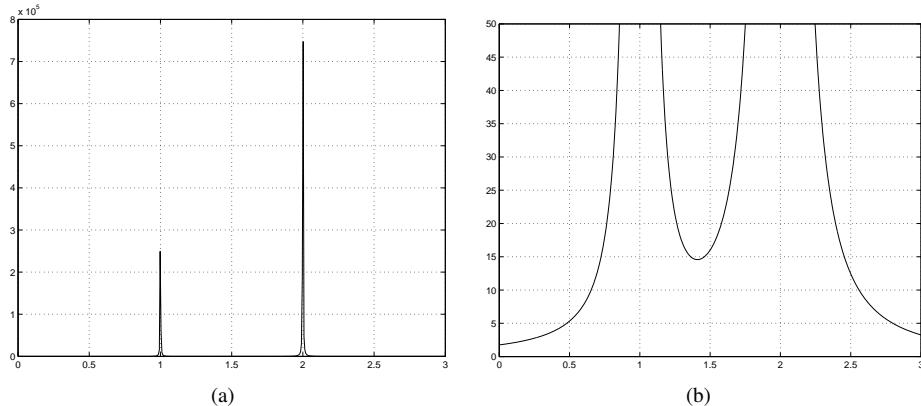


Figure 2.7: Use of `ylim` (right) to change automatic (left)  $y$ -axis limits.

---

### MATLAB Source 2.1 Epicycloid

---

```
a = 12; b=5;
t=0:0.05:10*pi;
x = (a+b)*cos(t)-b*cos((a/b+1)*t);
y =(a+b)*sin(t)-b*sin((a/b+1)*t);
plot(x,y)
axis equal
axis([-25 25 -25 25])
grid on
title('Epicycloid: $a=12$, $b=5$',...
    'Interpreter','LaTeX','FontSize',16)
xlabel('$x(t)$','Interpreter','LaTeX','FontSize',14),
ylabel('$y(t)$','Interpreter','LaTeX','FontSize',14)
```

---

The resulting picture appears in Figure 2.8. The `axis` limits were chosen to put some space around the epicycloid.

We can introduce texts in our graphs by using `text(x, y, s)`, where `x` and `y` are the coordinates of the text, and `s` is a string or a string-type variable. (A related function `gtext` allows the text location to be determined interactively via the mouse.) MATLAB allows to introduce in `s` some constructions borrowed from TeX, such as `_` for a subscript, `^` for a superscript, or Greek letters (`\alpha`, `\beta`, `\gamma`, etc.). Also, we can select certain text attributes, like font, font size and so on.

The commands

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi),pi,0,'o')
text(pi,0,' \leftarrow sin(\pi)', 'FontSize',18)
```

annotate the point of coordinates  $(\pi, 0)$  by the string  $\sin(\pi)$ . The result is given in Figure 2.9. We may use these capabilities in title, legends or axes labels, since these are text objects. Starting from MATLAB 7, text primitives support a strong `LATEX` subset. The corresponding

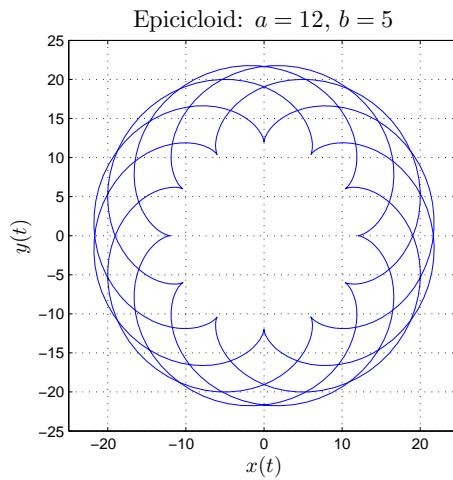
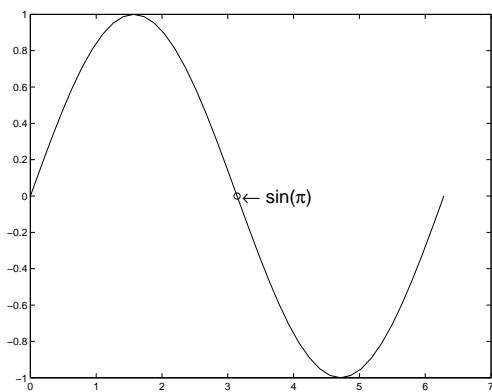


Figure 2.8: Epicycloid

Figure 2.9: Using `text` example

property is called `Interpreter` and its possible values are `TeX`, `LaTeX` or `none`. For examples of `LATEX` constructions usage see the example with epicycloid (MATLAB Source 2.1) or the script `graphLegendre.m`, page 164. See also `doc text` and click here on `Text Properties` for additional information.

Generally, typing `legend('string1','string2',...,'stringn')` will create a legend box that puts '`stringi`' next to the color/marker/line style information for the corresponding plot. By default, the box appears in the top right-hand corner of the axis area. The location of the box can be specified by adding an extra argument (see `help legend` or `doc legend`). The next example add a legend to a graph of hyperbolic sine and cosine (output in Figure 2.10):

```
x = -2:2/25:2;
plot(x,cosh(x),'-ro',x,sinh(x),'-.b')
h = legend('cosh','sinh',4);
```

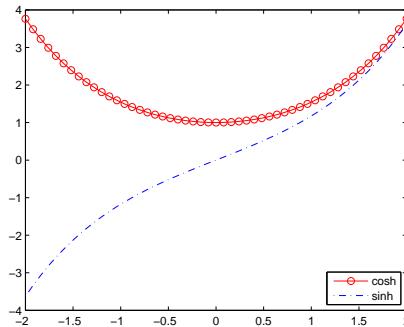


Figure 2.10: Graph with a legend

### 2.1.3 Multiple plots in a figure

MATLAB's `subplot` allows you to place a number of plots in a grid pattern together on the same figure. Typing `subplot(mnp)` or equivalently, `subplot(m,n,p)`, splits the figure window into an  $m$ -by- $n$  array of regions, each having its own axis. The current plotting commands will then apply to the  $p$ th of these regions, where the count moves along the first row, and then along the second row, and so on. So, for example, `subplot(425)` splits the figure window into a 4-by-2 matrix of regions and specifies that plotting commands apply to the fifth region, that is, the first region in the third row. If `subplot(427)` appears later, then the (4,1) position becomes active.

The following example generates the graphs in Figure 2.11.

```
t = 0:.1:2*pi;
subplot(2,2,1)
plot(cos(t),t.*sin(t))
subplot(2,2,2)
```

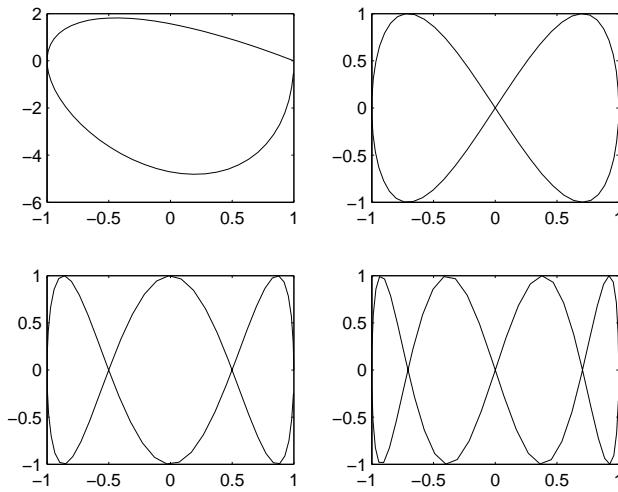


Figure 2.11: Example with subplot

```
plot(cos(t),sin(2*t))
subplot(2,2,3)
plot(cos(t),sin(3*t))
subplot(2,2,4)
plot(cos(t),sin(4*t))
```

To complete this section, we list in Table 2.2 the most popular 2D plotting functions in MATLAB.

## 2.2 Three-Dimensional Graphics

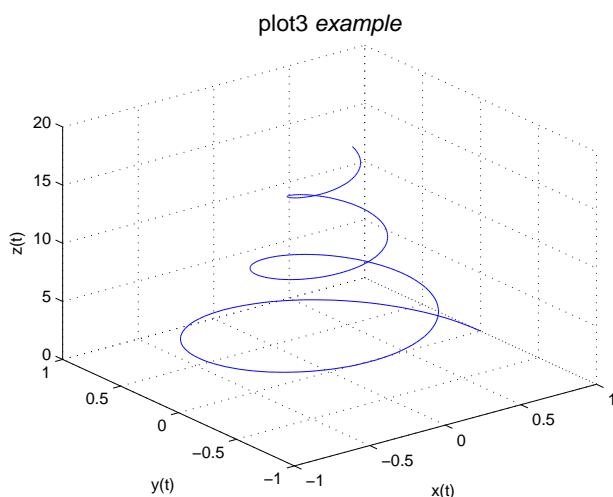
The function `plot3` is the three-dimensional analogue of `plot`. The following example illustrates the simple usage: `plot3(x,y,z)` draws a “join the dots” curve by taking the points  $x(i)$ ,  $y(i)$ ,  $z(i)$  in order. The result is shown in figure 2.12.

```
t = 0:pi/50:6*pi;
expt = exp(-0.1*t);
xt = expt.*cos(t); yt = expt.*sin(t);
plot3(xt, yt, t), grid on
xlabel('x(t)'), ylabel('y(t)'), zlabel('z(t)')
title('plot3 {\itexample}', 'FontSize', 14)
```

This example also uses the functions `xlabel`, `ylabel` and `title`, which were discussed in the previous section and the analogous `zlabel`. Note that we used the TeX notation in the title command to produce the italic text. The color, marker and line styles for

plot	Simple $x$ - $y$ plot
loglog	Plot with logarithmically scaled axis
semilogx	Plot with logarithmically scaled $x$ -axis
semilogy	Plot with logarithmically scaled $y$ -axis
plotyy	$x$ - $y$ plot with $y$ -axes on left and right
polar	Plot in polar coordinates
fplot	Automatic function plot
ezplot	Easy-to-use version of fplot
ezpolar	Easy-to-use version of polar
fill	Polygon fill
area	Filled area graph
bar	Bar graph
barh	Horizontal bar graph
hist	Histogram
pie	Pie chart
comet	Animated, comet-like $x$ - $y$ plot
errorbar	Error bar plot
quiver	Quiver (velocity vector) plot
scatter	Scatter plot

Table 2.2: 2D plotting functions

Figure 2.12: 3D plot created with `plot3`

`plot3` can be controlled in the same way as for `plot`. The axis limit in 3D are automatically computed, but they can be changed by

```
axis([xmin, xmax, ymin, ymax, zmin, zmax])
```

In addition to `xlim`, `ylim`, there exists `zlim`, that changes  $z$ -axis limits.

To plot a bivariate function  $z = f(x, y)$ , we need to define it on a rectangular grid  $\{x_i : i = 1, \dots, m\} \times \{y_j : j = 1, \dots, m\}$ ,  $z_{i,j} = f(x_i, y_j)$ . All you need is some way to get your data into a matrix format. If you have a vector of  $x$  values, and a vector of  $y$  values, MATLAB provides a useful function called `meshgrid` that can be used to simplify the generation of  $X$  and  $Y$  matrix arrays used in 3-D plots. It is invoked using the form  $[X, Y] = \text{meshgrid}(x, y)$ , where  $x$  and  $y$  are vectors that help specify the region in which coordinates, defined by element pairs of the matrices  $X$  and  $Y$ , will lie. The matrix  $X$  will contain replicated rows of the vector  $x$ , while  $Y$  will contain replicated columns of vector  $y$ . The next example will show how `meshgrid` works:

```
x = [-1 0 1];
y = [9 10 11 12];
[X, Y] = meshgrid(x, y)
```

MATLAB returns

```
X =
-1      0      1
-1      0      1
-1      0      1
-1      0      1
Y =
  9      9      9
 10     10     10
 11     11     11
 12     12     12
```

The command `meshgrid(x)` is equivalent to `meshgrid(x, x)`.

The `mesh` function generate a wire-frame plotting of a surface. It creates many criss-crossed lines that look like a net draped over the surface defined by your data. To understand what the command is plotting, consider three M-by-N matrices,  $X$ ,  $Y$ , and  $Z$ , that together specify coordinates of some surface in a three-dimensional space. A mesh plot of these matrices can be generated with the command `mesh(X, Y, Z)`. Each  $(x(i, j), y(i, j), z(i, j))$  triplet, corresponding to the element in the  $i$ th row and  $j$ th column of each of the  $X$ ,  $Y$ , and  $Z$  matrices, is connected to the triplets defined by the elements in neighboring columns and rows. Vertices defined by triplets created from elements that are not in either an outer (i.e., first or last) row or column of the matrix will, therefore, be joined to four adjacent vertices. Vertices on the edge of the surface will be joined to three adjacent ones. Finally, vertices defining the corners of the surface will be joined only to the two adjacent ones. Consider the following example which will produce the plot shown in Figure 2.13(a).

```
[X, Y] = meshgrid(linspace(0, 2*pi, 50), linspace(0, pi, 50));
Z = sin(X).*cos(Y);
mesh(X, Y, Z)
xlabel('x'); ylabel('y'); zlabel('z');
axis([0 2*pi 0 pi -1 1])
```

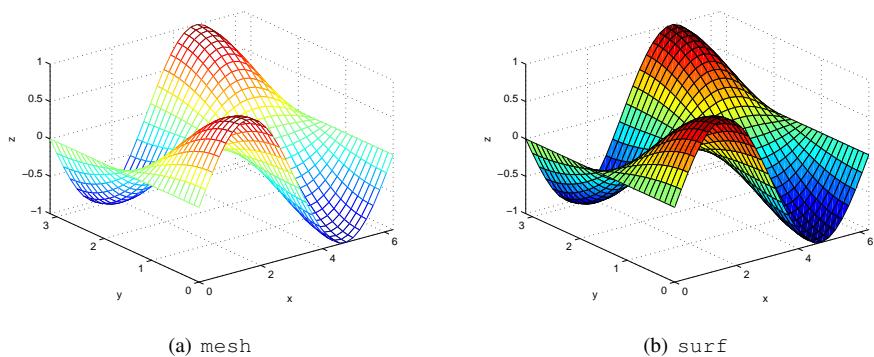


Figure 2.13: Surface plotted with `mesh` and `surf`

The `surf` function produce similar plots as `mesh`, except it paints the inside of each mesh cell with color, so an image of surfaces is created. If, in the previous example, we replace `mesh` by `surf`, we obtain Figure 2.13(b).

The view parameters of a plot may be changed by `view([Az, El])` or `view(x, y, z)`. Here  $Az$  is the azimuthal angle and  $El$  is the elevation angle (in degrees). The angles are defined with respect to the axis origin, where the azimuth angle,  $Az$ , is in the  $xy$ -plane and the elevation angle,  $El$ , is relative to the  $xy$  plane. Figure 2.14 depicts how to interpret the azimuth and elevation angles relative to the plot coordinate system. The default values are

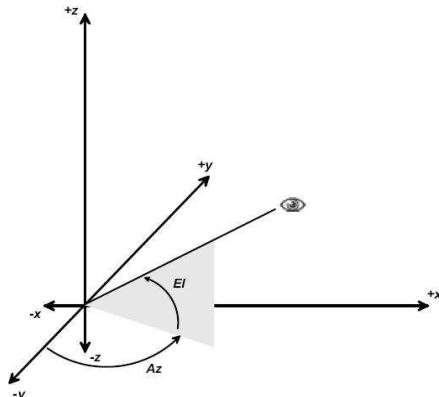


Figure 2.14: The point-of-view in a 3-D plot

$Az = -37.5^\circ$  and  $El = 30^\circ$ . The forms of the function `view(3)` and `view(2)` will restore the current plot to the default 3-D or 2-D views respectively. The form `[Az, El]=view`

returns the current view parameters. Figure 2.15 presents multiple views of the function peaks (see `help peaks`), created with the following code.

```
azrange=-60:20:0;
elrange=0:30:90;
spr=length(azrange);
spc=length(elrange);
pane=0;
for az=azrange
    for el=elrange
        pane=1+pane;
        subplot(spr,spc,pane);
        [x,y,z]=peaks(20);
        mesh(x,y,z);
        view(az,el);
        tstring=['Az=',num2str(az),...
                  ' El=',num2str(el)];
        title(tstring)
        axis off
    end
end
```

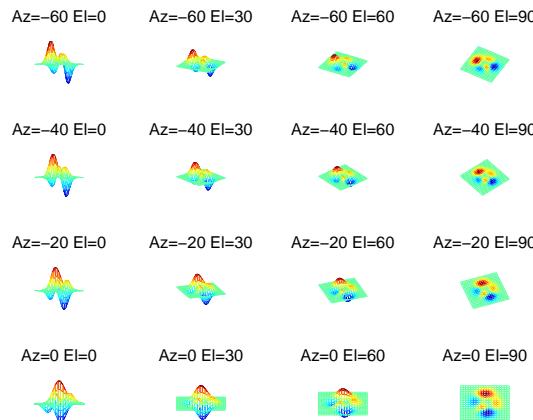


Figure 2.15: Various views

Contour of a function in a two-dimensional array can be plotted by `contour`. Its basis synopsis is

```
contour(X,Y,Z,level)
```

Here `X`, `Y`, and `Z` have the same meaning as in `mesh` or `surf`, and `level` is a vector containing the levels of contour. `level` may be replaced by an integer, which will be interpreted as the number of contour levels. The next examples produces contours for the function

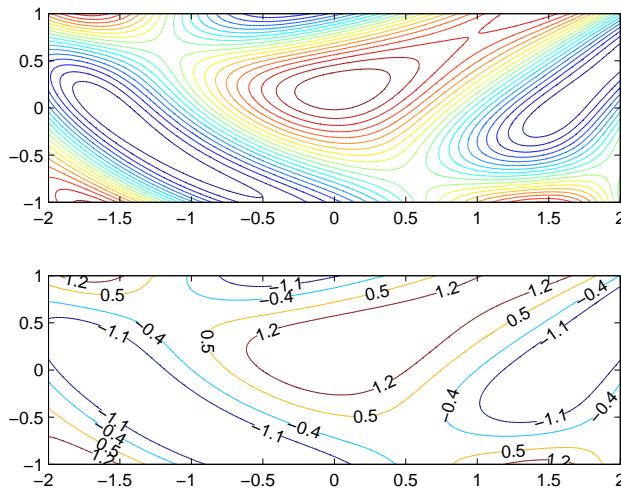


Figure 2.16: Two plots generated with `contour`.

$\sin(3y - x^2 + 1) + \cos(2y^2 - 2x)$  over the range  $-2 \leq x \leq 2$  and  $-1 \leq y \leq 1$ ; the result can be seen in Figure 2.16.

```
x=-2:.01:2; y=-1:0.01:1;
[X,Y] = meshgrid(x,y);
Z = sin(3*Y-X.^2+1)+cos(2*Y.^2-2*X);
subplot(2,1,1)
contour(X,Y,Z,20);
subplot(2,1,2)
[C,h]=contour(X,Y,Z,[1.2,0.5,-0.4,-1.1]);
clabel(C,h)
```

The upper half was generated by choosing automatically 20 contour level. The lower half annotates the contour with `clabel`.

For applications in mechanics which use `contour` function see [51].

We can improve the appearance of our graphs by using `shading` function, that controls the color shading of surface and patch graphics objects. It has three options

- `shading flat` – each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.
- `shading faceted` – flat shading with superimposed black mesh lines. This is the default shading mode.
- `shading interp` varies the color in each line segment and face by interpolating the colors across the line or face.

Figure 2.17 example shows the effects of shading on the hyperbolic paraboloid surface given by the equation  $z = x^2 - y^2$  over the domain  $[-2, 2] \times [-2, 2]$ . Here is the code:

```
[X,Y]=meshgrid(-2:0.25:2);
Z=X.^2-Y.^2;
subplot(1,3,1)
surf(X,Y,Z);
axis square
shading flat
title('shading flat','FontSize',14)

subplot(1,3,2)
surf(X,Y,Z);
axis square
shading faceted
title('shading faceted','FontSize',14)

subplot(1,3,3)
surf(X,Y,Z);
axis square
shading interp
title('shading interp','FontSize',14)
```

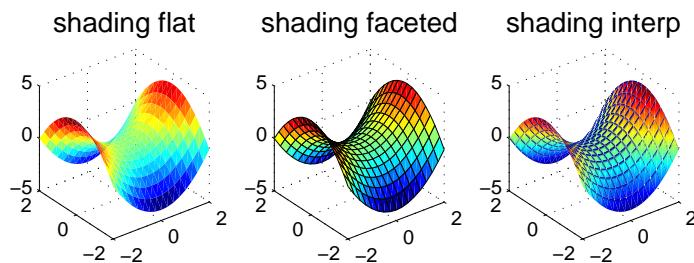


Figure 2.17: The effect of shading

The camera toolbar is a set of tools which allow you to change interactively figure, view-

plot3*	Simple $x$ - $y$ - $z$ plot
contour*	Contour plot
contourf*	Filled contour plot
contour3	3D contour plot
mesh*	Wireframe surface
meshc*	Wireframe surface plus contours
meshz	Wireframe surface with curtain
surf*	Solid surface
surfc*	Solid surface plus contours
waterfall	Unidirectional wire frame
bar3	3D bar graph
bar3h	3D horizontal bar graph
pie3	3D pie chart
fill3	Polygon fill
comet3	3D animated, comet like plot
scatter3	3D scatter plot
stem3	Stem plot

These functions fun have `ezfun` counterparts, too

Table 2.3: 3D plotting functions

ing, light and other parameters of your plot. It is available from Figure window.

The table 2.3 summarizes the most popular 3D plotting functions. As the table indicates, several of the functions have “easy to use” alternative versions with names beginning with `ez`.

## 2.3 Handles and properties

Every rendered object has an identifier or handle. The functions `gcf`, `gca`, and `gco` return the handles to the active figure, axes, and object (usually the most recently drawn). Properties can be accessed and changed by the functions `get` and `set`, or interactively in graphical mode (see the Plot Edit Toolbar, Plot Browser, and Property Editor in the figures View menu). Here is just a taste of what you can do:

```
>> h = plot(t,sin(t));
>> set(h,'Color','m','LineWidth',2,'Marker','s')
>> set(gca,'pos',[0 0 1 1],'visible','off')
```

Here is a way to make a “dynamic” graph or simple animation:

```
clf, axis([-2 2 -2 2]), axis equal
h = line(NaN,NaN,'marker','o','linestyle','-', 'erasemode','none');
t = 6*pi*(0:0.02:1);
for n = 1:length(t)
    set(h,'XData',2*cos(t(1:n)), 'YData',sin(t(1:n)))
    pause(0.05)
end
```

From this example you see that even the data displayed by an object are settable properties (`XData` and `YData`). Among other things, this means you can extract the precise values used to plot any object from the object itself.

Because of the way handles are used, plots in MATLAB are usually created first in a basic form and then modified to look exactly as you want. However, it can be useful to change the default property values that are first used to render an object. You can do this by resetting the defaults at any level above the target objects type. For instance, to make sure that all future Text objects in the current figure have font size 10, enter

```
>> set(gcf, 'DefaultTextFontSize', 10)
```

All figures are also considered to be children of a `root` object that has handle 0, so this can be used to create global defaults. Finally, you can make it a global default for all future MATLAB sessions by placing the `set` command in a file called `startup.m` in a certain directory (see the documentation).

## 2.4 Saving and Exporting Graphs

The `print` command allows you to send your graph to a printer or to save it on disk in a graphical format or as an M-file. The syntax is:

```
print -ddevice -options filename
```

It has several options, type `help print` to see them. Among the devices, we mention here `deps` for encapsulated postscript, `dpng` for Portable Network Graphic format and `djpeg -<nn>` for jpeg image at quality level nn.

If you have set your printer properly, the command `print` will send the content of your figure to it. The command

```
print -deps2 myfig.eps
```

creates an encapsulated level 2 black and white PostScript file `myfig.eps` that can subsequently be printed on a PostScript printer or included in a document. This file can be incorporated into a LATEX document, as in the following outline:

```
\documentclass{article}
\usepackage[dvips]{graphics}

...
\begin{document}
...
\begin{figure}
\begin{center}
\includegraphics[width=8cm]{myfig.eps}
\end{center}
\caption{...}
\end{figure}
...
\end{document}
```

The `saveas` command saves a figure to a file in a form that can be reloaded into MATLAB. For example,

```
saveas(gcf,'myfig','fig')
```

saves the current figure as a binary FIG-file, which can be reloaded into MATLAB with `open('myfig.fig')`.

It is also possible to save and print figures from the pulldown File menu in the figure window.

## 2.5 Application - Snail and Shell Surfaces

Consider the family of parametric surfaces:

$$\begin{aligned}x(u,v) &= e^{wu} (h + a \cos v) \cos cu, \\y(u,v) &= e^{wu} (h + a \cos v) \sin cu, \\z(u,v) &= e^{wu} (k + b \sin v),\end{aligned}\tag{2.5.1}$$

where  $u \in [u_{\min}, u_{\max}]$ ,  $v \in [0, 2\pi]$ , and  $a, b, c, h, k, w$ , and  $R$  are given parameters.  $R$  is a direction parameter and may have only  $\pm 1$  values, and the endpoints for  $u$  interval,  $u_{\min}$  and  $u_{\max}$  are given. These surfaces are used to model the shape of some snails house and shells' shell.

The MATLAB function `SnailsandShells` (see MATLAB source 2.2) computes the points of a surfaces given by equations (2.5.1).

---

### MATLAB Source 2.2 Snail/Shell surface

---

```
function [X,Y,Z]=SnailsandShells(a,b,c,h,k,w,umin,umax,R,nu,nv)
%SNAILSANDSHELLS - plot snails & shell surface
%call SNAILSANDSHELLS(A,B,C,H,K,W,R,UMIN,UMAX]
%R = +/-1 direction
%a, b, c, h, k, w - shape parameters
%umin, umax - interval for u
%nu,nv - number of points

if nargin<11, nv=100; end
if nargin<10, nu=100; end
if nargin<9, R=1; end
v=linspace(0,2*pi,nv);
u=linspace(umin,umax,nu);
[U,V]=meshgrid(u,v);
ewu=exp(w*U);
ewv=exp(w*V);
X=(h+a*cos(V)).*ewu.*cos(c*U);
Y=R*(h+a*cos(V)).*ewu.*sin(c*U);
Z=(k+b*sin(V)).*ewv;
```

---

We illustrate the usage of this source to model the house of *Pseudoheliceras subcatenatum*:

```
%Pseudoheliceras subcatenatum
```

```
a=1.6; b=1.6; c=1; h=1.5; k=-7.0;
w=0.075; umin=-50; umax=-1;

[X,Y,Z]=SnailsandShells(a,b,c,h,k,w,umin,umax,1,512,512);
surf(X,Y,Z)
view(87,21)
shading interp
camlight right
light('Position', [3, -0.5, -4])
axis off
```

To obtain a pleasant-looking graph we invoked light and camera functions. The `light` function add a light object to the current axes. Here, only `Position` property of a light object is used. The `camlight` function creates and sets position of a light. For details on lighting see the corresponding helps or [59].

We recommend the user to try the following funny examples (we give the species and parameters):

- *Nautilus*, with  $a = 1$ ,  $b = 0.6$ ,  $c = 1$ ,  $h = 1$ ,  $k = 0$ ,  $w = 0.18$ ,  $u_{\min} = -20$ , and  $u_{\max} = 1$ .
- *Natica stelata*, with  $a = 2.6$ ,  $b = 2.4$ ,  $c = 1.0$ ,  $h = 1.25$ ,  $k = -2.8$ ,  $w = 0.18$ ,  $u_{\min} = -20$ , and  $u_{\max} = 1.0$ .
- *Mya arenaria*, with  $a = 0.85$ ,  $b = 1.6$ ,  $c = 3.0$ ,  $h = 0.9$ ,  $k = 0$ ,  $w = 2.5$ ,  $u_{\min} = -1$ , and  $u_{\max} = 0.52$ .
- *Euhoplites*, with  $a = 0.6$ ,  $b = 0.4$ ,  $c = 1.0$ ,  $h = 0.9$ ,  $k = 0.0$ ,  $w = 0.1626$ ,  $u_{\min} = -40$ , and  $u_{\max} = -1$ .
- *Bellerophina*, with  $a = 0.85$ ,  $b = 1.2$ ,  $c = 1.0$ ,  $h = 0.75$ ,  $k = 0.0$ ,  $w = 0.06$ ,  $u_{\min} = -10$ , and  $u_{\max} = -1$ .
- *Astroceras*, with  $a = 1.25$ ,  $b = 1.25$ ,  $c = 1.0$ ,  $h = 3.5$ ,  $k = 0$ ,  $w = 0.12$ ,  $u_{\min} = -40$ , and  $u_{\max} = -1$ .

See the front and the back cover for a plot of some of them.

For details on geometry of this kind of surfaces see [36, 63].

## Problems

**Problem 2.1 (Lissajous curve (Bodwitch)).** Plot the parametric curve

$$\alpha(t) = (a \sin(nt + c), b \sin t), \quad t \in [0, 2\pi],$$

for (a)  $a = 2$ ,  $b = 3$ ,  $c = 1$ ,  $n = 2$ , (b)  $a = 5$ ,  $b = 7$ ,  $c = 9$ ,  $n = 4$ , (c)  $a = b = c = 1$ ,  $n = 10$ .

**Problem 2.2 (Butterfly curve).** Plot the polar curve

$$r(t) = e^{\cos(t)} - a \cos(bt) + \sin^5(ct),$$

for (a)  $a = 2, b = 4, c = 1/12$ , (b)  $a = 1, b = 2, c = 1/4$ , (c)  $a = 3, b = 1, c = 1/2$ . Experiment for  $t$  ranging various intervals of the form  $[0, 2k\pi]$ ,  $k \in \mathbb{N}$ .

**Problem 2.3 (Spherical rodonea).** Plot the tridimensional curve

$$\alpha(t) = a(\sin(nt) \cos t, \sin(nt) \sin t, \cos(nu)),$$

for (a)  $a = n = 2$ , (b)  $a = 1/2, n = 1$ , (c)  $a = 3, n = 1/4$ , (d)  $a = 1/3, n = 5$ .

**Problem 2.4.** [27] Play the “chaos game”. Let  $P_1, P_2$ , and  $P_3$  be the vertices of an equilateral triangle. Start with a point anywhere inside the triangle. At random, pick one of the three vertices and move halfway toward it. Repeat indefinitely. If you plot all the points obtained, a very clear pattern will emerge. (Hint: This is particularly easy to do if you use complex numbers. If  $z$  is complex, then `plot(z)` is equivalent to `plot(real(z), imag(z))`.)

**Problem 2.5.** Make surface plots of the following functions over the given ranges.

- (a)  $(x^2 + 3y^2)e^{-x^2-y^2}$ ,  $-3 \leq x \leq 3$ ,  $-3 \leq y \leq 3$ .
- (b)  $-3y/(x^2 + y^2 + 1)$ ,  $|x| \leq 2$ ,  $|y| \leq 4$ .
- (c)  $|x| + |y|$ ,  $|x| \leq 1$ ,  $|y| \leq 1$ .

**Problem 2.6.** Make contour plots of the functions in the previous exercise.

**Problem 2.7.** [27] Make a contour plot of

$$f(x, y) = e^{-4x^2+2y^2} \cos 8x + e^{-3((2x+1/2)^2+2y^2)}$$

for  $-1.5 < x < 1.5$ ,  $-2.5 < y < 2.5$ , showing only the contour at the level  $f(x, y) = 0.001$ . You should see a friendly message.

**Problem 2.8 (Twisted sphere (Corkscrew), [70]).** Plot the parametric surface

$$\chi(u, v) = (a \cos u \cos v, a \sin u \cos v, a \sin v + bu)$$

where  $(u, v) \in [0, 2\pi) \times [-\pi, \pi)$ , for (a)  $a = b = 1$ , (b)  $a = 3, b = 1$ , (c)  $a = 1, b = 0$ , (d)  $a = 1, b = -3/2$ .

**Problem 2.9 (Helicoid, [70]).** Plot the parametric surface given by

$$\chi(u, v) = (av \cos u, bv \sin u, cu + ev)$$

where  $(u, v) \in [0, 2\pi) \times [-d, d)$ , for (a)  $a = 2, b = c = 1, e = 0$ , (b)  $a = 3, b = 1, c = 2, e = 1$ .

**Problem 2.10.** Plot the parametrical surface

$$\chi(u, v) = (u(3 + \cos(v)) \cos(2u), u(3 + \cos(v)) \sin(2u), u \sin(v) - 3u),$$

for  $0 \leq u \leq 2\pi$ ,  $0 \leq v \leq 2\pi$ .

# CHAPTER 3

---

## Errors and Floating Point Arithmetic

---

Computation errors evaluation is one of the main goal of *Numerical Analysis*. Several type of error which can affect the accuracy may occur:

1. *Input data error*;
2. *Rounding error*;
3. *Approximation error*.

Input data errors are out of computation control. They are due, for example, to the inherent imperfections of physical measures.

Rounding errors are caused since we perform our computation using a finite representation, as usual.

For the third error type, many methods do not provide the exact solution of a given problem  $P$ , even if the computation is carried out exactly (without rounding), but rather the solution of a simpler problem,  $\tilde{P}$ , which approximates  $P$ . As an example we consider the summation of an infinite series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

which could be replaced with a simpler problem  $\tilde{P}$  which consist of a summation of a finite number of series terms. Such an error is called *truncation error* (nevertheless, this name is also use for rounding errors obtained by removing the last digits of the representation – *chopping*). Many approximation problems result by “discretising” the original problem  $P$ : definite integrals are approximated by finite sums, derivatives by differences, and so on. Some authors extend the term “truncation error” to cover also the discretization error.

The aim of this chapter is to study the overall effect of input error and rounding error on a computational result. The approximation errors will be discussed when we expose the numerical methods individually.

## 3.1 Numerical Problems

A *numerical problem* is a combination of a constructive mathematical problem (MP) and a precision specification (PS).

**Example 3.1.1.** Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  and  $x \in \mathbb{R}$ . We wish to compute  $y = f(x)$ . Generally,  $x$  is not representable inside the computer; for this reason we shall use an approximation  $x^* \approx x$ . Also, it is possible that  $f$  could not be computed exactly; we shall replace  $f$  with an approximation  $f_A$ . The computed value would be  $f_A(x^*)$ . So our numerical problem is:

**MP.** Given  $x$  and  $f$ , compute  $f(x)$ .

**PS.**  $|f(x) - f_A(x^*)| < \varepsilon$ , for a given  $\varepsilon$ . ◊

## 3.2 Error Measuring

**Definition 3.2.1.** Let  $X$  be a normed linear space,  $A \subseteq X$  and  $x \in X$ . An element  $x^* \in A$  is an approximation of  $x$  from  $A$  (notation  $x^* \approx x$ ).

**Definition 3.2.2.** If  $x^*$  is an approximation of  $x$  the difference  $\Delta x = x - x^*$  is called error, and

$$\|\Delta x\| = \|x^* - x\| \quad (3.2.1)$$

is an absolute error.

**Definition 3.2.3.** The quantity

$$\delta x = \frac{\|\Delta x\|}{\|x\|}, \quad x \neq 0 \quad (3.2.2)$$

is called a relative error.

**Remark 3.2.4.**

1. Since  $x$  is unknown in practice, one uses the approximation  $\delta x = \frac{\|\Delta x\|}{\|x^*\|}$ . If  $\|\Delta x\|$  is small relatively to  $\|x^*\|$ , then the approximation is accurate.
2. If  $X = \mathbb{R}$ , then it is to use  $\delta x = \frac{\Delta x}{x}$  and  $\Delta x = x^* - x$ . ◊

### 3.3 Propagated error

Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $x = (x_1, \dots, x_n)$  and  $x^* = (x_1^*, \dots, x_n^*)$ . We want to evaluate the absolute error  $\Delta f$ , and the relative error  $\delta f$ , respectively, when  $f$  is approximated by  $f(x^*)$ . These are *propagated errors*, because they describe how the initial error (absolute or relative) is propagated during the computation of  $f$ . Let suppose  $x = x^* + \Delta x$ , where  $\Delta x = (\Delta x_1, \dots, \Delta x_n)$ . For the absolute error, using Taylor's formula we obtain

$$\begin{aligned}\Delta f &= f(x^* + \Delta x_1, \dots, x_n^* + \Delta x_n) - f(x_1^*, \dots, x_n^*) = \\ &= \sum_{i=1}^n \Delta x_i \frac{\partial f}{\partial x_i^*}(x_1^*, \dots, x_n^*) + \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \Delta x_i \Delta x_j \frac{\partial^2 f}{\partial x_i^* \partial x_j^*}(\theta),\end{aligned}$$

where  $\theta \in [(x_1^*, \dots, x_n^*), (x_1^* + \Delta x_1, \dots, x_n^* + \Delta x_n)]$ .

If the quantities  $\Delta x_i$  are sufficiently small, then  $\Delta x_i \Delta x_j$  are negligible with respect to  $\Delta x_i$ , and we have

$$\Delta f \approx \sum_{i=1}^n \Delta x_i \frac{\partial f}{\partial x_i^*}(x_1^*, \dots, x_n^*). \quad (3.3.1)$$

Analogously, for the relative error

$$\begin{aligned}\delta f &= \frac{\Delta f}{f} \approx \sum_{i=1}^n \Delta x_i \frac{\frac{\partial f}{\partial x_i^*}(x^*)}{f(x^*)} = \sum_{i=1}^n \Delta x_i \frac{\partial}{\partial x_i^*} \ln f(x^*) = \\ &= \sum_{i=1}^n x_i^* \delta x_i \frac{\partial}{\partial x_i^*} \ln f(x^*).\end{aligned}$$

Thus

$$\delta f = \sum_{i=1}^n x_i^* \frac{\partial}{\partial x_i^*} \ln f(x^*) \delta x_i. \quad (3.3.2)$$

The inverse problem has also a great importance: what accuracy is needed for the input data such that the result be of a desired accuracy? That is, given  $\varepsilon > 0$ , how much  $\Delta x_i$  or  $\delta x_i$ ,  $i = 1, \dots, n$  would be such that  $\Delta f$  or  $\delta f < \varepsilon$ ? A solution method is based on *equal effects principle*: one supposes all terms which appear in (3.3.1) have (3.3.2) the same effect, i. e.

$$\frac{\partial f}{\partial x_1^*}(x^*) \Delta x_1 = \dots = \frac{\partial f}{\partial x_n^*}(x^*) \Delta x_n.$$

(3.3.1) implies

$$\Delta x_i \approx \frac{\Delta f}{n \left| \frac{\partial f}{\partial x_i^*}(x^*) \right|}. \quad (3.3.3)$$

Analogously,

$$\delta x_i = \frac{\delta f}{n \left| x_i^* \frac{\partial}{\partial x_i^*} \ln f(x^*) \right|}. \quad (3.3.4)$$

## 3.4 Floating-Point Representation

### 3.4.1 Parameters

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation. The floating-point representation parameters are a base  $\beta$  (which is always assumed to be even), a precision  $p$ , and a largest and a smallest allowable exponent,  $e_{\max}$  and  $e_{\min}$ , all being natural numbers. In general, a floating-point number will be represented as

$$x = \pm d_0.d_1d_2 \dots d_{p-1} \times \beta^e, \quad 0 \leq d_i < \beta \quad (3.4.1)$$

where  $d_0.d_1d_2 \dots d_{p-1}$  is called the *significand* or *mantissa*, and  $e$  is the *exponent*. The value of  $x$  is

$$\pm (d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \dots + d_{p-1}\beta^{-(p-1)})\beta^e. \quad (3.4.2)$$

In order to achieve the uniqueness of representation, the floating-point number are *normalized*, that is, we change the representation, not the value, such that  $d_0 \neq 0$ . Zero is represented as  $1.0 \times \beta^{e_{\min}-1}$ . Thus, the numerical ordering of nonnegative real numbers corresponds to the lexicographical ordering of their floating-point representation with exponent stored to the left of the significand.

The term *floating-point number* will be used to mean a real number that can be exactly represented in this format. Each interval  $[\beta^e, \beta^{e+1})$  in  $\mathbb{R}$  contains exactly  $\beta^p$  floating-point



Figure 3.1: The distribution of normalized floating point numbers on the real axis without denormalization

numbers (the number of all possible significands). The interval  $(0, \beta^{e_{\min}})$  is empty; for these reason the *denormalized numbers* are introduced, i.e. numbers whose significand has the form  $0.d_1d_2 \dots d_{p-1}$  and whose exponent is  $\beta^{e_{\min}-1}$ . The availability of denormalization is an additional parameter of the representation. The set of floating-numbers for a set fixed parameters of representation will be denoted

$$\mathbb{F}(\beta, p, e_{\min}, e_{\max}, \text{denorm}), \quad \text{denorm} \in \{\text{true}, \text{false}\}.$$

This set is not equal to  $\mathbb{R}$  because:

1. is a finite subset of  $\mathbb{Q}$ ;
2. for  $x \in \mathbb{R}$  it is possible to have  $|x| > \beta \times \beta^{e_{\max}}$  (overflow) or  $|x| < 1.0 \times \beta^{e_{\min}}$  (underflow).

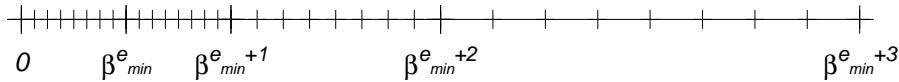


Figure 3.2: The distribution of normalized floating point numbers on the real axis with de-normalization

The usual arithmetic operation on  $\mathbb{F}(\beta, p, e_{\min}, e_{\max}, \text{denorm})$  are denoted by  $\oplus, \ominus, \otimes, \oslash$ , and the name of usual functions are capitalized: SIN, COS, EXP, LN, SQRT, and so on.  $(\mathbb{F}, \oplus, \otimes)$  is not a field since

$$\begin{aligned}(x \oplus y) \oplus z &\neq x \oplus (y \oplus z) & (x \otimes y) \otimes z &\neq x \otimes (y \otimes z) \\ (x \oplus y) \otimes z &\neq x \otimes z \oplus y \otimes z.\end{aligned}$$

In order to measure the error one uses the relative error and *ulps* – units in the *last place*. If the number  $z$  is represented as  $d_0.d_1d_2\dots d_{p-1} \times \beta^e$ , then the error is

$$|d_0.d_1d_2\dots d_{p-1} - z/\beta^e| \beta^{p-1} \text{ulps}.$$

The relative error that corresponds to  $\frac{1}{2}$ ulps is

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2} \text{ulps} \leq \frac{\beta}{2}\beta^{-p},$$

since  $\underbrace{0.0\dots 0}_{p} \beta' \times \beta^e$ , with  $\beta' = \frac{\beta}{2}$ . The value  $\text{eps} = \frac{\beta}{2}\beta^{-p}$  is referred to as *machine epsilon*.

The default rounding obeys the even digit rule: if  $x = d_0.d_1\dots d_{p-1}d_p\dots$  and  $d_p > \frac{\beta}{2}$  then the rounding is upward, if  $d_p < \frac{\beta}{2}$  the rounding is downward, and if  $d_p = \frac{\beta}{2}$  and among the removed digits there exists a nonzero one the rounding is upward; otherwise, the last preserved digit is even. If  $\text{fl}$  denotes the rounding operation, we can define the floating-point arithmetic operation by

$$x \odot y = \text{fl}(x \circ y). \quad (3.4.3)$$

Another kind of rounding can be chosen: to  $-\infty$ , to  $+\infty$ , to 0 (chopping). During the reasoning concerning the floating-point operations we shall use the following model

$$\forall x, y \in \mathbb{F}, \exists \delta \text{ with } |\delta| < \text{eps} \text{ such that } x \odot y = (x \circ y)(1 + \delta). \quad (3.4.4)$$

Intuitively, each floating point arithmetic operation is exact within a boundary of at most  $\text{eps}$  for the relative errors.

The formula (3.4.4) is called *the fundamental axiom of floating-point arithmetic*.

### 3.4.2 Cancellation

From formulae (3.3.2) for the relative error, if  $x \approx x(1 + \delta_x)$  and  $y \approx y(1 + \delta_y)$ , we obtain the relative error for the floating-point arithmetic:

$$\delta_{xy} = \delta_x + \delta_y \quad (3.4.5)$$

$$\delta_{x/y} = \delta_x - \delta_y \quad (3.4.6)$$

$$\delta_{x+y} = \frac{x}{x+y}\delta_x + \frac{y}{x+y}\delta_y \quad (3.4.7)$$

Only subtraction of two nearby quantities  $x \approx y$  is critical; in this case,  $\delta_{x-y} \rightarrow \infty$ . This phenomenon is called *cancellation* and is depicted in Figure 3.3. There  $b, b', b''$  stand for binary digits which are reliable, and the  $g$ s represent binary digits contaminated by errors (garbage digits). Note in that garbage - garbage = garbage, but more important, the normalization of the result moves the first garbage digit from the 12th position to the 3rd.

x	=	1	0	1	1	0	0	1	0	1	b	b	g	g	g	g	e
y	=	1	0	1	1	0	0	1	0	1	b'	b'	g	g	g	g	e
x-y	=	0	0	0	0	0	0	0	0	0	b''	b''	g	g	g	g	e
	=	b''	b''	g	g	g	g	?	?	?	?	?	?	?	?	?	e-9

Figure 3.3: The cancellation phenomenon

We have two kind of cancellation: *benign*, when subtracting exactly known quantities and *catastrophic*, when the subtraction operands are subject to rounding errors. The programmer must be aware of the possibility of its occurrence and he/she must try to avoid it. The expressions which lead to cancellation must be rewritten, and a catastrophic cancellation must be converted into a benign one. We shall give some examples in the sequel.

**Example 3.4.1.** If  $a \approx b$ , then the expression  $a^2 - b^2$  is rewritten into  $(a - b)(a + b)$ . The initial form is preferred when  $a \gg b$  or  $b \gg a$ .  $\diamond$

**Example 3.4.2.** If cancellation appears within an expression containing square roots, then we rewrite:

$$\sqrt{x+\delta} - \sqrt{x} = \frac{\delta}{\sqrt{x+\delta} + \sqrt{x}}, \quad \delta \approx 0. \quad \diamond$$

**Example 3.4.3.** The difference of two values of the same function for nearby arguments is rewritten using Taylor expansion:

$$f(x + \delta) - f(x) = \delta f'(x) + \frac{\delta^2}{2} f''(x) + \dots \quad f \in C^n[a, b]. \quad \diamond$$

**Example 3.4.4.** The solution of a quadratic equation  $ax^2 + bx + c = 0$  can involve catastrophic cancellation when  $b^2 \gg 4ac$ . The usual formulae

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad (3.4.8)$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad (3.4.9)$$

can lead to cancellation as follows: for  $b > 0$  the cancellation affects the computation of  $x_1$ ; for  $b < 0$   $x_2$ . We can correct the situation using the conjugate

$$x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \quad (3.4.10)$$

$$x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}. \quad (3.4.11)$$

For the first case we use formulae (3.4.10) and (3.4.9); for the second case (3.4.8) and (3.4.11). Consider the quadratic equation with  $a = 1$ ,  $b = 100000000$ , and  $c = 1$ . Applying formulas 3.4.8 and 3.4.9 we obtain:

```
>>a=1; c=1; b=-100000000;
>>x1=(-b+sqrt(b^2-4*a*c))/(2*a)

x1 =
100000000

>>x2=(-b-sqrt(b^2-4*a*c))/(2*a)

x2 =
7.45058059692383e-009
```

If we amplify by the conjugate to compute  $x_2$  we have:

```
>>x1=(-b+sqrt(b^2-4*a*c))/(2*a)

x1 =
100000000

>> x2a=2*c/(-b+sqrt(b^2-4*a*c))

x2a =
1e-008
```

The function `root` yields the same results.  $\diamond$

## 3.5 IEEE Standard

There are two different standards for floating point computation: IEEE 754 that require  $\beta = 2$  and IEEE 854 that allows either  $\beta = 2$  or  $\beta = 10$ , but it is more permissive concerning representation.

We deal only with the first standard. Table 3.1 gives its parameters.

Parameter	Format			
	Single	Single Extended	Double	Double extended
p	24	$\geq 32$	53	$\geq 64$
$e_{\max}$	+127	$\geq +1023$	+1023	$\geq +16383$
$e_{\min}$	-126	$\leq -1022$	-1022	$\leq -16382$
Exponent width	8	$\geq 11$	11	$\geq 15$
Number width	32	$\geq 43$	64	$\geq 79$

Table 3.1: IEEE 754 Format Parameters

Why extended formats?

1. A better precision.
2. The conversion from binary to decimal and then back to binary needs 9 digits in single precision and 17 digits in double precision.

The relation  $|e_{\min}| < e_{\max}$  is motivated by the fact that  $1/2^{e_{\min}}$  must not lead to overflow.

The operations  $\oplus, \ominus, \otimes, \oslash$  must be exactly rounded. The accuracy is achieved using two guard digit and a third sticky bit.

The exponent is *biased*, i.e. instead of  $e$  the standard represents  $e + D$ , where  $D$  is fixed when the format is chosen.

For IEEE 754 single precision,  $D = 127$ , and for double precision,  $D = 1023$ .

### 3.5.1 Special Quantities

The IEEE standard specifies the following special quantities:

Exponent	Significand	Represents
$e = e_{\min} - 1$	$f = 0$	$\pm 0$
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \leq e \leq e_{\max}$		$1.f \times 2^e$
$e = e_{\max} + 1$	$f = 0$	$\pm \infty$
$e = e_{\max} + 1$	$f \neq 0$	NaN

**NaN.** In fact we have a family of NaNs. The illegal and indeterminate operations lead to NaN:  $\infty + (-\infty)$ ,  $0 \times \infty$ ,  $0/0$ ,  $\infty/\infty$ ,  $x \text{ REM } 0$ ,  $\infty \text{ REM } y$ ,  $\sqrt{x}$  for  $x < 0$ . If one operand is a NaN the result is a NaN too.

**Infinity.**  $1/0 = \infty$ ,  $-1/0 = -\infty$ . The infinite values allow the continuation of computation when an overflow occurs. This is safer than aborting or returning the largest representable number.

$\frac{x}{1+x^2}$  for  $x = \infty$  gives 0.

**Signed Zero.** We have two zeros:  $+0, -0$ ; the relations  $+0 = -0$  and  $-0 < +\infty$  hold. Advantages: simpler treatment of underflow and discontinuity. We can make a distinction between  $\log 0 = -\infty$  and  $\log x = \text{NaN}$  for  $x < 0$ . Without signed zero we can not make any distinction between the logarithm of a negative number which leads to overflow and the logarithm of 0.

## 3.6 MATLAB Floating-Point Arithmetic

MATLAB uses mainly the IEEE double-precision format. Starting from MATLAB 7, it exists support for single-precision IEEE arithmetic. There is no distinction between integer and real numbers. The command `format hex` is useful to display the floating-point representation. For example, one obtains the representation of 1, -1, 0.1 and golden ratio,  $\phi = (1 + \sqrt{5})/2$ , respectively, by using

```
>> format hex
>> 1, -1
ans =
3ff0000000000000
ans =
bff0000000000000
>> 0.1
ans =
3fb999999999999a
>> phi=(1+sqrt(5))/2
phi =
3ff9e3779b97f4a8
```

One considers that the fraction  $f$  satisfies  $0 \leq f < 1$ , and the exponent  $-1022 \leq e \leq 1023$ . We can characterize the MATLAB floating-point system by three constants: `realmin`, `realmax`, and `eps`. `realmin` is the smallest normalized floating-point number. Any quantity less than either it represents a denormalized number or produces an underflow. `realmax` is the largest representable floating-point number. Anything larger than it produces an overflow. The values of these constants are

	binary	decimal	hexadecimal
<code>eps</code>	$2^{-52}$	2.220446049250313e-016	3cb0000000000000
<code>realmin</code>	$2^{-1022}$	2.225073858507201e-308	001000000000000
<code>realmax</code>	$(2-\text{eps}) \times 2^{1023}$	1.797693134862316e+308	7fefffffffffffffff

Functions in sources 3.1 and 3.2 compute `eps`. It is instructive to type each line of the second variant individually.

It is interesting to try:

**MATLAB Source 3.1** Computation of eps - 1st variant

---

```
function eps=myeps1
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2;
```

---

**MATLAB Source 3.2** Computation of eps - 2nd variant

---

```
function z=myeps2
x=4/3-1;
y=3*x;
z=1-y;
```

---

```
>> format long
>> 2*realmax
ans =
      Inf
>> realmin*eps
ans =
   4.940656458412465e-324
>> realmin*eps/2
ans =
      0
```

or in hexadecimal:

```
>> format hex
>> 2*realmax
ans =
  7ff0000000000000
>> realmin*eps
ans =
  0000000000000001
>> realmin*eps/2
ans =
  0000000000000000
```

The denormalized numbers are within interval  $[eps * realmin, realmin]$ . They are represented taking  $e = -1023$ . The displacement is  $D = 1023$ . The infinity, Inf, is represented taking  $e = 1024$  and  $f = 0$ , and NaN with  $e = 1024$  and  $f \neq 0$ . For example,

```
>> format short
>> Inf-Inf
ans =
      NaN
>> Inf/Inf
```

```
ans =
NaN
```

Two MATLAB functions that take apart and put together floating-point numbers are `log2` and `pow2`. The expression `[F, E]=log2(X)` for a real array `X`, returns an array `F` of real numbers, usually in the range  $0.5 \leq \text{abs}(F) < 1$ , and an array `E` of integers, so that  $X = F \cdot 2^E$ . Any zeros in `X` produce `F = 0` and `E = 0`. The expression `X=pow2(F, E)` for a real array `F` and an integer array `E` computes  $X = F \cdot 2^E$ . The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. In IEEE arithmetics, the command `[F, E] = log2(X)` yields to:

F	E	X
<code>1/2</code>	<code>1</code>	<code>1</code>
<code>pi/4</code>	<code>2</code>	<code>pi</code>
<code>-3/4</code>	<code>2</code>	<code>-3</code>
<code>1/2</code>	<code>-51</code>	<code>eps</code>
<code>1-eps/2</code>	<code>1024</code>	<code>realmax</code>
<code>1/2</code>	<code>-1021</code>	<code>realmin</code>

and `X = pow2(F, E)` recovers `X`.

We have seen in §1.5.4 that in MATLAB 7 there exist support for single-precision floating-point numbers (the function `single`). For example,

```
>> a = single(5);
```

assign to `a` the single-precision floating-point representation of 5. We can compare this with the corresponding double-precision representation:

```
>> b=5;
>> whos
  Name      Size            Bytes  Class
  a          1x1                  4  single array
  b          1x1                  8  double array

Grand total is 2 elements using 12 bytes
```

```
>> format hex
>> a,b
a =
  40a00000
b =
  4014000000000000
```

The conversion from `double` to `single` may affect the value, due to rounding:

```
>> format long
>> single(3.14)
ans =
  3.1400001
```

The result of a binary operation with `single`-type operation has the type `single`. The result of a binary operation between a `single` and a `double` has the type `single`, as the examples below show:

```
>> x = single(2)*single(3)
x =
    6
>> class(x)
ans =
    single
>> x = single(8)+3
x =
    11
>> class(x)
ans =
    single
```

We may call `eps`, `realmin`, and `realmax` for `single`-precision too:

```
>> eps('single')
ans =
    1.1921e-007
>> realmin('single'), realmax('single')
ans =
    1.1755e-038
ans =
    3.4028e+038
```

Starting from MATLAB 7, the function `eps` gives the distance between floating-point numbers. The command `d=eps(x)`, where `x` is a `single` or `double` floating-point number, gives the distance from `abs(x)` to the closest floating-point number greater than `x` and having the same precision as `x`. Example:

```
>> format long
>> eps
ans =
    2.220446049250313e-016
>> eps(5)
ans =
    8.881784197001252e-016
```

`eps('double')` is equivalent to `eps` or to `eps(1.0)`, and `eps('single')` is equivalent to `eps(single(1.0))`.

Clearly, the distance between `single`-precision floating-point numbers is greater than the distance between `double`-precision floating-point numbers. For example,

```
>> x = single(5)
>> eps(x)
returns
```

```
ans =
    4.7683716e-007
```

that is larger than `eps(5)`.

## 3.7 The Condition of a Problem

We may think a problem as a map

$$f : \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad y = f(x). \quad (3.7.1)$$

We are interested in the sensitivity of the map  $f$  at some given point  $x$  to a small perturbation of  $x$ , that is, how much bigger (or smaller) the perturbation in  $y$  is compared to the perturbation in  $x$ . In particular, we wish to measure the degree of sensitivity by a single number – the *condition number* of the map  $f$  at the point  $x$ . The function  $f$  is assumed to be evaluated exactly, with infinite precision, as we perturb  $x$ . *The condition of  $f$ , therefore, is an inherent property of the map  $f$  and does not depend on any algorithmic considerations concerning its implementation.*

It does not mean that the knowledge of the condition of a problem is irrelevant to any algorithmic solution of the problem. On the contrary! The reason is that quite often the *computed* solution  $y^*$  of (3.7.1) (computed in floating point machine arithmetic, using a specific algorithm) can be demonstrated to be the *exact* solution of a “nearby” problem; that is

$$y^* = f(x^*) \quad (3.7.2)$$

where

$$x^* = x + \delta \quad (3.7.3)$$

and moreover, the distance  $\|\delta\| = \|x^* - x\|$  can be estimated in terms of the machine precision. Therefore, if we know how strongly or weakly the map  $f$  reacts to small perturbation, such as  $\delta$  in (3.7.3), we can say something about the error  $y^* - y$  in the solution caused by the perturbation.

We can consider more general spaces for  $f$ , but for practical implementation the finite dimensional spaces are sufficient.

Let

$$\begin{aligned} x &= [x_1, \dots, x_m]^T \in \mathbb{R}^m, \quad y = [y_1, \dots, y_n]^T \in \mathbb{R}^n, \\ y_\nu &= f_\nu(x_1, \dots, x_m), \quad \nu = 1, \dots, n. \end{aligned}$$

We think  $y_\nu$  as a function of one single variable  $x_\mu$

$$\gamma_{\nu\mu} = (\text{cond}_{\nu\mu} f)(x) = \left| \frac{x_\mu \frac{\partial f_\nu}{\partial x_\mu}}{f_\nu(x)} \right|. \quad (3.7.4)$$

These give us a matrix of condition numbers

$$\Gamma(x) = \begin{pmatrix} \frac{x_1 \frac{\partial f_1}{\partial x_1}}{f_1(x)} & \dots & \frac{x_m \frac{\partial f_1}{\partial x_m}}{f_1(x)} \\ \vdots & \ddots & \vdots \\ \frac{x_1 \frac{\partial f_n}{\partial x_1}}{f_n(x)} & \dots & \frac{x_m \frac{\partial f_n}{\partial x_m}}{f_n(x)} \end{pmatrix} = [\gamma_{\nu\mu}(x)] \quad (3.7.5)$$

and we shall consider as *condition number*

$$(\text{cond } f)(x) = \|\Gamma(x)\|. \quad (3.7.6)$$

**Another approach.** We consider the norm  $\|\cdot\|_\infty$

$$\begin{aligned} \Delta y_\nu &\approx \sum_{\mu=1}^m \frac{\partial f_\nu}{\partial x_\mu} \Delta x_\mu (= f_\nu(x + \Delta x) - f_\nu(x)) \\ |\Delta y_\nu| &\leq \sum_{\mu=1}^n \left| \frac{\partial f_\nu}{\partial x_\mu} \right| \Delta x_\mu \leq \max_\mu |\Delta x_\mu| \sum_{\mu=1}^m \left| \frac{\partial f_\nu}{\partial x_\mu} \right| \leq \\ &\leq \max_\mu |\Delta x_\mu| \max_\nu \sum_{\mu=1}^m \left| \frac{\partial f_\nu}{\partial x_\mu} \right| \end{aligned}$$

Therefore

$$\|\Delta y\|_\infty \leq \|\Delta x\|_\infty \left\| \frac{\partial f}{\partial x} \right\|_\infty \quad (3.7.7)$$

where

$$J(x) = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \in \mathbb{R}^n \times \mathbb{R}^m \quad (3.7.8)$$

is the Jacobian matrix of  $f$

$$\frac{\|\Delta y\|_\infty}{\|y\|_\infty} \leq \frac{\|x\|_\infty \left\| \frac{\partial f}{\partial x} \right\|_\infty}{\|f(x)\|_\infty} \cdot \frac{\|\Delta x\|}{\|x\|_\infty}. \quad (3.7.9)$$

If  $m = n = 1$ , then both approaches lead to

$$(\text{cond } f)(x) = \left| \frac{xf'(x)}{f(x)} \right|,$$

for  $x \neq 0, y \neq 0$ .

If  $x = 0 \wedge y \neq 0$ , then we take the absolute error for  $x$  and the relative error for  $y$

$$(\text{cond } f)(x) = \left| \frac{f'(x)}{f(x)} \right|.$$

For  $y = 0 \wedge x \neq 0$  we take the absolute error for  $y$  and the relative error for  $x$ . For  $x = y = 0$

$$(\text{cond } f)(x) = |f'(x)|.$$

**Example 3.7.1 (Systems of linear algebraic equations).** Given a nonsingular square matrix  $A \in \mathbb{R}^{n \times n}$  and a vector  $b \in \mathbb{R}^n$  solve the system

$$Ax = b. \quad (3.7.10)$$

Here the input data are the elements of  $A$  and  $b$ , and the result is the vector  $x$ . To simplify matters let's assume that  $A$  is a fixed matrix not subject to change, and only  $b$  is undergoing perturbations. We have a map  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  given by

$$x = f(b) := A^{-1}b,$$

which is linear. Therefore  $\frac{\partial f}{\partial b} = A^{-1}$  and using (3.7.9),

$$\begin{aligned} (\text{cond } f)(b) &= \frac{\|b\|\|A^{-1}\|}{\|A^{-1}b\|} = \frac{\|Ax\|\|A^{-1}\|}{\|A^{-1}b\|}, \\ \max_{\substack{b \in \mathbb{R}^n \\ b \neq 0}} (\text{cond } f)(b) &= \max_{\substack{x \in \mathbb{R}^n \\ x \neq 0}} \frac{\|Ax\|}{\|x\|} \|A^{-1}\| = \|A\|\|A^{-1}\|. \end{aligned} \quad (3.7.11) \quad \diamond$$

The number  $\|A\|\|A^{-1}\|$  is called the condition number of the matrix  $A$  and we denote it by  $\text{cond } A$ .

$$\text{cond } A = \|A\|\|A^{-1}\|.$$

## 3.8 The Condition of an algorithm

Let us consider the problem

$$f : \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad y = f(x). \quad (3.8.1)$$

Along with the problem  $f$ , we are also given an algorithm  $A$  that solves the problem. That is, given a vector  $x \in \mathbb{F}(\beta, p, e_{min}, e_{max}, \text{denorm})$ , the algorithm  $A$  produces a vector  $y_A$  (in floating-point arithmetic), that is supposed to approximate  $y = f(x)$ . Thus we have another map  $f_A$  describing how the problem  $f$  is solved by the algorithm  $A$

$$f_A : \mathbb{F}^m(\dots) \rightarrow \mathbb{F}^n(\dots), \quad y_A = f_A(x).$$

In order to be able to analyze  $f_A$  in this general terms, we must make a basic assumption, namely, that

$$(BA) \quad \forall x \in \mathbb{F}^m \exists x_A \in \mathbb{F}^m : \quad f_A(x) = f(x_A). \quad (3.8.2)$$

That is, the computed solution corresponding to some input  $x$  is the exact solution for some different input  $x_A$  (not necessarily a machine vector and not necessarily uniquely determined) that we hope is close to  $x$ . The closer we can find an  $x_A$  to  $x$ , the more confidence we should place in the algorithm  $A$ .

We define the *condition of  $A$  at  $x$*  by comparing the relative error with eps:

$$(\text{cond } A)(x) = \inf_{x_A} \frac{\|x_A - x\|}{\|x\|} / \text{eps}.$$

Motivation:

$$\delta_y = \frac{f_A(x) - f(x)}{f(x)} = \frac{(x_A - x)f'(\xi)}{f(x)} \approx \frac{x_A - x}{x} \cdot \frac{1}{\text{eps}} \frac{xf'(x)}{f(x)} \text{eps}.$$

The infimum is over all  $x_A$  satisfying  $y_A = f(x_A)$ . In practice one can take any such  $x_A$  and then obtain an upper bound for the condition number

$$(\text{cond } A)(x) \leq \frac{\frac{\|x_A - x\|}{\|x\|}}{\text{eps}}. \quad (3.8.3)$$

### 3.9 Overall error

The problem to be solved is again

$$f : \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad y = f(x). \quad (3.9.1)$$

This is the mathematical (idealized) problem, where the data are exact real numbers, and the solution is the mathematically exact solution. When solving such a problem on a computer, in floating-point arithmetic with precision  $\text{eps}$ , and using some algorithm  $A$ , one first of all rounds the data, and then applies to these rounded data not  $f$ , but  $f_A$ .

$$x^* = x \text{ rounded}, \quad \frac{\|x^* - x\|}{\|x\|} = \varepsilon, \quad y_A^* = f_A(x^*).$$

Here  $\varepsilon$  represents the rounding error in the data. (It could also be due to sources other than rounding, e.g., measurement.) The total error that we wish to estimate is

$$\frac{\|y_A^* - y\|}{\|y\|}.$$

By the basic assumption (3.8.2, BA) made on the algorithm  $A$ , and choosing  $x_A$  optimally, we have

$$\begin{aligned} f_A(x^*) &= f(x_A^*), \\ \frac{\|x_A^* - x^*\|}{\|x^*\|} &= (\text{cond } A)(x^*) \text{eps}. \end{aligned} \quad (3.9.2)$$

Let  $y^* = f(x^*)$ . Using the triangle inequality, we have

$$\frac{\|y_A^* - y\|}{\|y\|} \leq \frac{\|y_A^* - y^*\|}{\|y\|} + \frac{\|y^* - y\|}{\|y\|} \approx \frac{\|y_A^* - y^*\|}{\|y^*\|} + \frac{\|y^* - y\|}{\|y^*\|}.$$

We supposed  $\|y\| \approx \|y^*\|$ . By virtue of (3.9.2) we now have for the first term on the right,

$$\begin{aligned} \frac{\|y_A^* - y^*\|}{\|y^*\|} &= \frac{\|f_A(x^*) - f(x^*)\|}{\|f(x^*)\|} = \frac{\|f(x_A^*) - f(x^*)\|}{\|f(x^*)\|} \leq \\ &\leq (\text{cond } f)(x^*) \frac{\|x_A^* - x^*\|}{\|x^*\|} = (\text{cond } f)(x^*) (\text{cond } A)(x^*) \text{eps}, \end{aligned}$$

and for the second

$$\frac{\|y^* - y\|}{\|y\|} = \frac{\|f(x^*) - f(x)\|}{\|f(x)\|} \leq (\text{cond } f)(x) \frac{\|x^* - x\|}{\|x\|} = (\text{cond } f)(x)\varepsilon.$$

Assuming finally that  $(\text{cond } f)(x^*) \approx (\text{cond } f)(x)$ , we get

$$\frac{\|y_A^* - y\|}{\|y\|} \leq (\text{cond } f)(x)[\varepsilon + (\text{cond } A)(x^*) \text{eps}]. \quad (3.9.3)$$

Interpretation: The data error and eps contribute together towards the total error. Both are amplified by the condition of the problem, but the latter is further amplified by the condition of the algorithm.

## 3.10 Ill-Conditioned Problems and Ill-Posed Problems

If the condition number of a problem is large  $((\text{cond } f)(x) \gg 1)$ , then even for small (relative) errors, huge errors in output data could be expected. Such problems are called *ill-conditioned problems*. It is not possible to draw a clear separation line between well-conditioned and ill-conditioned problems. The classification depends on precision specifications. If we wish

$$\frac{\|y^* - y_A^*\|}{\|y\|} < \tau$$

and in (3.9.3)  $(\text{cond } f)(x)\varepsilon \geq \tau$ , then the problem is surely ill-conditioned.

It is important to choose a reasonable boundary off error, since otherwise, even if we increase the iteration number, we can not achieve the desired accuracy.

If the result of a mathematical problem depends discontinuously on continuous input data, then it is impossible to obtain an accurate numerical solution in a neighborhood of the discontinuity. In such cases the result is significantly perturbed, even if the input data are accurate and the computation is performed using multiple precision. These problems are called *ill-posed problems*. An ill-posed problem can appear if, for example, an integer result is computed from real input data (which vary continuously). As examples we can cite the number of real zeros of a polynomial and the rank of a matrix.

**Example 3.10.1 (The number of real zeros of a polynomial).** The equation

$$P_3(x, c_0) = c_0 + x - 2x^2 + x^3$$

can have one, two or three real zeros, depending on how  $c_0$  is: strictly positive, zero or strictly negative. Therefore, if  $c_0$  is close to zero, the number of real zeros of  $P_3$  is an ill-posed problem.  $\diamond$

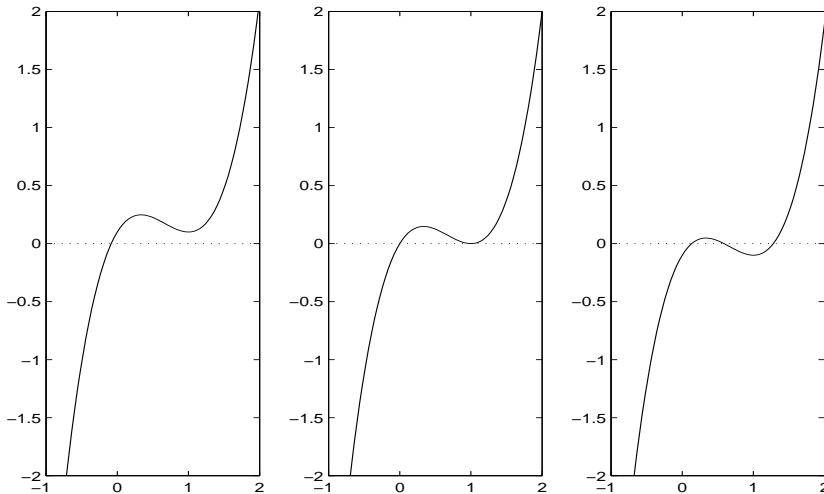


Figure 3.4: An ill-posed problem

## 3.11 Stability

### 3.11.1 Asymptotical notations

We shall introduce here basic notations and some common abuses.

For a given function  $g(n)$ ,  $\Theta(g(n))$  will denote the set of functions

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \leq n_0\}.$$

Although  $\Theta(g(n))$  is a set we write  $f(n) = \Theta(g(n))$  instead of  $f(n) \in \Theta(g(n))$ . These abuse has some advantages.  $g(n)$  will be called an *asymptotically tight bound* for  $f(n)$ .

The  $\Theta(g(n))$  definition requires that every member of it to be *asymptotically nonnegative*, that is,  $f(n) \geq 0$  for sufficiently large  $n$ .

For a given function  $g(n)$ ,  $O(g(n))$  will denote the set

$$O(g(n)) = \{f(n) : \exists c, n_0 \quad 0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0\}.$$

Also for  $f(n) \in O(g(n))$  we shall use  $f(n) = O(g(n))$ . Note that  $f(n) = \Theta(g(n))$  implies  $f(n) = O(g(n))$ , since the  $\Theta$  notation is stronger than the  $O$  notation. In set theory terms,  $\Theta(g(n)) \subseteq O(g(n))$ . One of the funny properties of the  $O$  notation is  $n = O(n^2)$ .  $g(n)$  will be called an *asymptotically upper bound* for  $f$ .

For a given function  $g(n)$ ,  $\Omega(g(n))$  is defined as the set of functions

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 \quad 0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0\}.$$

This notation provide an *asymptotically lower bound*. The definitions of asymptotic notations imply immediately:

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)).$$

The functions  $f$  and  $g : \mathbb{N} \rightarrow \mathbb{R}$  are *asymptotically equivalent* (notation  $\sim$ ) if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1.$$

The extension of asymptotic notations to real numbers is obvious. For example,  $f(t) = O(g(t))$  means that there exists a positive constant  $C$  such that for all  $t$  sufficiently close to an understood limit (e.g.,  $t \rightarrow 0$  or  $t \rightarrow \infty$ ),

$$|f(t)| \leq Cg(t). \quad (3.11.1)$$

### 3.11.2 Accuracy and stability

In this section, we think a *problem* as a map  $f : X \rightarrow Y$ , where  $X$  and  $Y$  are normed linear spaces (for our purpose finite dimensional spaces are sufficient). We are interested in the problem behavior at a particular point  $x \in X$  (the behavior may vary from one point to another). A combination of a problem  $f$  with prescribed input data  $x$  might be called a *problem instance*, but it is usually, though occasionally, confusing to use the term problem for both notions.

Since the complex numbers are represented as a pair of floating-point numbers, the axiom (3.4.4) also holds for complex numbers, except that for  $\otimes$  and  $\oslash$   $\text{eps}$  must be enlarged by a factor of the order  $2^{3/2}$  and  $2^{5/2}$ , respectively.

An algorithm can be viewed as another map  $f_A : X \rightarrow Y$ , where  $X$  and  $Y$  are as above. Let us consider a problem  $f$ , a computer whose floating-point number system satisfies (3.4.4), but not necessarily (3.4.3), an algorithm  $f_A$  for  $f$ , an implementation of this algorithm as a computer program,  $A$ , all fixed. Given input data  $x \in X$ , we round it to a floating point number and then supply it to the program. The result is a collection of floating-point numbers forming a vector from  $Y$  (since the algorithm was designed to solve  $f$ ). Let this computer result be called  $f_A(x)$ .

Except in trivial cases,  $f_A$  cannot be continuous. One might say that an algorithm  $f_A$  for the problem  $f$  is *accurate*, if for each  $x \in X$ , his relative error satisfies

$$\frac{\|f_A(x) - f(x)\|}{\|f(x)\|} = O(\text{eps}). \quad (3.11.2)$$

If the problem  $f$  is ill-conditioned, the goal of accuracy, as defined by (3.11.2) is unreasonable ambitious. Rounding errors on input data are unavoidable on a digital computer, and even if all the subsequent computation could be carried out perfectly, this perturbation alone might lead to a significant change in the result. Instead of aiming at accuracy in all cases, it is the most appropriate to search for general stability. We say that an algorithm  $f_A$  for a problem  $f$  is *stable* if for each  $x \in X$

$$\frac{\|f_A(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(\text{eps}), \quad (3.11.3)$$

for some  $\tilde{x}$  with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\text{eps}). \quad (3.11.4)$$

In words,

*A stable algorithm gives nearly the right answer to nearly the right question.*

Many algorithms of Numerical Linear Algebra satisfy a condition that is both stronger and simpler than stability. We say that an algorithm  $f_A$  for the problem  $f$  is *backward stable* if

$$\forall x \in X \exists \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O(\text{eps}) \text{ such that } f_A(x) = f(\tilde{x}). \quad (3.11.5)$$

This is a tightening of the definition of stability in that the  $O(\text{eps})$  in (3.11.3) was replaced by zero. In words

*A backward stable algorithm gives exactly the right answer to nearly the right question.*

**Remark 3.11.1.** The notation

$$\|\text{computed quantity}\| = O(\text{eps}) \quad (3.11.6)$$

has the following meaning:

- $\|\text{computed quantity}\|$  represents the norm of some number or collection of numbers determined by an algorithm  $f_A$  for a problem  $f$ , depending on both the input data  $x \in X$  for  $f$  and  $\text{eps}$ . An example is the relative error.
- The implicit limit process is  $\text{eps} \rightarrow 0$  (i.e.  $\text{eps}$  corresponds to  $t$  in (3.11.1)).
- The  $O$  applies uniformly to all data  $x \in X$ . This uniformity is default in the statement of stability results.
- In any particular machine arithmetic,  $\text{eps}$  is a fixed quantity. Speaking of the limit  $\text{eps} \rightarrow 0$ , we are considering an idealization of a computer or a family of computers. Equation (3.11.6) means that if we were to run the algorithm in question on computers satisfying (3.4.3) and (3.4.4) for a sequence of values of  $\text{eps}$  decreasing to zero, then  $\|\text{computed quantity}\|$  would be guaranteed to decrease in proportion to  $\text{eps}$  or faster. These ideal computers are required to satisfy (3.4.3) and (3.4.4), but nothing else.
- The default constant in  $O$  can depend also on the size of the argument (e.g. the solution of a nonsingular system  $Ax = b$  on  $A$  and  $b$  size). Generally, in practice, the error growing due to the size of argument is slow enough, but there are situations with factors like  $2^m$ ; they make such bounds useless for practice.  $\diamond$

Due to the equivalence of norms on finite dimensional linear spaces, for problems  $f$  and algorithms  $f_A$  defined on such spaces, the properties of accuracy, stability and backward stability all hold or fail to hold independently of the choice of norms in  $X$  and  $Y$ .

### 3.11.3 Backward Error Analysis

Backward stability and well-conditioning imply accuracy in relative sense.

**Theorem 3.11.2.** *Suppose a backward stable algorithm  $f_A$  is applied to solve a problem  $f : X \rightarrow Y$  with condition number  $(\text{cond } f)(x)$  on a computer satisfying the axioms (3.4.3) and (3.4.4). Then the relative error satisfies*

$$\frac{\|f_A(x) - f(x)\|}{\|f(x)\|} = O((\text{cond } f)(x) \text{eps}). \quad (3.11.7)$$

*Proof.* By the definition (3.11.5) of backward stability we have  $f_A(x) = f(\tilde{x})$  for some  $\tilde{x} \in X$  satisfying

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\text{eps}).$$

By the definition (3.7.5) and (3.7.6) of  $(\text{cond } f)(x)$ , this implies

$$\frac{\|f_A(x) - f(x)\|}{\|f(x)\|} \leq ((\text{cond } f)(x) + o(1)) \frac{\|\tilde{x} - x\|}{\|x\|}, \quad (3.11.8)$$

where  $o(1)$  denotes a quantity which converges to zero as  $\text{eps} \rightarrow 0$ . Combining these bounds gives (3.11.7).  $\square$

The process just carried out in proving Theorem 3.11.2 is known as *backward error analysis*. We obtained an accuracy estimate by two steps. One step is to investigate the condition of the problem. The other is to investigate the stability of the algorithm. By Theorem 3.11.2, if the algorithm is backward stable, then the final accuracy reflects that condition number.

There exists also a *forward error analysis*. Here, the rounding error introduced at each step of the calculation are estimated, and somehow, a total is maintained of how they compound from step to step (section 3.3).

Experience has shown that for the most of the algorithms of numerical linear algebra, forward error analysis is harder to carry out than the backward error analysis. The best algorithms of linear algebra do no more, in general, than to compute exact solutions for slightly perturbed data. Backward error analysis is a method of reasoning fitted neatly to this backward reality.

## 3.12 Applications

### 3.12.1 Inverting the hyperbolic cosine

The aim of this application is to compute the inverse of  $\cosh$  in floating point arithmetic when  $x \gg 1$ . From

$$x = \cosh(y) = \frac{e^y + e^{-y}}{2},$$

it follows, for  $x > 1$

$$y = \text{arccosh } x = -\ln \left( x - \sqrt{x^2 - 1} \right).$$

This way of computing  $\text{arccosh}$  suffers of catastrophic cancelation. Because of

$$\sqrt{x^2 - 1} = x\sqrt{1 - \frac{1}{x^2}} \approx x \left(1 - \frac{1}{2x^2} + \dots\right),$$

we obtain

$$y = -\ln \left(x - \sqrt{x^2 - 1}\right) \approx -\ln \frac{1}{2x} = \ln 2x.$$

The first difficulty occurs when we try to evaluate  $x^2$ , for  $x$  large ; this could cause overflow. The overflow is unnecessary because the argument we are trying to compute is on scale. If  $x$  is large, but not so large that  $x^2$  o cause overflows,  $\text{fl}(x^2 - 1) = \text{fl}(x^2)$ . We have a small error in subtraction and then in square root. Cancelation occurs when we try to compute the argument of natural logarithm.

How we can avoid the difficulty? A little calculation shows that

$$-\ln \left(x - \sqrt{x^2 - 1}\right) = \ln \frac{1}{x - \sqrt{x^2 - 1}} = \ln \left(x + \sqrt{x^2 - 1}\right),$$

a form that avoids cancelation. A better way of handling the rest of the argument is

$$\sqrt{x^2 - 1} = x\sqrt{1 - \left(\frac{1}{x}\right)^2}.$$

Notice that we wrote  $(1/x)^2$  instead of  $1/x^2$ , because we convert an overflow when forming  $x^2$  into a less harmful underflow. Finally, we see that the expression

$$y = \ln \left(x + x\sqrt{1 - \left(\frac{1}{x}\right)^2}\right)$$

avoids all the difficulties of the original expression for  $\text{arccosh}(x) = \cosh^{-1}(x)$ . Indeed, it is clear that for large  $x$ , evaluation of this expression in floating point arithmetic will lead to an approximation of  $\ln(2x)$ , as it should.

The next MATLAB example inverts  $\cosh y$  for  $y = 15, 16, \dots, 20$ .

```
>> y=15:20;
>> x=cosh(y);
>> -log(x-sqrt(x.^2-1))
ans =
    Columns 1 through 6
    14.9998  15.9986  17.0102  18.0218    Inf      Inf
>> log(x+x.*sqrt(1-(1./x).^2))
ans =
    15      16      17      18      19      20
```

### 3.12.2 Conditioning of a root of a polynomial equation

Consider the algebraic equation:

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0, \quad a_0 \neq 0 \quad (3.12.1)$$

and one of its simple root:

$$p(\xi) = 0, \quad p'(\xi) \neq 0.$$

Our problem is to find  $\xi$ , given  $p$ . The input data consists of the vector of  $p$ 's coefficients

$$a = [a_0, a_1, \dots, a_{n-1}]^T \in \mathbb{R}^n$$

and the result is  $\xi$ , a real or complex number. So, our problem is

$$\xi : \mathbb{R}^n \rightarrow \mathbb{C}, \quad \xi = \xi(a_0, a_1, \dots, a_{n-1})$$

What is the condition of  $\xi$ ?

We define

$$\gamma_\nu = (\text{cond}_\nu \xi)(a) = \left| \frac{a_\nu \frac{\partial \xi}{\partial a_\nu}}{\xi} \right|, \quad \nu = 0, 1, \dots, n-1 \quad (3.12.2)$$

Then we take a convenient norm of the vector  $\gamma = [\gamma_0, \dots, \gamma_{n-1}]^T$ , for example

$$\|\gamma\|_1 := \sum_{\nu=0}^{n-1} |\gamma_\nu|$$

to define

$$(\text{cond } \xi)(a) = \sum_{\nu=0}^{n-1} (\text{cond}_\nu \xi)(a) \quad (3.12.3)$$

To find the partial derivatives of  $\xi$  with respect to  $a_\nu$ , we start from the identity

$$\begin{aligned} & [\xi(a_0, a_1, \dots, a_{n-1})]^n + a_{n-1}[\xi(a_0, a_1, \dots, a_{n-1})]^{n-1} + \cdots + \\ & + a_\nu[\xi(a_0, a_1, \dots, a_{n-1})]^\nu + \cdots + a_0 = 0. \end{aligned}$$

Differentiating it with respect to  $a_\nu$  we obtain

$$\begin{aligned} & n[\xi(a_0, a_1, \dots, a_{n-1})]^{n-1} \frac{\partial \xi}{\partial a_\nu} + a_{n-1}(n-1)[\xi(a_0, a_1, \dots, a_{n-1})]^{n-2} \frac{\partial \xi}{\partial a_\nu} + \cdots + \\ & + a_\nu \nu [\xi(a_0, a_1, \dots, a_{n-1})]^{\nu-1} \frac{\partial \xi}{\partial a_\nu} + \cdots + a_1 \frac{\partial \xi}{\partial a_\nu} + [\xi(a_0, a_1, \dots, a_{n-1})]^\nu \equiv 0, \end{aligned}$$

where the last term comes from the differentiating the first factor of  $a_\nu \xi^\nu$ . The last identity can be written as

$$p'(\xi) \frac{\partial \xi}{\partial a_\nu} + \xi^\nu = 0$$

Since  $p'(\xi) \neq 0$ , we solve for  $\frac{\partial \xi}{\partial a_\nu}$  and substitute the result in (3.12.2) and (3.12.3) to obtain

$$(\text{cond } \xi)(a) = \frac{1}{|\xi p'(\xi)|} \sum_{\nu=0}^{n-1} |a_\nu| |\xi|^\nu \quad (3.12.4)$$

We illustrate (3.12.4) by considering a famous example, due to Wilkinson [104]

$$p(x) = \prod_{\nu=1}^n (x - \nu) = x^n + a_{n-1}x^{n-1} + \cdots + a_0. \quad (3.12.5)$$

If we take  $\xi_\mu = \mu$ ,  $\mu = 1, 2, \dots, n$ , it can be shown that [32]

$$\begin{aligned} \min_\mu \text{cond } \xi_\mu &= \text{cond } \xi_1 \sim n^2 \text{ when } n \rightarrow \infty \\ \max_\mu \text{cond } \xi_\mu &\sim \frac{1}{(2-\sqrt{2})\pi n} \left( \frac{\sqrt{2}+1}{\sqrt{2}-1} \right)^n \text{ when } n \rightarrow \infty. \end{aligned}$$

The worst conditioned root is  $\xi_{\mu_0}$  with  $\mu_0$  the integer closest to  $n/\sqrt{2}$  when  $n$  is large. This condition number grows exponentially fast in  $n$ . For example when  $n = 20$ , then  $\mu_0 = 14$  and  $\text{cond } \xi_{\mu_0} = 0.540 \times 10^{14}$ .

The moral of this example is that the roots of an algebraic equation written in form (3.12.1) can be extremely sensitive to small changes in the coefficients (and thus, severe ill-conditioned). Hence, avoid to express polynomials as sum of powers as in (3.12.1) and (3.12.5). This is also true for characteristic polynomials of matrices. It is better to compute their roots as solutions of an eigenvalue problem, which is better conditioned.

Now, we apply formula (3.12.4) to implement a MATLAB function that computes the condition numbers of the roots of a polynomial equation (see MATLAB Source 3.3).

---

### MATLAB Source 3.3 Conditioning of the roots of an algebraic equation

---

```
function nc=condpol(p, xi)
%CONDPOL - condition of the roots of an algebraic equation
%call NC=CONDPOL(P, XI)

if nargin<2
    xi=roots(p);
end
n=length(p)-1;
dp=polyder(p); %derivative;
nc=1./ (abs(xi.*polyval(dp,xi))).*(polyval(abs(p(2:end)),abs(xi)));
```

---

Let us use this function to the Wilkinson example. The results are given in ascending order of condition number, and they are in accordance to the theory.

```
>> format short g
>> xi=1:20;
>> nc=condpol(poly(xi),xi);
>> [ncs,i]=sort(nc');
```

```
>> [ncs,i]
ans =
        420          1
        43890         2
    2.0189e+006         3
    5.1483e+007         4
    8.2373e+008         5
    8.9237e+009         6
    6.8839e+010         7
    1.378e+011        20
    3.9156e+011         8
    1.3781e+012        19
    1.6816e+012         9
    5.5566e+012        10
    6.3797e+012        18
    1.4194e+013        11
    1.8121e+013        17
    2.8523e+013        12
    3.5438e+013        16
    4.4307e+013        13
    5.0243e+013        15
    5.401e+013        14
```

We close the section with a graphical study of influence of small perturbations on Wilkinson's polynomial. The next MATLAB function perturbs the coefficients with small normal random numbers with mean 0 and variance  $10^{-10}$  and then plots the theoretical roots and the perturbed roots.

```
function Wilkinson(n)
%perturbations for Wilkinson example

p=poly(1:n);
h=plot([1:n],zeros(1,n),'.');
set(h,'Markersize',15);
hold on
for k=1:1000
    r=randn(1,n+1);
    pr=p.* (1+1e-10*r);
    z=roots(pr);
    h2=plot(z,'k.');
    set(h2,'Markersize', 4)
end
axis equal
```

Figure 3.5 shows the results of  $\text{Wilkinson}(20)$ .

This treatment of the topic of conditioning of algebraic equations follows the Gautschi's book [33]. For multiple roots, see [91]. The graphical experiment is proposed in [96].

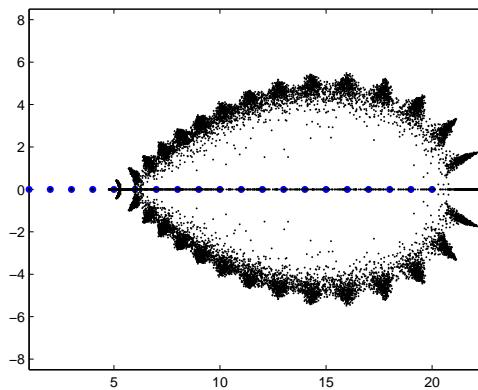


Figure 3.5: Results generated by execution of Wilkinson (20)

## Problems

**Problem 3.1.** Code MATLAB functions to compute  $\sin x$  and  $\cos x$  using Taylor formula:

$$\begin{aligned}\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \dots \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!} + \dots\end{aligned}$$

The following facts are well known (see any Calculus course):

- absolute value of the error is less than the absolute value of the first neglected term;
- the convergence radius is  $R = \infty$ .

What is happened for  $x = 10\pi$  (and in general for  $x = 2k\pi$ , if  $k$  is large)? Explain the phenomenon and find a remedy?

**Problem 3.2.** Let

$$E_n = \int_0^1 x^n e^{x-1} dx.$$

We see that  $E_1 = 1/e$  and  $E_n = 1 - nE_{n-1}$ ,  $n = 2, 3, \dots$ . It is possible to show that

$$0 < E_n < \frac{1}{n+1}$$

and if  $E_1 = c$ , then

$$\lim_{n \rightarrow \infty} E_n = \begin{cases} 0, & \text{for } c = 1/e \\ \infty, & \text{otherwise.} \end{cases}$$

Explain the phenomenon, find a remedy, and compute  $e$  with a precision of  $\text{eps}$ .

**Problem 3.3.** Consider the MATLAB functions in Section 3.12.2. Study experimentally the condition of the problem of finding the roots of the polynomial equation

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n = 0, \quad (3.12.6)$$

$a_k = 2^{-k}$ . Take  $n = 20$  for practical tests. Modify the graphical experiment if the perturbation obeys the uniform law.

**Problem 3.4.** What is the index of the largest Fibonacci number that could be represented exactly in MATLAB in double precision? What is the index of the largest Fibonacci number that could be represented in MATLAB in double precision without overflow?

**Problem 3.5.** Let  $F$  be the set of all IEEE floating-point numbers, excepting `NaN` and `Inf`, with biased exponent `7ff` (in hexadecimal) and denormalized numbers, with biased exponent `000` (in hexadecimal).

- (a) What is the cardinal of  $F$ ?
- (b) What proportion of  $F$  elements lies within interval  $[1, 2)$ ?
- (c) What proportion of  $F$  elements lies within interval  $[1/64, 1/32)$ ?
- (d) Find using a random selection the proportion of  $F$  elements satisfying the MATLAB logical relation

$$x * (1/x) == 1$$

**Problem 3.6.** What familiar real numbers are approximated by floating-point numbers, for which, by using `format hex`, the following values are displayed:

```
4059000000000000
3f847ae147ae147b
3fe921fb54442d18
```

**Problem 3.7.** Explain the results displayed by

```
t = 0.1
n = 1:10
e = n/10 - n*t
```

**Problem 3.8.** What each of the following program does? How many lines does each of them produce? What are the last two values of  $x$  displayed?

```
x=1; while 1+x>1, x=x/2, pause (.02), end
x=1; while x+x>x, x=2*x, pause (.02), end
x=1; while x+x>x, x=x/2, pause (.02), end
```

**Problem 3.9.** Write a MATLAB function that computes the area of a triangle, given edges, using Hero's formula:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

where  $p$  is the half-perimeter. What happens when the triangle is almost degenerated? Propose a remedy, and give an error estimation(see [37]).

**Problem 3.10.** The sample variance is defined by

$$s^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2,$$

where

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i.$$

One can compute them alternatively by formula

$$s^2 = \frac{1}{N-1} \left[ \sum_{i=1}^N x_i^2 - \frac{1}{N} \left( \sum_{i=1}^N x_i \right)^2 \right].$$

What formula is more accurate from numerical point of view? Give an example that argues the answer.

# CHAPTER 4

---

## Numerical Solution of Linear Algebraic Systems

---

There are two classes of methods for the solution of algebraic linear systems (ALS):

- *direct* or *exact* methods – they provide a solution in a finite number of steps, under the assumption all computations are carried out exactly (Cramer, Gaussian elimination, Cholesky)
- *iterative* methods – they approximates the solution by generating a sequence converging to that solution (Jacobi, Gauss-Seidel, SOR).

### 4.1 Notions of Matrix Analysis

**Definition 4.1.1.** *The  $p$ -norm of a vector  $x \in \mathbb{K}^n$  is defined by*

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p} \quad 1 \leq p < \infty.$$

*For  $p = \infty$  the norm is defined by*

$$\|x\|_\infty = \max_{i=1,n} |x_i|.$$

*The norm  $\|\cdot\|_2$  is called Euclidian norm,  $\|\cdot\|_1$  is called Minkowski norm, and  $\|\cdot\|_\infty$  is called Chebyshev norm.*

The MATLAB function `norm` computes the  $p$ -norm of a vector. It is invoked as `norm(x, p)`, with default value `p=2`. As a special case, for `p=-Inf`, the quantity  $\min_i |x_i|$  is computed. Example:

```

>> x = 1:4;
>> [norm(x,1), norm(x,2), norm(x,inf), norm(x,-inf) ]
ans =
    10.0000    5.4772    4.0000    1.0000

```

Figure 4.1 shows the pictures of unit sphere in  $\mathbb{R}^2$  for various  $p$ -norms. It was obtained via `contour` function.

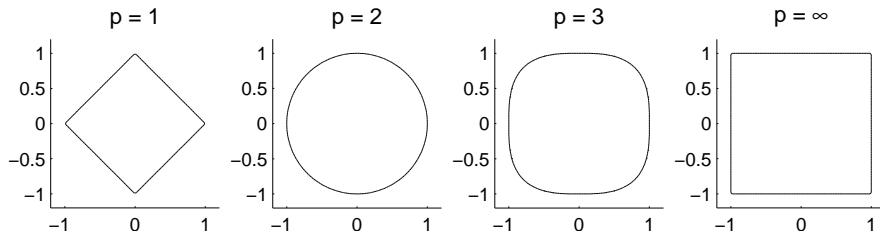


Figure 4.1: The unit sphere in  $\mathbb{R}^2$  for four  $p$ -norms

Let  $A \in \mathbb{K}^{n \times n}$ .

- The polynomial  $p(\lambda) = \det(A - \lambda I)$  – *characteristic polynomial* of  $A$ ;
- zeros of  $p$  – *eigenvalues* of  $A$ ;
- If  $\lambda$  is an eigenvalue of  $A$ , the vector  $x \neq 0$  such that  $(A - \lambda I)x = 0$  is an *eigenvector* of  $A$  corresponding to the eigenvalue  $\lambda$ ;
- $\rho(A) = \max\{|\lambda| \mid \lambda \text{ eigenvalue of } A\}$  – *spectral radius* of the matrix  $A$ .

$A^T$  – the transpose of  $A$ ;  $A^*$  the conjugate transpose (adjoint) of  $A$ .

**Definition 4.1.2.** A matrix  $A$  is called:

1. normal, if  $AA^* = A^*A$ ;
2. unitary, if  $AA^* = A^*A = I$ ;
3. hermitian, if  $A = A^*$ ;
4. orthogonal, if  $AA^T = A^TA = I$ ,  $A$  real;
5. symmetric, if  $A = A^T$ ,  $A$  real;
6. upper Hessenberg, if  $a_{ij} = 0$ , for  $i > j + 1$

**Definition 4.1.3.** A matrix norm is a map  $\|\cdot\| : \mathbb{K}^{m \times n} \rightarrow \mathbb{R}$  that for each  $A, B \in \mathbb{K}^{m \times n}$  and  $\alpha \in \mathbb{K}$  satisfy

(NMI)  $\|A\| \geq 0$ ,  $\|A\| = 0 \Leftrightarrow A = O_{m \times n}$ ;

- (NM2)  $\|\alpha A\| = |\alpha| \|A\|$ ;
- (NM3)  $\|A + B\| \leq \|A\| + \|B\|$ ;
- (NM4)  $\|AB\| \leq \|A\| \|B\|$ .

A simple way to obtain matrix norm is: given a vector norm  $\|\cdot\|$  on  $\mathbb{C}^n$ , the map  $\|\cdot\| : \mathbb{C}^{n \times n} \rightarrow \mathbb{R}$

$$\|A\| = \sup_{\substack{v \in \mathbb{C}^n \\ v \neq 0}} \frac{\|Av\|}{\|v\|} = \sup_{\substack{v \in \mathbb{C}^n \\ \|v\| \leq 1}} \|Av\| = \sup_{\substack{v \in \mathbb{C}^n \\ \|v\|=1}} \|Av\|$$

is a matrix norm called *subordinate matrix norm* (to the given vector norm) or *natural norm* (induced by the given vector norm).

**Remark 4.1.4.** A subordinate matrix norm verifies  $\|I\| = 1$ .  $\diamond$

The norms subordinate to vector norms  $\|\cdot\|_1, \|\cdot\|_2, \|\cdot\|_\infty$  are given by the following result.

**Theorem 4.1.5.** Let  $A \in \mathbb{K}^{n \times n}(\mathbb{C})$ . Then

$$\begin{aligned} \|A\|_1 &:= \sup_{v \in \mathbb{C}^n \setminus \{0\}} \frac{\|Av\|_1}{\|v\|_1} = \max_j \sum_i |a_{ij}|, \\ \|A\|_\infty &:= \sup_{v \in \mathbb{C}^n \setminus \{0\}} \frac{\|Av\|_\infty}{\|v\|_\infty} = \max_i \sum_j |a_{ij}|, \\ \|A\|_2 &:= \sup_{v \in \mathbb{C}^n \setminus \{0\}} \frac{\|Av\|_2}{\|v\|_2} = \sqrt{\rho(A^*A)} = \sqrt{\rho(AA^*)} = \|A^*\|_2. \end{aligned}$$

The norm  $\|\cdot\|_2$  is invariant to the unit transforms,

$$UU^* = I \Rightarrow \|A\|_2 = \|AU\|_2 = \|UA\|_2 = \|U^*AU\|_2.$$

If  $A$  is normal, then

$$AA^* = A^*A \Rightarrow \|A\|_2 = \rho(A).$$

*Proof.* For any vector  $v$  we have

$$\begin{aligned} \|Av\|_1 &= \sum_i \left| \sum_j a_{ij} v_j \right| \leq \sum_j |v_j| \sum_i |a_{ij}| \leq \\ &\leq \left( \max_j \sum_i |a_{ij}| \right) \|v\|_1. \end{aligned}$$

In order to show that  $\max_j \sum_i |a_{ij}|$  is actually the smallest  $\alpha$  having the property  $\|Av\|_1 \leq \alpha \|v\|_1$ ,  $\forall v \in \mathbb{C}^n$ , it is sufficient to construct a vector  $u$  (that depends on  $A$ ) such that:

$$\|Au\|_1 = \left\{ \max_j \sum_i |a_{ij}| \right\} \|u\|_1.$$

If  $j_0$  is a subscript such that

$$\max_j \sum_i |a_{ij}| = \sum_i |a_{ij_0}|,$$

then the vector  $u$  entries are  $u_i = 0$  for  $i \neq j_0$ ,  $u_{j_0} = 1$ .

Similarly

$$\|Av\|_\infty = \max_i \left| \sum_j a_{ij} v_j \right| \leq \left( \max_i \sum_j |a_{ij}| \right) \|v\|_\infty.$$

Let  $i_0$  be a subscript such that

$$\max_i \sum_j |a_{ij}| = \sum_j |a_{i_0 j}|.$$

The vector  $u$  such that  $u_j = \frac{\overline{a_{i_0 j}}}{|a_{i_0 j}|}$  for  $a_{i_0 j} \neq 0$ ,  $u_j = 1$  for  $a_{i_0 j} = 0$  verifies

$$\|Au\|_\infty = \left\{ \max_i \sum_j |a_{ij}| \right\} \|u\|_\infty.$$

Since  $AA^*$  is a Hermitian matrix, there exists an eigendecomposition  $AA^* = Q\Lambda Q^*$  of  $A$ , where  $Q$  is a unitary matrix whose columns are eigenvectors, and  $\Lambda$  is a diagonal matrix whose entries are eigenvalues of  $A$  (all of them must be real). If there exists a negative eigenvalue and  $q$  is the corresponding eigenvector, then  $0 \leq \|Aq\|_2^2 = q^* A^* A q = q^* \lambda q = \lambda \|q\|_2^2$ . So,

$$\begin{aligned} \|A\|_2 &= \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \max_{x \neq 0} \frac{(x^* A^* Ax)^{1/2}}{\|x\|_2} = \max_{x \neq 0} \frac{(x^* Q\Lambda Q^* x)^{1/2}}{\|x\|_2} \\ &= \max_{x \neq 0} \frac{((Q^* x)^* \Lambda Q^* x)^{1/2}}{\|Q^* x\|_2} = \max_{y \neq 0} \frac{(y^* \Lambda y)^{1/2}}{\|y\|_2} = \max_{y \neq 0} \sqrt{\frac{\sum \lambda_i y_i^2}{\sum y_i^2}} \\ &\leq \max_{y \neq 0} \sqrt{\lambda_{\max}} \sqrt{\frac{\sum y_i^2}{\sum y_i^2}}; \end{aligned}$$

the equality holds if  $y$  is a conveniently chosen column of a unit matrix.

Let us prove now that  $\rho(A^* A) = \rho(AA^*)$ . If  $\rho(A^* A) > 0$  there exists a  $p$  such that  $p \neq 0$ ,  $A^* Ap = \rho(A^* A)p$  and  $Ap \neq 0$  ( $\rho(A^* A) > 0$ ). Since  $Ap \neq 0$  and  $AA^*(Ap) = \rho(A^* A)Ap$  it follows that  $0 < \rho(A^* A) \leq \rho(AA^*)$ , and therefore  $\rho(AA^*) = \rho(A^* A)$  (because  $(A^*)^* = A$ ).

A). If  $\rho(A^*A) = 0$ , then  $\rho(AA^*) = 0$ . Hence, in all cases  $\|A\|_2^2 = \rho(A^*A) = \rho(AA^*) = \|A^*\|_2^2$ .

The invariance of  $\|\cdot\|_2$  norm to unitary transforms is a translation of relations:

$$\rho(A^*A) = \rho(U^*A^*AU) = \rho(A^*U^*UA) = \rho(U^*A^*UU^*AU).$$

Finally, if  $A$  is normal, there exists a matrix  $U$  such that

$$U^*AU = \text{diag}(\lambda_i(A)) \stackrel{\text{def}}{=} \Lambda.$$

In this case

$$A^*A = (U\Lambda U^*)^*U\Lambda U = UD^*\Lambda U^*,$$

which shows us that

$$\rho(A^*A) = \rho(\Lambda^*\Lambda) = \max_i |\lambda_i(A)|^2 = (\rho(A))^2.$$

□

**Remark 4.1.6.** 1) If  $U$  is hermitian or symmetric (therefore normal),

$$\|A\|_2 = \rho(A).$$

2) If  $U$  is unitary or orthogonal (therefore normal),

$$\|A\|_2 = \sqrt{\rho(A^*A)} = \sqrt{\rho(I)} = 1.$$

3) Theorem 4.1.5 states that normal matrices and  $\|\cdot\|_2$  norm verify

$$\|A\|_2 = \rho(A).$$

◇

An important matrix norm, which is not a subordinate matrix norm is the *Frobenius norm*:

$$\|A\|_E = \left\{ \sum_i \sum_j |a_{ij}|^2 \right\}^{1/2} = \{\text{tr}(A^*A)\}^{1/2}$$

It is not a subordinate norm, since  $\|I\|_E = \sqrt{n}$ .

**Theorem 4.1.7.** (1) Let  $A$  be an arbitrary square matrix and  $\|\cdot\|$  a certain matrix norm (subordinate or not). Then

$$\rho(A) \leq \|A\|. \quad (4.1.1)$$

(2) Given a matrix  $A$  and a number  $\varepsilon > 0$ , there exists a subordinate matrix norm such that

$$\|A\| \leq \rho(A) + \varepsilon. \quad (4.1.2)$$

*Proof.* (1) Let  $p$  be a vector verifying  $p \neq 0$ ,  $Ap = \lambda p$ ,  $|\lambda| = \rho(A)$  and  $q$  a vector such that  $pq^T \neq 0$ . Since

$$\rho(A)\|pq^T\| = \|\lambda pq^T\| = \|Apq^T\| \leq \|A\|\|pq^T\|,$$

(4.1.1) results immediately.

(2) Let  $A$  be a given matrix. There exists an invertible matrix  $U$  such that  $U^{-1}AU$  is upper-triangular (in fact,  $U$  is unitary)

$$U^{-1}AU = \begin{pmatrix} \lambda_1 & t_{12} & t_{13} & \dots & t_{1,n} \\ & \lambda_2 & t_{23} & \dots & t_{2,n} \\ & & \ddots & & \vdots \\ & & & \lambda_{n-1} & t_{n-1,n} \\ & & & & \lambda_n \end{pmatrix};$$

the scalars  $\lambda_i$  are the eigenvalues of  $A$ . To any scalar  $\delta \neq 0$  we associate a matrix

$$D_\delta = \text{diag}(1, \delta, \delta^2, \dots, \delta^{n-1}),$$

such that

$$(UD_\delta)^{-1}A(UD_\delta) = \begin{pmatrix} \lambda_1 & \delta t_{12} & \delta^2 t_{13} & \dots & \delta^{n-1} t_{1n} \\ & \lambda_2 & \delta t_{23} & \dots & \delta^{n-2} t_{2n} \\ & & \ddots & & \vdots \\ & & & \lambda_{n-1} & \delta t_{n-1,n} \\ & & & & \lambda_n \end{pmatrix}.$$

Given  $\varepsilon > 0$ , we take a  $\delta$  fixed, such that

$$\sum_{j=i+1}^n |\delta^{j-i} t_{ij}| \leq \varepsilon, \quad 1 \leq i \leq n-1.$$

Then, the map

$$\|\cdot\| : B \in \mathbb{K}^{n \times n} \rightarrow \|B\| = \|(UD_\delta)^{-1}B(UD_\delta)\|_\infty, \quad (4.1.3)$$

which depends on  $A$  and  $\varepsilon$  solve the problem. Indeed,

$$\|A\| \leq \rho(A) + \varepsilon$$

and according to the choice of  $\delta$  and the definition of  $\|\cdot\|_\infty$  ( $\|c_{ij}\|_\infty = \max_i \sum_j |c_{ij}|$ ) norm, the norm given by 4.1.3 is a matrix norm subordinated to the vector norm

$$v \in \mathbb{K}^n \rightarrow \|(UD_\delta)^{-1}v\|_\infty.$$

□

**Theorem 4.1.8.** *Let  $B$  a square matrix. The following statements are equivalent:*

$$(I) \lim_{k \rightarrow \infty} B^k = 0;$$

- (2)  $\lim_{k \rightarrow \infty} B^k v = 0, \forall v \in \mathbb{K}^n;$   
(3)  $\rho(B) < 1;$   
(4) There exists a subordinate matrix norm such that  $\|B\| < 1.$

*Proof.* (1)  $\Rightarrow$  (2)

$$\|B^k v\| \leq \|B^k\| \|v\| \Rightarrow \lim_{k \rightarrow \infty} B^k v = 0$$

(2)  $\Rightarrow$  (3) If  $\rho(B) \geq 1$ , we can find  $p$  such that  $p \neq 0, B_p = \lambda_p, |\lambda| \geq 1$ . Then the vector sequence  $(B^k p)_{k \in \mathbb{N}}$  could not converge to 0.

(3)  $\Rightarrow$  (4)  $\rho(B) < 1 \Rightarrow \exists \|\cdot\|$  such that  $\|B\| \leq \rho(B) + \varepsilon, \forall \varepsilon > 0$  hence  $\|B\| < 1.$

(4)  $\Rightarrow$  (1) It is sufficient to apply the inequality  $\|B^k\| \leq \|B\|^k$ .  $\square$

For matrices, the norm `norm` function is invoked as `norm(A, p)`, where  $A$  is a matrix, and  $p=1, 2, \text{inf}$  for a  $p$ -norm and  $p='fro'$  for the Frobenius norm. Example:

```
>> A=[1:3;4:6;7:9]
A =
    1     2     3
    4     5     6
    7     8     9
>> [norm(A,1) norm(A,2) norm(A,inf) norm(A,'fro')]
ans =
    18.0000    16.8481    24.0000    16.8819
```

When the computation of the 2-norm of a matrix is two expensive (it requires the eigenvalues of the matrix, the call `normest(A, tol)` can be used to obtain an estimation based on power method. (See Chapter 8). The default is `tol=1e-6`.

## 4.2 Condition of a linear system

We are interested in the conditioning of the problem: given the matrix  $A \in \mathbb{K}^{n \times n}$  and the vector  $b \in \mathbb{K}^{n \times 1}$ , solve the system

$$Ax = b.$$

Let the system (this example is due to Wilson)

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix},$$

having the solution  $(1, 1, 1, 1)^T$  and we consider the perturbed system where the right-hand side is slightly modified, the system matrix remaining unchanged

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 + \delta x_1 \\ x_2 + \delta x_2 \\ x_3 + \delta x_4 \\ x_4 + \delta x_4 \end{pmatrix} = \begin{pmatrix} 32.1 \\ 22.9 \\ 33.1 \\ 30.9 \end{pmatrix},$$

having the solution  $(9.2, -12.6, 4.5, -1.1)^T$ . In other words a 1/200 error in input data causes 10/1 relative error on result, hence an approx. 2000 times growing of the relative error!

Let now the system with the perturbed matrix

$$\begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix} \begin{pmatrix} x_1 + \Delta x_1 \\ x_2 + \Delta x_2 \\ x_3 + \Delta x_4 \\ x_4 + \Delta x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix},$$

having the solution  $(-81, 137, -34, 22)^T$ . Again, a small variation on input data (here, matrix elements) modifies dramatically the output result. The matrix has a “good” shape, is symmetric, its determinant is equal to 1, and its inverse is

$$\begin{pmatrix} 25 & -41 & 10 & -6 \\ -41 & 68 & -17 & 10 \\ 10 & -17 & 5 & -3 \\ -6 & 10 & -3 & 2 \end{pmatrix},$$

which is also “nice”.

Let us now consider the system parameterized by  $t$

$$(A + t\Delta A)x(t) = b + t\Delta b, \quad x(0) = x.$$

$A$  being nonsingular, the function  $x$  is differentiable at  $t = 0$ :

$$\dot{x}(0) = A^{-1}(\Delta b - \Delta Ax).$$

The Taylor expansion of  $x(t)$  is given by

$$x(t) = x + t\dot{x}(0) + O(t^2).$$

It thus follows that the absolute error can be estimated using

$$\begin{aligned} \|\Delta x(t)\| &= \|x(t) - x\| \leq |t| \|x'(0)\| + O(t^2) \\ &\leq |t| \|A^{-1}\| (\|\Delta b\| + \|\Delta A\| \|x\|) + O(t^2) \end{aligned}$$

and (due to  $\|b\| \leq \|A\| \|x\|$ ) we get for the relative error

$$\begin{aligned} \frac{\|\Delta x(t)\|}{\|x\|} &\leq |t| \|A^{-1}\| \left( \frac{\|\Delta b\|}{\|x\|} + \|\Delta A\| \right) + O(t^2) \\ &\leq \|A\| \|A^{-1}\| |t| \left( \frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|} \right) + O(t^2). \end{aligned} \tag{4.2.1}$$

By introducing the notations

$$\rho_A(t) := |t| \frac{\|\Delta A\|}{\|A\|}, \quad \rho_b(t) := |t| \frac{\|\Delta b\|}{\|b\|}$$

for the relative errors in  $A$  and  $b$ , the relative error estimate can be written as

$$\frac{\|\Delta x(t)\|}{\|x\|} \leq \|A\| \|A^{-1}\| (\rho_A + \rho_b) + O(t^2). \tag{4.2.2}$$

**Definition 4.2.1.** If  $A$  is nonsingular, the number

$$\text{cond}(A) = \|A\| \|A^{-1}\| \quad (4.2.3)$$

is called the condition number of the matrix  $A$ .

The relation (4.2.2) can be rewritten as

$$\frac{\|\Delta x(t)\|}{\|x\|} \leq \text{cond}(A) (\rho_A + \rho_b) + O(t^2). \quad (4.2.4)$$

MATLAB has several functions for the computation or estimation of condition number.

- `cond(A, p)`, where  $p=1, 2, \inf, 'fro'$  are supported. The default is  $p=2$ . For  $p=2$  one uses `svd`, and for  $p=1$  and  $\infty$ , `inv`.
- `condest(A)` estimates  $\text{cond}_1 A$ . It uses `lu` and an algorithm due to Higham and Tisseur [45]. Suitable for large sparse matrices.
- `rcond(A)` estimates  $1/\text{cond}_1 A$ . It uses `lu(A)` and an algorithm implemented in LINPACK and LAPACK.

**Example 4.2.2 (Ill-conditioned matrix).** Consider the  $n$ -th order *Hilbert*<sup>1</sup> matrix,  $H_n = (h_{ij})$ , given by

$$h_{ij} = \frac{1}{i+j-1}, \quad i, j = \overline{1, n}.$$

This is a symmetric positive definite matrix, so it is nonsingular. For various values of  $n$  one gets in the Euclidean norm

$n$	10	20	40
$\text{cond}_2(H_n)$	$1.6 \cdot 10^{13}$	$2.45 \cdot 10^{28}$	$7.65 \cdot 10^{58}$

A system of order  $n = 10$ , for example, cannot be solved with any reliability in single precision on a 14-decimal computer. Double precision will be “exhausted” by the time we reach  $n = 20$ . The Hilbert matrix is thus a prototype of an ill-conditioned matrix. From a

David Hilbert (1862-1943) was the most prominent member of the Göttingen school of Mathematics. Hilbert's fundamental contributions to almost all parts of mathematics — algebra, number theory, geometry, integral equations, calculus of variations, and foundations — and in particular the 23 now famous problems he proposed in 1900 at the International Congress of Mathematicians in Paris, gave a new impetus, and new directions, to the 20th-century mathematics.



result of G. Szegő<sup>2</sup> it can be seen that

$$\text{cond}_2(H_n) \sim \frac{(\sqrt{2} + 1)^{4n+4}}{2^{15/4}\sqrt{\pi n}}. \quad \diamond$$

**Example 4.2.3 (Ill-conditioned matrix).** *Vandermonde matrices* are of the form

$$V_n = \begin{bmatrix} 1 & 1 & \dots & 1 \\ t_1 & t_2 & \dots & t_n \\ \vdots & \vdots & \ddots & \vdots \\ t_1^{n-1} & t_2^{n-1} & \dots & t_n^{n-1} \end{bmatrix},$$

where  $t_i$  are real parameters. If  $t_i$ 's are equally spaced in [-1,1], then it holds

$$\text{cond}_\infty(V_n) \sim \frac{1}{\pi} e^{-\frac{\pi}{4}} e^{n(\frac{\pi}{4} + \frac{1}{2} \ln 2)}.$$

For  $t_i = \frac{1}{i}$ ,  $i = \overline{1, n}$

$$\text{cond}_\infty(V_n) > n^{n+1}. \quad \diamond$$

Let us check practically the conditioning of the matrices in Examples 4.2.2 and 4.2.3. For Hilbert matrix we used the sequence (file `testcondhilb.m`):

```
fprintf(' n    cond_2           est. cond      theoretical\n')
for n=[10:15,20,40]
    H=hilb(n);
    et=(sqrt(2)+1)^(4*n+4)/(2^(14/4)*sqrt(pi*n));
    x=[n, norm(H)*norm(invhilb(n)), condest(H), et];
    fprintf('%d %g %g %g\n',x)
end
```

We obtained the following results:

n	cond_2	est. cond	theoretical
10	1.60263e+013	3.53537e+013	1.09635e+015
11	5.23068e+014	1.23037e+015	3.55105e+016
12	1.71323e+016	3.79926e+016	1.15496e+018
13	5.62794e+017	4.25751e+017	3.76953e+019
14	1.85338e+019	7.09955e+018	1.23395e+021
15	6.11657e+020	7.73753e+017	4.04966e+022
20	2.45216e+028	4.95149e+018	1.58658e+030
40	7.65291e+058	7.02056e+019	4.69897e+060

Gabor Szegő (1895-1985) Hungarian mathematician. Szegő's most important work was in the area of extremal problems and Toeplitz matrices. Orthogonal Polynomials appeared in 1939 and was published by the American Mathematical Society. It has proved highly successful, running to four editions and many reprints over the years. He cooperated with Pólya in bringing out a joint Problem Book: Aufgaben und Lehrsätze aus der Analysis, volumes I and II (Problems and Theorems in Analysis) (1925) which has since gone through many editions and which has had an enormous impact on later generations of mathematicians.



The sequence for Vandermonde matrix with equally spaced points in [-1,1] (MATLAB script condvander2.m) is:

```
warning off
fprintf(' n    cond_inf      cond.estimate    theoretical\n')
for n=[10,20,40,80]
    t=linspace(-1,1,n);
    V=vander(t);
    et=1/pi*exp(-pi/4)*exp(n*(pi/4+1/2*log(2)));
    x=[n, norm(V,inf)*norm(inv(V),inf), condest(V), et];
    fprintf('%d %e %e %e\n',x)
end
warning on
```

We give the results:

n	cond_inf	cond.estimate	theoretical
10	2.056171e+004	1.362524e+004	1.196319e+004
20	1.751063e+009	1.053490e+009	9.861382e+008
40	1.208386e+019	6.926936e+018	6.700689e+018
80	1.027994e+039	1.003485e+039	3.093734e+038

For the Vandermonde matrix with elements of the form  $1/i$  we used the sequence (file condvander.m)

```
warning off
fprintf(' n    cond_inf      cond.estimate    theoretical\n')
for n=10:15
    t=1./(1:n);
    V=vander(t);
    x=[n, norm(V,inf)*norm(inv(V),inf), condest(V), n^(n+1)];
    fprintf('%d %e %e %e\n',x)
end
warning on
```

and we obtained:

n	cond_inf	cond.estimate	theoretical
10	5.792417e+011	5.905580e+011	1.000000e+011
11	2.382382e+013	2.278265e+013	3.138428e+012
12	1.060780e+015	9.692982e+014	1.069932e+014
13	5.087470e+016	4.732000e+016	3.937376e+015
14	2.615990e+018	2.419007e+018	1.555681e+017
15	1.436206e+020	1.294190e+020	6.568408e+018

We used warning off to disable the display of warning messages like

Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = ...

### 4.3 Gaussian Elimination

Let us consider the linear system having  $n$  equations and  $n$  unknowns

$$Ax = b, \quad (4.3.1)$$

where  $A \in \mathbb{K}^{n \times n}$ ,  $b \in \mathbb{K}^{n \times 1}$  are given, and  $x \in \mathbb{K}^{n \times 1}$  must be determined, or written in a detailed fashion

$$\left\{ \begin{array}{ll} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 & (E_1) \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 & (E_2) \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n & (E_n) \end{array} \right. \quad (4.3.2)$$

The Gaussian <sup>3</sup> elimination method has two stages:

- e1) Transforming the given system into a triangular one.
- e2) Solving the triangular system using back substitution.

During the solution of the system (4.3.1) or (4.3.2) the following transforms are allowed:

1. The equation  $E_i$  can be multiplied by  $\lambda \in \mathbb{K}^*$ . This operation will be denoted by  $(\lambda E_i) \rightarrow (E_i)$ .
2. The equation  $E_j$  can be multiplied by  $\lambda \in \mathbb{K}^*$  and added to the equation  $E_i$ , the result replacing  $E_i$ . Notation  $(E_i + \lambda E_j) \rightarrow (E_i)$ .
3. The equation  $E_i$  and  $E_j$  can be interchanged; notation  $(E_i) \longleftrightarrow (E_j)$ .

In order to express conveniently the transform into a triangular system we shall use the extended matrix:

$$\tilde{A} = [A, b] = \left[ \begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n,n+1} \end{array} \right]$$

---

Johann Carl Friedrich Gauss (1777-1855) was one of the greatest mathematicians of the 19th century — and perhaps of all time. He spent almost his entire life in Göttingen, where he was the director of the observatory for some 40 years. Already as a student in Göttingen, Gauss discovered that the 17-gon can be constructed by compass and ruler, thereby settling a problem that had been open since antiquity. His dissertation gave the first proof of the Fundamental Theorem of Algebra. He went on to make fundamental contributions to number theory, differential and non-Euclidean Geometry, elliptic and hypergeometric functions, celestial mechanics and geodesy, and various branches of physics, notably magnetism and optics. His computational efforts in celestial mechanics and geodesy, based on the principle of least squares, required the solution (by hand) of large systems of linear equations, for which he used what today are known as Gaussian elimination and relaxation methods. Gauss's work on quadrature builds upon the earlier work of Newton and Cotes.

<sup>3</sup>



with  $a_{i,n+1} = b_i$ .

Assuming  $a_{11} \neq 0$ , we shall eliminate the coefficients of  $x_1$  in  $E_j$ , for  $j = \overline{2, n}$  using the operation  $(E_j - (a_{j1}/a_{11})E_1) \rightarrow (E_j)$ . We proceed similarly for the coefficients of  $x_i$ , for  $i = \overline{2, n-1}$ ,  $j = i+1, n$ . This requires  $a_{ii} \neq 0$ .

The procedure can be described as follows: one builds the following sequence of extended matrix  $\tilde{A}^{(1)}, \tilde{A}^{(2)}, \dots, \tilde{A}^{(n)}$ , where  $\tilde{A}^{(1)} = A$  and the elements  $a_{ij}^{(k)}$  of  $\tilde{A}^{(k)}$  are given by

$$\left( E_i - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} E_{k-1} \right) \longrightarrow (E_i).$$

**Remark 4.3.1.**  $a_{ij}^{(p)}$  denotes the value of  $a_{ij}$  at the  $p$ -th step.  $\diamond$

Thus

$$\tilde{A}^{(k)} = \left[ \begin{array}{cccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1,k-1}^{(1)} & a_{1k}^{(1)} & \cdots & a_{1n}^{(1)} & a_{1,n+1}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2,k}^{(2)} & \cdots & a_{2n}^{(2)} & a_{2,n+1}^{(2)} \\ \vdots & \ddots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \vdots & & & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} & a_{k-1,n+1}^{(k-1)} & a_{k-1,n+1}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} & a_{k,n+1}^{(k)} & a_{k,n+1}^{(k)} \\ \vdots & & & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} & a_{n,n+1}^{(k)} \end{array} \right]$$

represents an equivalent linear system where the variable  $x_{k-1}$  was eliminated from the equations  $E_k, E_{k+1}, \dots, E_n$ . The system corresponding to the matrix  $\tilde{A}^{(n)}$  is triangular and equivalent to

$$\left\{ \begin{array}{l} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1n}^{(1)}x_n = a_{1,n+1}^{(1)} \\ a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n = a_{2,n+1}^{(2)} \\ \vdots \\ a_{nn}^{(n)}x_n = a_{n,n+1}^{(n)} \end{array} \right.$$

One obtains

$$x_n = \frac{a_{n,n+1}^{(n)}}{a_{nn}^{(n)}}$$

and, generally

$$x_i = \frac{1}{a_{ii}^{(i)}} \left( a_{i,n+1}^{(i)} - \sum_{j=i+1}^n a_{ij}^{(i)}x_j \right), \quad i = \overline{n-1, 1}$$

The procedure is applicable only if  $a_{ii}^{(i)} \neq 0$ ,  $i = \overline{1, n}$ . The element  $a_{ii}^{(i)}$  is called *pivot*. If during the elimination process, at the  $k$ th step one obtains  $a_{kk}^{(k)} = 0$ , one can perform the line

interchange  $(E_k) \leftrightarrow (E_p)$ , where  $k + 1 \leq p \leq n$  is the smallest integer satisfying  $a_{pk}^{(k)} \neq 0$ . In practice, such operations are necessary even if the pivot is nonzero. The reason is that a pivot which is small cause large rounding errors and even cancelation. The remedy is to choose for pivoting the subdiagonal element on the same column having the largest absolute value. That is, we must find a  $p$  such that

$$|a_{pk}^{(k)}| = \max_{k \leq i \leq n} |a_{ik}^{(k)}|,$$

and then perform the interchange  $(E_k) \leftrightarrow (E_p)$ . This technique is called *column maximal pivoting* or *partial pivoting*.

Another technique which decreases errors and prevents from the floating-point cancellation is *scaled column pivoting*. We define in a first step a scaling factor for each line

$$s_i = \max_{j=1,n} |a_{ij}| \text{ or } s_i = \sum_{j=1}^n |a_{ij}|.$$

If an  $i$  such that  $s_i = 0$  does exist, the matrix is singular. The next steps will establish what interchange is to be done. In the  $i$ -th one finds the smallest integer  $p$ ,  $i \leq p \leq n$ , such that

$$\frac{|a_{pi}|}{s_p} = \max_{i \leq j \leq n} \frac{|a_{ji}|}{s_j}$$

and then,  $(E_i) \leftrightarrow (E_p)$ . Scaling guarantees us that the largest element in each column has the relative magnitude 1, before doing the comparisons needed for line interchange. Scaling is performed only for comparison purpose, so that the division by the scaling factor does not introduce any rounding error. The third method is *total pivoting* or *maximal pivoting*. In this method, at the  $k$ th step one finds

$$\max\{|a_{ij}|, i = \overline{k, n}, j = \overline{k, n}\}$$

and line and columns interchange are carried out.

Pivoting was introduced for the first time by Goldstine and von Neumann, 1947 [38].

**Remark 4.3.2.** Some suggestions which speeds-up the running time.

1. The pivoting need not physically row or column interchange. One can manage one (or two) permutation vector(s)  $p(q)$ ;  $p[i](q[i])$  means the line (column) that was interchanged to the  $i$ th line(column). This is a good solution if matrices are stored row by row or column by column; for other representation or memory hierarchies, physical interchange could yield better results.
2. The subdiagonal elements (that vanish) need not to be computed.
3. A matrix  $A$  can be inverted solving the systems  $Ax = e_k$ ,  $k = \overline{1, n}$ , where  $e_k$  are the vectors in the canonical base of  $\mathbb{K}^n$  – simultaneous equations method.  $\diamond$

---

**MATLAB Source 4.1** Solve the system  $Ax = b$  by Gaussian elimination with scaled column pivoting.

---

```

function x=Gausselim(A,b)
%GAUSELIM - Gaussian elimination with scaled colum pivoting
%call x=Gausselim(A,b)
%A - matrix, b- right hand side vector
%x - solution

[l,n]=size(A);
x=zeros(size(b));
s=sum(abs(A),2);
A=[A,b]; %extended matrix
piv=1:n;
%Elimination
for i=1:n-1
    [u,p]=max(abs(A(i:n,i))./s(i:n)); %pivoting
    p=p+i-1;
    if u==0, error('no unique solution'), end
    if p~=i %line interchange
        piv([i,p])=piv([p,i]);
    end
    for j=i+1:n
        m=A(piv(j),i)/A(piv(i),i);
        A(piv(j),i+1:n+1)=A(piv(j),i+1:n+1)-m*A(piv(i),i+1:n+1);
    end
end
%back substitution
x(n)=A(piv(n),n+1)/A(piv(n),n);
for i=n-1:-1:1
    x(i)=(A(piv(i),n+1)-A(piv(i),i+1:n)*x(i+1:n))/A(piv(i),i);
end

```

---

**Analysis of Gaussian elimination.** The method is given in MATLAB Source 4.1, the file Gausselim.m. Our *complexity measure* is the number of floating point operations or shortly, *flops*. In the innermost loop, lines 10–11 we have  $2n - 2i + 3$  flops, total  $(n - i)(2n - 2i + 3)$  flops. For the outer loop total

$$\sum_{i=1}^{n-1} (n - i)(2n - 2i + 3) \sim \frac{2n^3}{3}.$$

For back substitution  $\Theta(n^2)$  flops. **Overall total**,  $\Theta(n^3)$ .

## 4.4 Factorization based methods

### 4.4.1 LU decomposition

**Theorem 4.4.1.** If the Gaussian elimination for the system  $Ax = b$  can be done without line interchange, then  $A$  can factor as  $A = LU$  where  $L$  is a lower triangular matrix and  $U$  is an upper triangular matrix. The pair  $(L, U)$  is the LU decomposition of the matrix  $A$ .

**Advantages.**  $Ax = b \Leftrightarrow LUx = b \Leftrightarrow Ly = b \wedge Ux = y$ . If we have to solve several systems  $Ax = b_i$ ,  $i = \overline{1, m}$ , each takes  $\Theta(n^3)$ , total  $\Theta(mn^3)$ ; if we begin with an LU decomposition which takes  $\Theta(n^3)$  and solve each system in  $\Theta(n^2)$ , we need a  $\Theta(n^3)$  time.

**Remark 4.4.2.**  $U$  is the upper triangular matrix generated by Gaussian elimination, and  $L$  is the matrix of multipliers  $m_{ij}$ .  $\diamond$

If Gaussian elimination carries out with line interchange, it holds also  $A = LU$ , but  $L$  is no more lower triangular.

The method is called  $LU$  factorization.

We give two examples where Gaussian elimination can be carried out without interchanges:

-  $A$  is line diagonal dominant, that is

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|, \quad i = \overline{1, n}$$

-  $A$  is positive definite ( $\forall x \neq 0 x^* Ax > 0$ ).

*Proof of Theorem 4.4.1.* (sketch) For  $n > 1$  we split  $A$  in the following way:

$$A = \left[ \begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{array} \right] = \left[ \begin{array}{cc} a_{11} & w^* \\ v & A' \end{array} \right],$$

where  $v$  is a  $n - 1$  column vector, and  $w^*$  - is a  $n - 1$  line vector. We can factor  $A$

$$A = \left[ \begin{array}{cc} a_{11} & w^* \\ v & A' \end{array} \right] = \left[ \begin{array}{cc} 1 & 0 \\ v/a_{11} & I_{n-1} \end{array} \right] \left[ \begin{array}{cc} a_{11} & w^* \\ 0 & A' - vw^*/a_{11} \end{array} \right].$$

The matrix  $A' - vw^*/a_{11}$  is called a *Schur complement* of  $A$  with respect to  $a_{11}$ . Then, we proceed with the recursive decomposition of Schur complement:

$$A' - vw^*/a_{11} = L'U'.$$

$$\begin{aligned} A &= \left[ \begin{array}{cc} 1 & 0 \\ v/a_{11} & I_{n-1} \end{array} \right] \left[ \begin{array}{cc} a_{11} & w^* \\ 0 & A' - vw^*/a_{11} \end{array} \right] = \\ &= \left[ \begin{array}{cc} 1 & 0 \\ v/a_{11} & I_{n-1} \end{array} \right] \left[ \begin{array}{cc} a_{11} & w^* \\ 0 & L'U' \end{array} \right] = \\ &= \left[ \begin{array}{cc} 1 & 0 \\ v/a_{11} & L' \end{array} \right] \left[ \begin{array}{cc} a_{11} & w^* \\ 0 & U' \end{array} \right]. \end{aligned}$$

□

We have several choices for  $u_{ii}$  and  $l_{ii}$ ,  $i = \overline{1, n}$ . For example, if  $l_{ii} = 1$ , we have *Doolittle factorization*, and if  $u_{ii} = 1$ , we have *Crout factorization*.

#### 4.4.2 LUP decomposition

The idea behind *LUP decomposition* is to find out three square matrices  $L$ ,  $U$  and  $P$  where  $L$  is lower triangular,  $U$  is upper triangular, and  $P$  is a permutation matrix, such that  $PA = LU$ .

The triple  $(L, U, P)$  is called the *LUP decomposition* of the matrix  $A$ .

The solution of the system  $Ax = b$  is equivalent to the solution of two triangular systems, since

$$Ax = b \Leftrightarrow LUx = Pb \Leftrightarrow Ly = Pb \wedge Ux = y$$

and

$$Ax = P^{-1}LUx = P^{-1}Ly = P^{-1}Pb = b.$$

We shall choose as pivot  $a_{k1}$  instead of  $a_{11}$ . The effect is a multiplication by a permutation matrix  $Q$ :

$$QA = \begin{bmatrix} a_{k1} & w^* \\ v & A' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & A' - vw^*/a_{k1} \end{bmatrix}.$$

Then, we compute the *LUP-decomposition* of the Schur complement.

$$P'(A' - vw^*/a_{k1}) = L'U'.$$

We define

$$P = \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} Q,$$

which is a permutation matrix too. We have now

$$\begin{aligned} PA &= \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} QA = \\ &= \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} \begin{bmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & A' - vw^*/a_{k1} \end{bmatrix} = \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & A' - vw^*/a_{k1} \end{bmatrix} = \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & P'(A' - vw^*/a_{k1}) \end{bmatrix} = \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & L'U' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{bmatrix} \begin{bmatrix} a_{k1} & w^* \\ 0 & U' \end{bmatrix}. \end{aligned}$$

Note that in this reasoning, both the column vector and the Schur complement are multiplied by the permutation matrix  $P'$ .

For an implementation, see MATLAB Source 4.2, file `lup.m`. The function `lupsolve` implements the solution of a linear algebraic system using LUP decomposition (see MATLAB source 4.5). It uses forward and backward substitutions. See MATLAB sources 4.3 and 4.4 respectively.

---

**MATLAB Source 4.2 LUP Decomposition**

---

```

function [L,U,P]=lup(A)
%LUP find LUP decomposition of matrix A
%call [L,U,P]=lup(A)
%permute effectively lines

[m,n]=size(A);
P=zeros(m,n);
piv=(1:m)';
for i=1:m-1
    %pivoting
    [pm,kp]=max(abs(A(i:m,i)));
    kp=kp+i-1;
    %line interchange
    if i~=kp
        A([i,kp],:)=A([kp,i],:);
        piv([i,kp])=piv([kp,i]);
    end
    %Schur complement
    lin=i+1:m;
    A(lin,i)=A(lin,i)/A(i,i);
    A(lin,lin)=A(lin,lin)-A(lin,i)*A(i,lin);
end;
for i=1:m
    P(i,piv(i))=1;
end;
U=triu(A);
L=tril(A,-1)+eye(m);

```

---



---

**MATLAB Source 4.3 Forward substitution**

---

```

function x=forwardsubst(L,b)
%FORWARDSUBST - forward substitution
%L - lower triangular matrix
%b - right hand side vector

x=zeros(size(b));
n=length(b);
for k=1:n
    x(k)=(b(k)-L(k,1:k-1)*x(1:k-1))/L(k,k);
end

```

---

**MATLAB Source 4.4** Back substitution

---

```
function x=backsubst (U,b)
%BACKSUBST - backward substitution
%U - upper triangular matrix
%b - right hand side vector

n=length(b);
x=zeros(size(b));
for k=n:-1:1
    x(k)=(b(k)-U(k,k+1:n)*x(k+1:n)) / U(k,k);
end
```

---

**MATLAB Source 4.5** Solution of a linear system by LUP decomposition

---

```
function x=lupsolve(A,b)
%LUPSOOLVE - solution of a linear system by LUP decomposition
%A - matrix
%b - right hand side vector

[L,U,P]=lup(A);
y=forwardsubst(L,P*b);
x=backsubst(U,y);
```

---

**4.4.3 Cholesky factorization**

Hermitian positive definite matrices can be decomposed into triangular factors twice as quickly as general matrices. The standard algorithm for this, Cholesky<sup>4</sup> factorization, is a variant of Gaussian elimination, which operates on both the left and the right of the matrix at once, preserving and exploiting the symmetry.

Systems having hermitian positive definite matrices play an important role in Numerical Linear Algebra and its applications. Many matrices that arise in physical systems are hermitian positive definite because of the fundamental physical laws.

**Properties of hermitian matrices.** Let  $A$  be a  $m \times m$  hermitian and positive definite matrix.

1. If  $X$  is a full-rank  $m \times n$  matrix, then is  $X^*AX$  hermitian positive definite;
2. Any principal submatrix of  $A$  is positive definite;
3. Any diagonal element of  $A$  is a positive real number;

---

Andre-Louis Cholesky (1875-1918) was a French military officer involving in geodesy and surveying in Crete and North Africa just before the First World War. He developed the method now named after him to compute solutions to the normal equations for some least squares data-fitting problems arising in geodesy. He died in 1918, few time before the end of the First World War. His work was posthumously published on his behalf in 1924 by a fellow officer, commander Benoît, in the Bulletin Géodesique.



4. The eigenvalues of  $A$  are positive real numbers;
5. Eigenvectors corresponding to distinct eigenvalues of a hermitian matrix are orthogonal.

A *Cholesky factorization* of a matrix  $A$  is a decomposition

$$A = R^* R, \quad r_{jj} > 0, \quad (4.4.1)$$

where  $R$  is an upper triangular matrix.

**Theorem 4.4.3.** Every hermitian positive definite matrix  $A \in \mathbb{C}^{m \times m}$  has a unique Cholesky factorization (4.4.1).

*Proof.* (Existence) Since  $A$  is hermitian and positive definite  $a_{11} > 0$  and we may set  $\alpha = \sqrt{a_{11}}$ . Note that

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & w^* \\ w & K \end{bmatrix} \\ &= \begin{bmatrix} \alpha & 0 \\ w/\alpha & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a_{11} \end{bmatrix} \begin{bmatrix} \alpha & w^*/\alpha \\ 0 & I \end{bmatrix} = R_1^* A_1 R_1. \end{aligned} \quad (4.4.2)$$

This is the basic step that is repeated in Cholesky factorization. The matrix  $K - ww^*/a_{11}$  being a  $(m - 1) \times (m - 1)$  principal submatrix of the positive definite matrix  $R_1^* A R_1^{-1}$  is positive definite and hence his upper left element is positive. By induction, all matrices that appear during the factorization are positive definite and thus *the process cannot break down*. We proceed to the factorization of  $A_1 = R_2^* A_2 R_2$ , and thus,  $A = R_1^* R_2^* A_2 R_2 R_1$ ; the process can be employed until the lower right corner is reached, getting

$$A = \underbrace{R_1^* R_2^* \dots R_m^*}_{R^*} \underbrace{R_m \dots R_2 R_1}_R;$$

this decomposition has the desired form.

(Uniqueness) In fact, the above process also establishes uniqueness. At each step, (4.4.2), the value  $\alpha = \sqrt{a_{11}}$  is determined by the form of factorization  $R^* R$  and once  $\alpha$  is determined, the first row of  $R_1^*$  is also determined. Since the analogous quantities are determined at each step, the entire factorization is unique.  $\square$

Since only half the matrix needs to be stored, it follows that half of the arithmetic operations can be avoided. Here is one of many variants of Cholesky decomposition (see MATLAB source 4.6). The input matrix  $A$  contains the main diagonal and the upper-half triangular part of the Hermitian positive matrix  $m \times m$  to be factorized. The output matrix is the upper triangular factor in decomposition  $A = R^* R$ . Each outer iteration corresponds to a single elementary factorization: the upper triangular part of submatrix  $A_{k:m,k:m}^*$  is the part above the diagonal of the Hermitian matrix to be factored at the  $k$ th step. The inner loop dominates the work. A single execution of the line

```
A(j, j:m) = A(j, j:m) - A(k, j:m) * A(k, j) / A(k, k);
```

**MATLAB Source 4.6 Cholesky Decomposition**


---

```

function R=Cholesky(A)
%CHOLESKY - Cholesky factorization
%call R=Cholesky(A)
%A - HPD matrix
%R - upper triangular matrix

[m,n]=size(A);
for k=1:m
    if A(k,k)<=0
        error('matrix is not HPD')
    end
    for j=k+1:m
        A(j,j:m)=A(j,j:m)-A(k,j:m)*A(k,j)/A(k,k);
    end
    A(k,k:m)=A(k,k:m)/sqrt(A(k,k));
end
R=triu(A);

```

---

requires one division,  $m - j + 1$  multiplications, and  $m - j + 1$  subtractions, for a total of  $\sim 2(m - j)$  flops. This calculation is repeated once for each  $j$  from  $k + 1$  to  $m$ , and that loop is repeated for each  $k$  from 1 to  $m$ . The sum is straightforward to evaluate:

$$\sum_{k=1}^m \sum_{j=k+1}^m 2(m-j) \sim 2 \sum_{k=1}^m \sum_{j=1}^k j \sim \sum_{k=1}^m k^2 \sim \frac{1}{3}m^3 \text{ flops.}$$

Thus, Cholesky factorization involves half as many operations as Gaussian elimination.

#### 4.4.4 QR decomposition

**Theorem 4.4.4.** *Let  $A \in \mathbb{R}^{m \times n}$ , with  $m \geq n$ . Then, there exists a unique  $m \times n$  orthogonal matrix  $Q$  and a unique  $n \times n$  upper triangular matrix  $R$ , having a positive diagonal ( $r_{ii} > 0$ ) such that  $A = QR$ .*

*Proof.* It is a consequence of the algorithm 4.7 (to be given in this section).  $\square$

Orthogonal and unitary matrices are desirable for numerical computation because they preserve lengths, angles, and do not magnify errors.

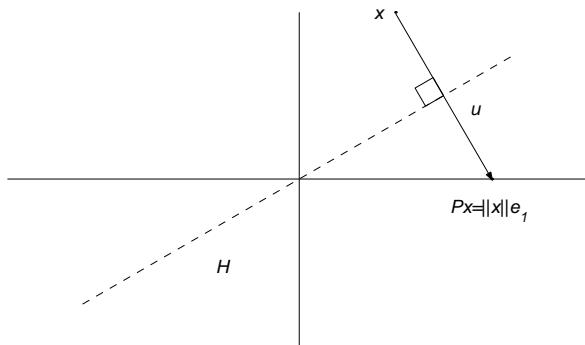


Figure 4.2: A Householder reflector

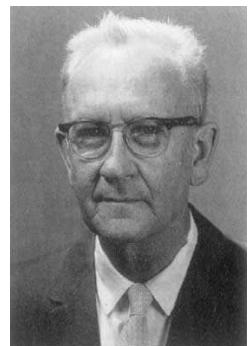
A *Householder*<sup>5</sup> transform (or a *reflection*) is a matrix of form  $P = I - 2uu^T$ , where  $\|u\|_2 = 1$ . One easily checks that  $P = P^T$  and

$$PP^T = (I - 2uu^T)(I - 2uu^T) = I - 4uu^T + 4uu^Tuu^T = I,$$

hence  $P$  is a symmetric orthogonal matrix. It is called a reflection since  $Px$  is the reflection of  $x$  with respect to the hyperplane  $H$  which pass through the origin and is orthogonal to  $u$  (Figure 4.2).

Given a vector  $x$ , it is easy to find a Householder reflection  $P = I - 2uu^T$  to zero out all but the first entry of  $x$ :  $Px = [c, 0, \dots, 0]^T = ce_1$ . We do this as follows. Write  $Px = x - 2u(u^Tx) = ce_1$ , so that  $u = \frac{1}{2(u^Tx)}(x - ce_1)$ , i.e.,  $u$  is a linear combination of  $x$  and  $e_1$ . Since  $\|x\|_2 = \|Px\|_2 = |c|$ ,  $u$  must be parallel to the vector  $\tilde{u} = x \pm \|x\|_2 e_1$ , and so  $u = \tilde{u}/\|\tilde{u}\|_2$ . One can verify that any choice of sign yields a  $u$  satisfying  $Px = ce_1$ , as long as  $\tilde{u} \neq 0$ . We will use  $\tilde{u} = x + \text{sign}(x_1)\|x\|_2 e_1$ , since this means that there is no cancellation in computing the first component of  $\tilde{u}$ . If  $x_1 = 0$ , we choose conventionally  $\text{sign}(x_1) = 1$ .

Alston S. Householder (1904-1993), American mathematician. Important contributions to mathematical biology and mainly to numerical linear algebra. His well known book "The theory of matrices in numerical analysis" has a great impact on development of numerical analysis and computer science.



In summary, we get

$$\tilde{u} = \begin{bmatrix} x_1 + \text{sign}(x_1)\|x\|_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \text{ with } u = \frac{\tilde{u}}{\|\tilde{u}\|_2}.$$

In practice, we can store  $\tilde{u}$  instead of  $u$  to save the work of computing  $u$ , and use the formula  $P = I - \frac{2}{\|u\|_2^2} \tilde{u} \tilde{u}^T$  instead of  $P = I - 2uu^T$ . The matrix  $P'_i$  needs not to be built; rather we can apply it directly:

$$\begin{aligned} A_{i:m,i:n} &= P'_i A_{i:m,i:n} = (I - 2u_i u_i^T) A_{i:m,i:n} \\ &= A_{i:m,i:n} - 2u_i (u_i^T A_{i:m,i:n}). \end{aligned}$$

MATLAB Source 4.7 describes the process of computing QR decomposition using Householder reflections.

---

#### MATLAB Source 4.7 QR decomposition using Householder reflections

---

```
function [R,Q]=HouseQR(A)
%HouseQR - QR decomposition using Householder reflections
%call [R,Q]=HouseQR(A)
%A mxn matrix, R upper triangular, Q orthogonal

[m,n]=size(A);
u=zeros(m,n); %reflection vectors
%compute R
for k=1:n
    x=A(k:m,k);
    x(1)=mysign(x(1))*norm(x)+x(1);
    u(k:m,k)=x/norm(x);
    A(k:m,k:n)=A(k:m,k:n)-2*u(k:m,k)*(u(k:m,k)'*A(k:m,k:n));
end
R=triu(A(1:n,:));
if nargout==2 %Q wanted
    Q=eye(m,n);
    for j=1:n
        for k=n:-1:1
            Q(k:m,j)=Q(k:m,j)-2*u(k:m,k)*(u(k:m,k)'*Q(k:m,j));
        end
    end
end
%sign
function y=mysign(x)
if x>=0, y=1;
else, y=-1;
end
```

---

Starting from the relation

$$Ax = b \Leftrightarrow QRx = b \Leftrightarrow Rx = Q^T b,$$

we can choose the following strategy for the solution of linear system  $Ax = b$ :

1. Determine the factorization  $A = QR$  of  $A$ ;
2. Compute  $y = Q^T b$ ;
3. Solve the upper triangular system  $Rx = y$ .

The computation of  $Q^T b$  can be performed by  $Q^T b = P_n P_{n-1} \dots P_1 b$ , so we need to store the product of  $b$  by  $P_1, P_2, \dots, P_n$  – see MATLAB Source 4.8.

---

#### MATLAB Source 4.8 Solution of $Ax = b$ using QR method

---

```
function x=QRSolve(A,b)
%QRSolve - solutions os a system using QR decomposition

[m,n]=size(A);
u=zeros(m,n); %reflection vectors
%compute R and Q^T*b
for k=1:n
    x=A(k:m,k);
    x(1)=mysign(x(1))*norm(x)+x(1);
    u(k:m,k)=x/norm(x);
    A(k:m,k:n)=A(k:m,k:n)-2*u(k:m,k)*(u(k:m,k)'*A(k:m,k:n));
    b(k:m)=b(k:m)-2*u(k:m,k)*(u(k:m,k)'*b(k:m));
end
R=triu(A(1:n,:));
x=R\b(1:n);
```

---

The cost of QR decomposition  $A = QR$  is  $2n^2m - \frac{2}{3}n^3$ , and the costs for  $Q^T b$  and  $Qx$  are both  $O(mn)$ .

If we wish to compute the matrix  $Q$  explicitly, we can build  $QI$  explicitly, by computing the column  $Qe_1, Qe_2, \dots, Qe_m$  of  $QI$ , as shown in MATLAB Source 4.7.

## 4.5 Strassen's algorithm for matrix multiplication

Let  $A, B \in \mathbb{R}^{n \times n}$ . We wish to compute  $C = AB$ . Suppose  $n = 2^k$ . We split  $A$  and  $B$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Classical algorithm requires 8 multiplications and 4 additions for one step; the running time is  $T(n) = \Theta(n^3)$ , since  $T(n) = 8T(n/2) + \Theta(n^2)$ .

We are interested in reducing the number of multiplications. Volker Strassen [92] discovered a method to reduce the number of multiplications to 7 per step. One computes the following quantities

$$\begin{aligned} p_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ p_2 &= (a_{21} + a_{22})b_{11} \\ p_3 &= a_{11}(b_{12} - b_{22}) \\ p_4 &= a_{22}(b_{21} - b_{11}) \\ p_5 &= (a_{11} + a_{12})b_{22} \\ p_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ p_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ \\ c_{11} &= p_1 + p_4 - p_5 + p_7 \\ c_{12} &= p_3 + p_5 \\ c_{21} &= p_2 + p_4 \\ c_{22} &= p_1 + p_3 - p_2 + p_6. \end{aligned}$$

Since we have 7 multiplications and 18 additions per step, the running times verifies the following recurrence

$$T(n) = 7T(n/2) + \Theta(n^2).$$

The solution is

$$T(n) = \Theta(n^{\log_2 7}) \sim 28n^{\log_2 7}.$$

For an implementation see MATLAB Source 4.9.

The algorithm can be extended to matrices of  $n = m \cdot 2^k$  size. If  $n$  is odd, then the last column of the result can be computed using standard method; then Strassen's algorithm is applied to  $n - 1$  by  $n - 1$  matrices:  $m \cdot 2^{k+1} \rightarrow m \cdot 2^k$ . The  $p$ -s can be computed in parallel; the  $c$ -s, too.

The theoretical speed-up of matrix multiplication translates into a speed-up of matrix inversion, hence to the solution of linear algebraic systems. If  $M(n)$  is the time for multiplication of two  $n \times n$  matrices and  $I(n)$  is the inversion time for a  $n \times n$  matrix, then  $M(n) = \Theta(n)$ . This can be proven in two steps: we show that  $M(n) = O(I(n))$  and then  $I(n) = O(M(n))$ .

**Theorem 4.5.1 (Multiplication is not harder than inversion).** *If we can invert a  $n \times n$  matrix in  $I(n)$  time, where  $I(n) = \Omega(n^2)$  satisfies the regularity condition  $I(3n) = O(I(n))$ , then we can multiply two  $n$ th order matrices in  $O(I(n))$  time.*

Note that  $I(n)$  satisfies the regularity condition only if  $I(n)$  has not large jumps in its values. For example, if  $I(n) = \Theta(n^c \log^d n)$ , for any constants  $c > 0$ ,  $d \geq 0$ , then  $I(n)$  satisfies the regularity conditions.

**Theorem 4.5.2 (Inversion is not harder than multiplication).** *If we can multiply two real  $n \times n$  matrices in  $M(n)$  time, where  $M(n) = \Omega(n^2)$ ,  $M(n)$  satisfies the regularity conditions  $M(n) = O(M(n+k))$  for each  $k$ ,  $0 \leq k \leq n$ , and  $M(n/2) \leq cM(n)$ , for any constant  $c < 1/2$ , then we can invert a real  $n \times n$  nonsingular matrix in  $O(M(n))$  time.*

For proofs of last two theorems, see [18].

**MATLAB Source 4.9** Strassen's algorithm for matrix multiplication

```

function C=strass(A,B,mmin)
%STRASS - Strassen algorithm for matrix multiplication
%size = 2^k
%A,B - square matrices
%C - product A*B
%mmin - minimum size

[m,n]=size(A);
if m<=mmin
    %classical multiplication
    C=A*B;
else
    %subdivision and recursive call
    n=m/2;
    u=1:n; v=n+1:m;
    P1=strass(A(u,u)+A(v,v),B(u,u)+B(v,v),mmin);
    P2=strass(A(v,u)+A(v,v),B(u,u),mmin);
    P3=strass(A(u,u),B(u,v)-B(v,v),mmin);
    P4=strass(A(v,v),B(v,u)-B(u,u),mmin);
    P5=strass(A(u,u)+A(u,v),B(v,v),mmin);
    P6=strass(A(v,u)-A(u,u),B(u,u)+B(u,v),mmin);
    P7=strass(A(u,v)-A(v,v),B(v,u)+B(v,v),mmin);
    C(u,u)=P1+P4-P5+P7;
    C(u,v)=P3+P5;
    C(v,u)=P2+P4;
    C(v,v)=P1+P3-P2+P6;
end

```

---

## 4.6 Solution of Algebraic Linear Systems in MATLAB

Let  $m$  be the number of equations and  $n$  the number of unknowns. The fundamental tool for solving a linear system of equations is the backslash operator, \ (see Section 1.3.3).

It handles three types of linear systems, squared ( $m = n$ ), overdetermined ( $m > n$ ) and underdetermined ( $m < n$ ). We shall give more details on overdetermined system in Chapter 5. More generally, the \ operator can be used to solve  $AX = B$ , where  $B$  is a matrix with  $n$  columns; in this case MATLAB solves  $AX(:,j) = B(:,j)$  for  $j = 1 : n$ . We can use  $X = B/A$  to solve systems of the form  $XA = B$ .

### 4.6.1 Square systems

If  $A$  is an  $n \times n$  nonsingular matrix, then  $A\b$  is the solution of the system  $Ax=b$ , computed by LU factorization with partial pivoting. During the solution process MATLAB computes  $rcond(A)$ , and it prints a warning message if the result is smaller than about  $\text{eps}$ :

```
x = hilb(15)\ones(15,1)
```

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.543404e-018.
```

MATLAB recognizes three special forms of square systems and takes advantage of them to reduce the computation.

- Triangular matrix, or permutation of a triangular matrix. The system is solved by substitution.
- Upper Hessenberg matrix. The system is solved by LU decomposition with partial pivoting, taking the advantage of the upper Hessenberg form.
- Hermitian positive definite matrix. Cholesky factorization is used instead of LU factorization. When `is` is called with a Hermitian matrix that has positive diagonal elements MATLAB attempts to Cholesky factorize the matrix. How does MATLAB know the matrix is definite? If the Cholesky factorization succeeds, the matrix is positive definite and it is used to solve the system; otherwise an LU factorization is carried out.

## 4.6.2 Overdetermined systems

In general, if  $m > n$ , the system  $Ax = b$  has no solution. MATLAB expression `A\b` gives a least squares solution to the system, that is, it minimizes `norm(A*x-b)` (the 2-norm of the residual) over all vectors  $x$ . If  $A$  has full rank  $n$  there is a unique least squares solution. If  $A$  has rank  $k$  less than  $n$  then  $\underline{A}$  is a basic solution—one with at most  $k$  nonzero elements ( $k$  is determined, and  $x$  computed, using the QR factorization with column pivoting). In the latter case MATLAB displays a warning message.

A least squares solution to  $Ax = b$  can also be computed as `xmin = pinv(A)*b`, where the `pinv` function computes the pseudo-inverse. In the case where  $A$  is rank-deficient, `xmin` is the unique solution of minimal 2-norm.

The (Moore-Penrose) pseudo-inverse generalizes the notion of inverse to rectangular and rank-deficient matrices  $A$  and is written  $A^+$ . It is computed with `pinv(A)`. The pseudo-inverse  $A^+$  of  $A$  can be characterized as the unique matrix  $X = A^+$  satisfying the four conditions  $AXA = A$ ,  $XAX = X$ ,  $(XA)^* = XA$  and  $(AX)^* = AX$ . It can also be written explicitly in terms of the singular value decomposition (SVD): if the SVD of  $A$  is  $A = U\Sigma V^*$ , then  $A^+ = V\Sigma^+U^*$ , where  $\Sigma^+$  is  $n \times m$  diagonal with  $(i;i)$  entry  $1/\sigma_i$  if  $\sigma_i > 0$  and otherwise 0. To illustrate,

```
>> Y=pinv(ones(3))
Y =
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
>> A=[0 0 0 0; 0 1 0 0; 0 0 2 0]
A =
    0      0      0      0
    0      1      0      0
    0      0      2      0
>> pinv(A)
```

```
ans =
0         0         0
0    1.0000         0
0         0    0.5000
0         0         0
```

A vector that minimizes the 2-norm of  $Ax - b$  over all nonnegative vectors  $x$ , for real  $A$  and  $b$ , is computed by `lsqnonneg`. The simplest usage is  $x = \text{lsqnonneg}(A, b)$ , and several other input and output arguments can be specified, including a starting vector for the iterative algorithm that is used. Example

```
>> A = gallery('lauchli', 2, 0.2), b = [1 2 4]';
A =
1.0000    1.0000
0.2000         0
         0    0.2000
>> x=A\b;
>> xn=lsqnonneg(A, b);
>> [x xn], [norm(A*x-b) norm(A*xn-b)]
ans =
-4.2157         0
 5.7843    1.7308
ans =
 4.0608    4.2290
```

### 4.6.3 Underdetermined systems

In the case of an underdetermined system, we can have either no solution or infinitely many. In the latter case,  $A\backslash b$  produces a basic solution, one with at most  $k$  nonzero elements, where  $k$  is the rank of  $A$ . This solution is generally not the solution of minimal 2-norm, which can be computed as `pinv(A)*b`. If the system has no solution (that is, it is inconsistent), then  $A\backslash b$  is a least squares solution. The next example illustrates the difference between the  $\backslash$  and `pinv` solutions:

```
>> A = [1 1 1; 1 1 -1], b=[3; 1]
A =
1         1         1
1         1        -1
b =
 3
 1
>> x=A\b; y = pinv(A)*b;
>> [x y]
ans =
 2.0000    1.0000
 0    1.0000
 1.0000    1.0000
>> [norm(x) norm(y)]
ans =
 2.2361    1.7321
```

MATLAB uses QR factorization with column pivoting. Consider the example

```
>> R=fix(10*rand(2, 4))
R =
    9      6      8      4
    2      4      7      0
>> b=fix(10*rand(2, 1))
b =
    8
    4
```

The system has 2 equations and 4 unknowns. Since the coefficient matrix contains small integers, it is appropriate to display the solution in rational format. One obtains a particular solution with:

```
>> format rat
>> p=R\b
p =
    24/47
    0
    20/47
    0
```

One of the nonzero components is  $p(3)$ , since  $R(:, 3)$  is the column of  $R$  with the largest norm. The other nonzero component is  $p(1)$ , because  $R(:, 1)$  dominates after  $R(:, 3)$  is eliminated.

The complete solution to the underdetermined system can be characterized by adding an arbitrary vector from the null space, which can be found using the `null` function with an option requesting a "rational" basis:

```
>> Z=null(R, 'r')
Z =
    5/12      -2/3
   -47/24      1/3
     1          0
     0          1
```

It can be confirmed that  $R \cdot Z$  is zero and that any vector  $x$  where  $R \cdot Z$  is the zero matrix, and any vector  $x$  where  $x = p + Z \cdot q$ , for an arbitrary vector  $q$  satisfies  $R \cdot x = b$ .

#### 4.6.4 LU and Cholesky factorizations

The `lu` function computes an LUP factorization with partial pivoting. The MATLAB call `[L, U, P]=lu(A)` returns the triangular factors and the permutation matrix. The form `[L, U]=lu(A)` returns  $L = P^T L$ , so  $L$  is a triangular matrix with row permuted. We illustrate with an example.

```
>> format short g
>> A = gallery('fiedler', 3), [L,U]=lu(A)
A =
```

```

0      1      2
1      0      1
2      1      0
L =
0          1          0
0.5       -0.5       1
1          0          0
U =
2      1      0
0      1      2
0      0      2

```

The `lu` function works also for rectangular matrices. If  $A$  is  $m \times n$ , the call `[L, U] = lu(A)` produces an  $m \times n$   $L$  and an  $n \times n$   $U$  if  $m \geq n$  and an  $m \times m$   $L$  and an  $m \times n$   $U$  if  $m < n$ .

The solution of  $Ax=b$  with square  $A$  by using  $x=A\backslash b$  is equivalent to the MATLAB sequence:

```
[L,U] = lu(A); x = U\ (L\b);
```

Determinants and matrix inverses are computed also through LU factorization:

```
det(A)=det(L)*det(U)=+-prod(diag(U))
inv(A)=inv(U)*inv(L)
```

The command `chol(A)`, where  $A$  is Hermitian positive definite, computes the Cholesky decomposition of  $A$ . Since \ operator knows to handle triangular matrices, the system can be solved quickly with  $x=R\backslash(R'\backslash R\backslash b)$ . We give an example of Cholesky factorization:

```

>> A=pascal(4)
A =
1      1      1      1
1      2      3      4
1      3      6      10
1      4     10     20
>> R=chol(A)
R =
1      1      1      1
0      1      2      3
0      0      1      3
0      0      0      1

```

Note that `chol` looks only at the elements in the upper triangle of  $A$  (including the diagonal) — it factorizes the Hermitian matrix agreeing with the upper triangle of  $A$ . An error is produced if  $A$  is not positive definite. The `chol` function can be used to test whether a matrix is positive definite `[R,p] = chol(A)`, where the integer  $p$  will be zero if the factorization succeeds and positive otherwise; see `help chol` for more details about  $p$ .

Function `cholupdate` modifies the Cholesky factorization when the original matrix is subjected to a rank 1 perturbation (that is a matrix of the form either  $+xx^*$  or  $-xx^*$ , where  $x$  is a vector).

### 4.6.5 QR factorization

There are four variants of the QR factorization — full (complete) or economy (reduced) size, and with or without column permutation.

The full QR factorization of an  $m \times n$  matrix  $C$ ,  $m > n$ , produce an  $m \times m$  matrix  $Q$ , orthogonal and an  $m \times n$  upper triangular matrix  $R$ . The call syntax is  $[Q, R] = qr(C)$ . In many cases, the last  $m - n$  columns are not needed, because they are multiplied by the zeros in the bottom portion of  $R$ . For example, for the matrix  $C$ , given below:

```
C =
    1     1
    1     2
    1     3
>> [Q, R] = qr(C)
Q =
    -0.5774    0.7071    0.4082
    -0.5774    0.0000   -0.8165
    -0.5774   -0.7071    0.4082
R =
    -1.7321   -3.4641
        0    -1.4142
        0         0
```

The economy size or reduced QR factorization produces an  $m \times n$  rectangular matrix  $Q$ , with orthonormal columns and an  $n \times n$  upper triangular matrix  $R$ . Example

```
>> [Q, R] = qr(C, 0)
Q =
    -0.5774    0.7071
    -0.5774    0.0000
    -0.5774   -0.7071
R =
    -1.7321   -3.4641
        0    -1.4142
```

For larger, highly rectangular matrices, the savings in both time and memory can be quite important.

In contrast to the LU factorization, the standard QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. A QR factorization with column pivoting has the form  $AP = QR$ , where  $P$  is a permutation matrix. The permutation strategy that is used produces a factor  $R$  whose diagonal elements are non-increasing:  $|r_{11}| \geq |r_{22}| \geq \dots \geq |r_{nn}|$ . Column pivoting is particularly appropriate when  $A$  is suspected of being rank-deficient, as it helps to reveal near rank-deficiency. Roughly speaking, if  $A$  closed to a matrix of rank  $r < n$  then the last  $n - r$  diagonal elements of  $R$  will be of order  $\text{eps} * \text{norm}(A)$ . A third output argument forces  $qr$  function to use column pivoting and return the permutation matrix:  $[Q, R, P] = qr(A)$ . Example:

```
>> [Q, R, P] = qr(C)
Q =
```

```

-0.2673    0.8729    0.4082
-0.5345    0.2182   -0.8165
-0.8018   -0.4364    0.4082
R =
-3.7417   -1.6036
      0     0.6547
      0       0
P =
  0     1
  1     0

```

If we combine pivoting with economy form, the `qr` function return a vector instead a permutation matrix:

```

>> [Q,R,P]=qr(C,0)
Q =
-0.2673    0.8729
-0.5345    0.2182
-0.8018   -0.4364
R =
-3.7417   -1.6036
      0     0.6547
P =
  2     1

```

Functions `qrdelete`, `qrinsert`, and `qrupdate` modify the QR factorization when a column of the original matrix is deleted or inserted or when a rank 1 perturbation is added.

Consider now a system  $Ax = b$ , where  $A$  is an  $n$ -order square matrix of the form:

$$A = (a_{i,j}), \quad a_{i,j} = \begin{cases} 1, & \text{pentru } i = j \text{ sau } j = n; \\ -1, & \text{pentru } i > j; \\ 0, & \text{în rest.} \end{cases}$$

For example, if  $n = 6$ ,

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix}.$$

For a given  $n$ , we can generate such a matrix with the sequence

```
A=[-tril(ones(n,n-1),-1)+eye(n,n-1),ones(n,1)]
```

Suppose we set  $b$  using  $b=A\ast ones(n,1)$ . The solution of our system is  $x = [1, 1, \dots, 1]^T$ . For  $n=100$ , the \ operator yields

```

>> x=A\b;
>> reshape(x,10,10)
ans =
    1     1     1     1     1     1     0     0     0     0
    1     1     1     1     1     1     0     0     0     0
    1     1     1     1     1     1     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     0
    1     1     1     1     1     0     0     0     0     1
>> norm(b-A*x)/norm(b)
ans =
    0.3191

```

a wrong result, although A is well conditioned

```

>> cond(A)
ans =
    44.8023

```

If we solve the system by QR method, we obtain

```

>> [Q,R]=qr(A);
>> x2=R\ (Q'*b);
>> x2'
ans =
    Columns 1 through 6
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    ...
    Columns 97 through 100
    1.0000    1.0000    1.0000    1.0000
>> norm(b-A*x2)/norm(b)
ans =
    8.6949e-016

```

## 4.6.6 The **linsolve** function

The **linsolve** function allows a faster solution of a square or a rectangular system, by indicating the properties of the system matrix. It can be an alternative to \ operator when efficiency is important. The call

```
x = linsolve(A,b,opts)
```

solves the linear system  $A*x=b$ , using the solver that is most appropriate given the properties of the matrix A, which you specify in the structure **opts**. If A does not have the properties that you specify in **opts**, **linsolve** returns incorrect results and does not return an error message. The following table lists all the field of **opts** and their corresponding matrix properties. The values of the fields of **opts** must be logical and the default value for all fields is false.

Field Name	Matrix Property
LT	Lower triangular
UT	Upper triangular
UHESS	Upper Hessenberg
SYM	Real symmetric or complex Hermitian
POSDEF	Positive definite
RECT	General rectangular
TRANSA	Conjugate transpose - specifies whether the function solves $A \cdot x = b$ or $A' \cdot x = b$

If `opts` is missing, `linsolve` solves the linear system using LU factorization with partial pivoting when  $A$  is square and QR factorization with column pivoting otherwise. It returns a warning if  $A$  is square and ill conditioned or if it is not square and rank deficient. Example:

```
>> A = triu(rand(5,3)); x = [1 1 1 0 0]';
>> b = A' * x;
>> y1 = (A') \ b
y1 =
    1.0000
    1.0000
    1.0000
    0
    0
>> opts.UT = true; opts.TRANSA = true;
>> y2 = linsolve(A,b,opts)
y2 =
    1.0000
    1.0000
    1.0000
    0
    0
```

## 4.7 Iterative refinement

If the solution method for  $Ax = b$  is unstable, then  $A\bar{x} \neq b$ , where  $\bar{x}$  is the computed solution. We shall compute a correction  $\Delta x$  such that

$$A(\bar{x} + \Delta x_1) = b \Rightarrow A\Delta x_1 = b - A\bar{x}$$

We solve the system and we obtain a new  $\bar{x}$ ,  $\bar{x} := x + \Delta x_1$ . If again  $Ax \neq b$ , then we compute a new correction, until

$$\|\Delta x_i - \Delta x_{i-1}\| < \varepsilon \quad \text{or} \quad \|Ax - b\| < \varepsilon.$$

The computation of the residual vector  $r = b - Ax$ , will be performed in double precision.

## 4.8 Iterative solution of Linear Algebraic Systems

We wish to compute the solution

$$Ax = b, \quad (4.8.1)$$

when  $A$  is invertible. Suppose we have found a matrix  $T$  and a vector  $c$  such that  $I - T$  is invertible and the unique fixpoint of the equation

$$x = Tx + c \quad (4.8.2)$$

equates the solution of the system  $Ax = b$ . Let  $x^*$  be the solution of (4.8.1) or, equivalently, of (4.8.2).

Iteration:  $x^{(0)}$  given; one defines the sequence  $(x^{(k)})$  by

$$x^{(k+1)} = Tx^{(k)} + c, \quad k \in \mathbb{N}. \quad (4.8.3)$$

**Lemma 4.8.1.** *If  $\rho(X) < 1$ , then there exists  $(I - X)^{-1}$  and*

$$(I - X)^{-1} = I + X + X^2 + \cdots + X^k + \dots$$

*Proof.* Let

$$S_k = I + X + \cdots + X^k$$

$$(I - X)S_k = I - X^{k+1}$$

$$\lim_{k \rightarrow \infty} (I - X)S_k = I \Rightarrow \lim_{k \rightarrow \infty} S_k = (I - X)^{-1}$$

since  $X^{k+1} \rightarrow 0 \Leftrightarrow \rho(X) < 1$  (Theorem 4.1.8).  $\square$

**Theorem 4.8.2.** *The following statements are equivalent*

- (1) method (4.8.3) is convergent;
- (2)  $\rho(T) < 1$ ;
- (3)  $\|T\| < 1$  for at least a matrix norm.

*Proof.*

$$\begin{aligned} x^{(k)} &= Tx^{(k-1)} + c = T(Tx^{(k-2)} + c) + c = \cdots = \\ &= T^k x^{(0)} + (I + T + \cdots + T^{n-1}) \end{aligned}$$

(4.8.3) convergent  $\Leftrightarrow I - T$  invertible  $\Leftrightarrow \rho(T) < 1 \Leftrightarrow \exists \|\cdot\|$  such that  $\|T\| < 1$  (from Theorem 4.1.8).  $\square$

Banach's fixpoint theorem implies:

**Theorem 4.8.3.** If there exists  $\|\cdot\|$  such that  $\|T\| < 1$ , the sequence  $(x^{(k)})$  given by (4.8.3) is convergent for any  $x^{(0)} \in \mathbb{R}^n$  and the following estimations hold

$$\|x^* - x^{(k)}\| \leq \|T\|^k \|x^{(0)} - x\| \quad (4.8.4)$$

$$\|x^* - x^{(k)}\| \leq \frac{\|T\|^k}{1 - \|T\|} \|x^{(1)} - x^{(0)}\| \leq \frac{\|T\|}{1 - \|T\|} \|x^{(1)} - x^{(0)}\|. \quad (4.8.5)$$

An iterative method for the solution of an linear algebraic system  $Ax = b$  starts from an initial approximation  $x^{(0)} \in \mathbb{R}^n(\mathbb{C}^n)$  and generates a sequence of vectors  $\{x^{(k)}\}$ , that converges to the solution of the system  $x^*$ . Such techniques transform the initial system into an equivalent system, having the form  $x = Tx + c$ ,  $T \in \mathbb{K}^{n \times n}$ ,  $c \in \mathbb{K}^n$ . One generates a sequence  $x^{(k)} = Tx^{(k-1)} + c$ .

The stopping criterion is

$$\|x^{(k)} - x^{(k-1)}\| \leq \frac{1 - \|T\|}{\|T\|} \varepsilon. \quad (4.8.6)$$

It is based on the result:

**Proposition 4.8.4.** If  $x^*$  is the solution of (4.8.2), and  $\|T\| < 1$ , then

$$\|x^* - x^{(k)}\| \leq \frac{\|T\|}{1 - \|T\|} \|x^{(k)} - x^{(k-1)}\|. \quad (4.8.7)$$

*Proof.* Let  $p \in \mathbb{N}^*$ . We have

$$\|x^{(k+p)} - x^{(k)}\| \leq \|x^{(k+1)} - x^{(k)}\| + \cdots + \|x^{(k+p)} - x^{(k+p-1)}\|. \quad (4.8.8)$$

On the other hand, (4.8.3) implies

$$\|x^{(m+1)} - x^{(m)}\| \leq \|T\| \|x^{(m)} - x^{(m-1)}\|$$

or, for  $k < m$

$$\|x^{(m+1)} - x^{(m)}\| \leq \|T\|^{m-k+1} \|x^{(k)} - x^{(k-1)}\|.$$

By applying successively these inequalities for  $m = \overline{k, k+p-1}$ , the relation (4.8.8) becomes

$$\begin{aligned} \|x^{(k+p)} - x^{(k)}\| &\leq (\|T\| + \cdots + \|T\|^p) \|x^{(k)} - x^{(k-1)}\| \\ &\leq (\|T\| + \cdots + \|T\|^p + \dots) \|x^{(k)} - x^{(k-1)}\|. \end{aligned}$$

Since  $\|T\| < 1$ , we have

$$\|x^{(k+p)} - x^{(k)}\| \leq \frac{\|T\|}{1 - \|T\|} \|x^{(k)} - x^{(k-1)}\|,$$

which when passing to the limit with respect to  $p$  yields to (4.8.7).  $\square$

If  $\|T\| \leq 1/2$ , inequality (4.8.7) becomes

$$\|x^* - x^{(k)}\| \leq \|x^k - x^{(k-1)}\|,$$

and the stopping criterion

$$\|x^k - x^{(k-1)}\| \leq \varepsilon.$$

Iterative methods are seldom used to the solution of small systems since the time required to attain the desired accuracy exceeds the time required for Gaussian elimination. For large sparse systems (i.e. systems whose matrix has many zeros), iterative methods are efficient both in time and space.

Let the system  $Ax = b$ . Suppose we can split  $A$  as  $A = M - N$ . If  $M$  can be easily inverted (diagonal, triangular, and so on) it is more convenient to carry out the computation in the following manner

$$Ax = b \Leftrightarrow Mx = Nx + b \Leftrightarrow x = M^{-1}Nx + M^{-1}b$$

The last equation is in the form  $x = Tx + c$ , where  $T = M^{-1}N = I - M^{-1}A$ . One obtains the sequence

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \quad k \in \mathbb{N},$$

where  $x^{(0)}$  is an arbitrary vector.

The first splitting we consider is  $A = D - L - U$ , where

$$(D)_{ij} = a_{ij}\delta_{ij}, \quad (-L)_{ij} = \begin{cases} a_{ij}, & i > j \\ 0, & \text{otherwise} \end{cases}$$

$$(-U)_{ij} = \begin{cases} a_{ij}, & i < j \\ 0, & \text{otherwise} \end{cases}$$

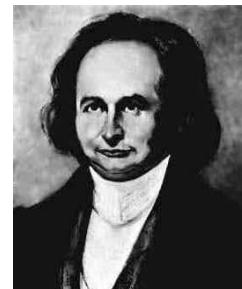
Taking  $M = D$ ,  $N = L + U$ , one successively obtains

$$Ax = b \Leftrightarrow Dx = (L + U)x + b \Leftrightarrow x = D^{-1}(L + U)x + D^{-1}b$$

Thus,  $T = T_J = D^{-1}(L + U)$ ,  $c = c_J = D^{-1}b$ ; — this is *Jacobi's method* ( $D$  is invertible, why?), due to Carl Jacobi.<sup>6</sup> [49]

Another decomposition is  $A = D - L - U$ ,  $M = D - L$ ,  $N = U$  which yields to  $T_{GS} = (D - L)^{-1}U$ , and  $c_{GS} = (D - L)^{-1}b$  — called *Gauss-Seidel method* ( $D - L$  invertible, why?)

Carl Gustav Jacob Jacobi (1804-1851) was a contemporary of Gauss, and with him one of the most important 19th-century mathematicians in Germany. His name is connected with elliptic functions, partial differential equations of dynamics, calculus of variations, celestial mechanics; functional determinants also bear his name. In his work on celestial mechanics he invented what is now called the Jacobi method for solving linear algebraic systems.



**MATLAB Source 4.10 Jacobi method for linear systems**

---

```
function [x,ni]=Jacobi(A,b,x0,err,nitmax)
%JACOBI Jacobi method
%call [x,ni]=Jacobi(A,b,x0,err,nitmax)
%parameters
%A - system matrix
%b - right hand side vector
%x0 - starting vector
%err - tolerance (default 1e-3)
%nitmax - maximum number of iterations (default 50)
%x - solution
%ni -number of actual iterations

%parameter check
if nargin < 5, nitmax=50; end
if nargin < 4, err=1e-3; end
if nargin <3, x0=zeros(size(b)); end
[m,n]=size(A);
if (m~=n) | (n~=length(b))
    error('illegal size')
end
%compute T and c (prepare iterations)
M=diag(diag(A));
N=M-A;
T=inv(M)*N;
c=inv(M)*b;
alfa=norm(T,inf);
x=x0(:,1);
for i=1:nitmax
    x0=x;
    x=T*x0+c;
    if norm(x-x0,inf)<(1-alfa)/alfa*err
        ni=i;
        return
    end
end
error('iteration number exceeded')
```

---

Let us examine Jacobi iteration  $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k-1)} \right)$ .

Computation of  $x_i^{(k)}$  uses all components of  $x^{(k-1)}$  (simultaneous substitution). Since for  $i > 1$ ,  $x_1^{(k)}, \dots, x_{i-1}^{(k)}$  have already been computed, and we suppose they are better approximations of the solution components than  $x_1^{(k-1)}, \dots, x_{i-1}^{(k-1)}$  it seems reasonable to compute  $x_i^{(k)}$  using the most recent values, i.e.

$$x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{k-1} a_{ij} x_j^{(k)} - \sum_{k=i+1}^n a_{ij} x_j^{(k-1)} \right).$$

One can state necessary and sufficient conditions for the convergence of Jacobi and Gauss-Seidel methods

$$\rho(T_J) < 1$$

$$\rho(T_{GS}) < 1$$

and sufficient conditions: there exists  $\|\cdot\|$  such that

$$\|T_J\| < 1$$

$$\|T_{GS}\| < 1.$$

We can improve Gauss-Seidel method introducing a parameter  $\omega$  and splitting

$$M = \frac{D}{\omega} - L.$$

We have

$$A = \left( \frac{D}{\omega} - L \right) - \left( \frac{1-\omega}{\omega} D + U \right),$$

and the iteration is

$$\left( \frac{D}{\omega} - L \right) x^{(k+1)} = \left( \frac{1-\omega}{\omega} D + U \right) x^{(k)} + b$$

Finally, we obtain the matrix

$$\begin{aligned} T = T_\omega &= \left( \frac{D}{\omega} - L \right)^{-1} \left( \frac{1-\omega}{\omega} D + U \right) \\ &= (D - \omega L)^{-1} ((1-\omega)D + \omega U). \end{aligned}$$

The method is called *relaxation method*. We have the following variants:

- $\omega > 1$  overrelaxation (SOR - Successive Over Relaxation)
- $\omega < 1$  subrelaxation

---

**MATLAB Source 4.11 Successive Overrelaxation method (SOR)**

---

```
function [x,ni]=relax(A,b,omega,x0,err,nitmax)
%RELAX Successive overrelaxation (SOR) method
%call [z,ni]=relax(A,b,omega,err,nitmax)
%parameters
%A - system matrix
%b - right hand side vector
%omega - relaxation parameter
%x0 - starting vector
%err - tolerance (default 1e-3)
%nitmax - maximum number of iterations (default 50)
%z - solution
%ni -actual number of iterations

%check parameters
if nargin < 6, nitmax=50; end
if nargin < 5, err=1e-3; end
if nargin < 4, x0=zeros(size(b)); end
if (omega<=0) | (omega>=2)
    error('illegal relaxation parameter')
end
[m,n]=size(A);
if (m~=n) | (n~=length(b))
    error('illegal size')
end
%compute T and c (prepare iterations)
M=1/omega*diag(diag(A))+tril(A,-1);
N=M-A;
T=M\N;
c=M\b;
alfa=norm(T,inf);
x=x0(:);
for i=1:nitmax
    x0=x;
    x=T*x0+c;
    if norm(x-x0,inf)<(1-alfa)/alfa*err
        ni=i;
        return
    end
end
error('iteration number exceeded')
```

---

- $\omega = 1$  Gauss-Seidel

In the sequel, we state two theorems on the convergence of relaxation method.

**Theorem 4.8.5 (Kahan).** *If  $a_{ii} \neq 0$ ,  $i = \overline{1, n}$ ,  $\rho(T_\omega) \geq |\omega - 1|$ . This implies the following necessary conditions  $\rho(T_\omega) < 1 \Rightarrow 0 < \omega < 2$ .*

**Theorem 4.8.6 (Ostrowski-Reich).** *If  $A$  is a positive definite matrix, and  $0 < \omega < 2$ , then SOR converges for any choice of the initial approximation  $x^{(0)}$ .*

**Remark 4.8.7.** For Jacobi (and Gauss-Seidel) method a sufficient condition for convergence is

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \quad (\text{$A$ row diagonal dominant})$$

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ji}| \quad (\text{$A$ column diagonal dominant})$$

◊

The optimal value for  $\omega$  is

$$\omega_O = \frac{2}{1 + \sqrt{1 - (\rho(T_J))^2}}.$$

For an implementation, see MATLAB Source 4.12. In practice, finding the optimal relaxation parameter is inefficient. The function has only a didactic purpose.

---

#### MATLAB Source 4.12 Finding optimal value of relaxation parameter

---

```
function omega=relopt(A)
%RELOPT find optimal value of relaxation parameter
%call omega=relopt(A)
M=diag(diag(A)); %find Jacobi matrix
N=M-A;
T=M\ N;
e=eig(T);
rt=max(abs(e)); %spectral radius of Jacobi matrix
omega=2/(1+sqrt(1-rt^2));
```

---

## 4.9 Applications

### 4.9.1 The finite difference method for linear two-points boundary value problem

Consider the two-point boundary value problem  $y''(x) - p(x)y'(x) - q(x)y(x) = r(x)$  on the interval  $[a, b]$  with boundary conditions  $y(a) = \alpha$ ,  $y(b) = \beta$ . We also assume  $q(x) \geq \underline{q} > 0$ .

This equation may be used to model the heat flow in a long, thin rod, for example. To solve the differential equation numerically, we *discretize* it by seeking its solution only at the evenly spaced mesh point  $x_i = a + ih$ ,  $i = 0, \dots, N + 1$ , where  $h = (b - a)/(N + 1)$  is the mesh spacing. Define  $p_i = p(x_i)$ ,  $r_i = r(x_i)$ , and  $q_i = q(x_i)$ . We need to derive equations to solve for our desired approximations  $y_i \approx y(x_i)$ , where  $y_0 = \alpha$  and  $y_{N+1} = \beta$ .

To derive these equations, we approximate the derivative  $y'(x_i)$  by the following *finite difference approximation*:

$$y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

and the second derivative by

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

Inserting these approximations into the differential equations yields

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - p_i \frac{y_{i+1} - y_{i-1}}{2h} - q_i y_i = r_i, \quad 1 \leq i \leq N.$$

Rewriting this as a linear system we get  $Ay = b$ , where

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}, \quad b = \frac{-h^2}{2} \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix} + \begin{bmatrix} \left(\frac{1}{2} + \frac{h}{4}p_1\right)\alpha \\ 0 \\ \vdots \\ 0 \\ \left(\frac{1}{2} - \frac{h}{4}p_1\right)\beta \end{bmatrix},$$

and

$$A = \begin{bmatrix} a_1 & -c_1 & & & \\ -b_2 & \ddots & \ddots & & \\ & \ddots & \ddots & -c_{N-1} & \\ & & -b_N & a_N & \end{bmatrix}, \quad \begin{aligned} a_i &= 1 + \frac{h^2}{2}q_i, \\ b_i &= \frac{1}{2}\left(1 + \frac{h}{2}p_i\right), \\ c_i &= \frac{1}{2}\left(1 - \frac{h}{2}p_i\right). \end{aligned}$$

Note that  $a_i > 0$ , and also  $b_i > 0$  and  $c_i > 0$ , if  $h$  is small enough.

This is a nonsymmetric *tridiagonal* system to solve for  $y$ . We will show how to change it to a symmetric positive definite tridiagonal system, so that we may use *band Cholesky* to solve it.

Choose

$$D = \text{diag}\left(1, \sqrt{\frac{c_1}{b_2}}, \sqrt{\frac{c_1 c_2}{b_2 b_3}}, \dots, \sqrt{\frac{c_1 c_2 \dots c_{N-1}}{b_2 b_3 \dots b_N}}\right).$$

Then we may change  $Ay = b$  to  $(DAD^{-1})(Dy) = Db$  or  $\tilde{A}\tilde{y} = \tilde{b}$ , where

$$\tilde{A} = \begin{bmatrix} a_1 & -\sqrt{c_1 b_2} & & & \\ -\sqrt{c_1 b_2} & a_2 & -\sqrt{c_2 b_3} & & \\ & -\sqrt{c_2 b_3} & \ddots & \ddots & \\ & & \ddots & \ddots & -\sqrt{c_{N-1} b_N} \\ & & & -\sqrt{c_{N-1} b_N} & a_N \end{bmatrix}.$$

It is easy to see that  $\tilde{A}$  is symmetric, and it has the same eigenvalues of  $A$  because  $\tilde{A}$  and  $A$  are similar. By Gershgorin's Theorem<sup>7</sup> eigenvalues of  $A$  lie inside the disk centered at  $1 + h^2 q_i \geq 1 + h^2 q/2$  with radius 1; in particular, they must all have positive real parts. Since  $A$  is symmetric, its eigenvalues are real and hence positive, so  $\tilde{A}$  is positive definite. Its smallest eigenvalue is bounded below by  $h^2 q/2$ . Thus, it can be solved by Cholesky method.

For implementations of both methods, see MATLAB Sources 4.13 and 4.14, respectively.

---

**MATLAB Source 4.13 Two point boundary value problem – finite difference method**


---

```
function [x,y]=bilocal(p,q,r,a,b,alpha,beta,N)
%BILOCAL - two-point boundary value problem
%y''(x)-p(x)y'(x)-q(x)y(x)=r(x), x in [a,b]
%y(a)=alpha, y(b)=beta
%call Y=BILOCAL(P,Q,R,A,B,ALPHA,BETA,N)
%P,Q,R - functions
%[A,B] - interval
%alpha,beta - values at endpoints
%N - no. of points

h=(b-a)/(N+1); x=a+[1:N]*h;
vp=p(x); vr=r(x); vq=q(x);
av=1+h^2/2*vq;
bv=1/2*(1+h/2*vp);
cv=1/2*(1-h/2*vp);
B=[[-bv(2:end);0],av,[0;-cv(1:end-1)]];
A=spdiags(B, [-1:1], N, N);
bb=-h^2/2*vr;
bb(1)=bb(1)+(1/2+h/4*vp(1))*alpha;
bb(N)=bb(N)+(1/2-h/4*vp(N))*beta;
y=A\bb;
x=[a;x;b];
y=[alpha;y;beta];
```

---

We test both methods for the problem

$$y'' = -\frac{2}{x}y' + \frac{2}{x^2}y + \frac{\sin(\ln x)}{x^2}, \quad x \in [1, 2], \quad y(1) = 1, \quad y(2) = 2,$$

with exact solution

$$y = c_1 x + \frac{c_2}{x^2} - \frac{3}{10} \sin(\ln x) - \frac{1}{10} \cos(\ln x),$$

---

<sup>7</sup>Gershgorin's Theorem has the following statement: Let  $B$  be an arbitrary matrix. Then the eigenvalues  $\lambda$  of  $B$  are located in the union of the  $n$  disks

$$|\lambda - b_{kk}| \leq \sum_{j \neq k} |b_{kj}|.$$

---

**MATLAB Source 4.14** Two point boundary value problem – finite difference method and symmetric matrix
 

---

```

function [x,y]=bilocalsim(p,q,r,a,b,alfa,beta,N)
%BILOCALSIM - two-point boundary value problem
%y''(x)-p(x)y'(x)-q(x)y(x)=r(x), x in [a,b]
%y(a)=alpha, y(b)=beta
%call Y=BILOCALSIM(P,Q,R,A,B,ALPHA,BETA,N)
%P,Q,R - functions
%[A,B] - interval
%alpha,beta - values at endpoints
%N - no. of points
%transform the matrix into a symmetric positive one

h=(b-a)/(N+1); x=a+[1:N]'*h;
vp=p(x); vr=r(x); vq=q(x);
av=1+h^2/2*vq;
bv=1/2*(1+h/2*vp);
cv=1/2*(1-h/2*vp);
dd=-sqrt(cv(1:end-1).*bv(2:end));
B=[[dd;0],av,[0;dd]];
A=spdiags(B,[-1:1],N,N);
bb=-h^2/2*vr;
bb(1)=bb(1)+(1/2+h/4*vp(1))*alfa;
bb(N)=bb(N)+(1/2-h/4*vp(N))*beta;
%Cholesky method
R=chol(A);
D=diag(sqrt(cumprod([1;cv(1:end-1)./bv(2:end)])));
y=D\ (R\ (R'\ (D*bb)));
x=[a;x;b];
y=[alfa;y;beta];

```

---

where

$$\begin{aligned}
 c_2 &= \frac{1}{70}[8 - 12 \sin(\ln 2) - 4 \cos(\ln 2)], \\
 c_1 &= \frac{11}{10} - c_2.
 \end{aligned}$$

Here is the code. It calls the methods and tabulate the computed solutions and the exact solution.

```

p=@(x) -2./x;
q=@(x) 2./x.^2;
r=@(x) sin(log(x))./x.^2;
a=1; b=2;
alpha=1; bet=2;
N=9;
[x,y]=bilocal(p,q,r,a,b,alfa,bet,N);

```

```
[x2,y2]=bilocalsim(p,q,r,a,b,alpha,bet,N);
%Exact solution
c2=1/70*(8-12*sin(log(2))-4*cos(log(2)));
c1=11/10-c2;
ye=c1*x+c2./x.^2-3/10*sin(log(x))-1/10*cos(log(x));
disp([x,y,y2, ye])
```

and the output

1.0000	1.0000	1.0000	1.0000
1.1000	1.0926	1.0926	1.0926
1.2000	1.1870	1.1870	1.1871
1.3000	1.2833	1.2833	1.2834
1.4000	1.3814	1.3814	1.3814
1.5000	1.4811	1.4811	1.4812
1.6000	1.5824	1.5824	1.5824
1.7000	1.6850	1.6850	1.6850
1.8000	1.7889	1.7889	1.7889
1.9000	1.8939	1.8939	1.8939
2.0000	2.0000	2.0000	2.0000

See also Problem 4.7.

### 4.9.2 Computing a plane truss

This example is adapted after [66].

Figure 4.3 depicts a plane truss having 21 members (the numbered lines) connecting 12 joints (the numbered circles). The indicating loads, in tons, are applied at joints 2, 5, 6, 9, and 10, and we want to determine the resulting force on each member of the truss.

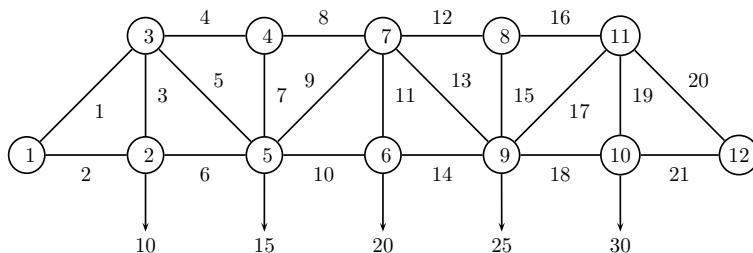


Figure 4.3: A plane truss

For the truss to be in static equilibrium, there must be no net force, horizontally or vertically, at any joint. Thus, we can determine the member forces, by equating the horizontal forces to the left and right at each joint, and similarly equating the vertical forces upward and downward at each joint. For the 12 joints, this would give 24 equations, which is more than the 21 unknowns factors to be determined. For the truss to be statically determined, that

is, for there to be a unique solution, we assume that joint 1 is rigidly fixed both horizontally and vertically and that joint 12 is fixed vertically. Resolving the number of forces into horizontal and vertical components and defining  $\alpha = 1/\sqrt{2}$ , we obtain the following system of equations for the member forces  $f_i$ :

Joint 2	$f_2 = f_6$ $f_3 = 10$	Joint 3	$\alpha f_1 = f_4 + \alpha f_5$ $\alpha f_1 + f_3 + \alpha f_5 = 0$
Joint 4	$f_4 = f_8$ $f_7 = 0$	Joint 5	$\alpha f_5 + f_6 = \alpha f_9 + f_{10}$ $\alpha f_5 + f_7 + \alpha f_9 = 15$
Joint 6	$f_{10} = f_{14}$ $f_{11} = 20$	Joint 7	$f_8 + \alpha f_9 = f_{12} + \alpha f_{13}$ $\alpha f_9 + f_{11} + \alpha f_{13} = 0$
Joint 8	$f_{12} = f_{16}$ $f_{15} = 0$	Joint 9	$\alpha f_{13} + f_{14} = \alpha f_{17} + f_{18}$ $\alpha f_{13} + f_{15} + \alpha f_{17} = 25$
Joint 10	$f_{18} = f_{21}$ $f_{19} = 30$	Joint 11	$f_{16} + \alpha f_{17} = f_{20}$ $\alpha f_{17} + f_{19} + \alpha f_{20} = 0$
Joint 12	$\alpha f_{20} + f_{21} = 0$		

We give here only a fragment of MATLAB code. You can download the whole code from author's web page (file `mytruss2.m`).

```
% MYTRUSS2    Solution to the truss problem.
```

```
n = 21;
A = zeros(n,n);
b = zeros(n,1);
alpha = 1/sqrt(2);

% Joint 2: f2 = f6
%           f3 = 10
A(1,2) = 1;
A(1,6) = -1;
A(2,3) = 1;
b(2) = 10;

% Joint 3: alpha f1 = f4 + alpha f5
%           alpha f1 + f3 + alpha f5 = 0
A(3,1) = alpha;
A(3,4) = -1;
A(3,5) = -alpha;
A(4,1) = alpha;
A(4,3) = 1;
A(4,5) = alpha;

% Joint 4: f4 = f8
%           f7 = 0
A(5,4) = 1;
A(5,8) = -1;
A(6,7) = 1;
b(6) = 0;
```

```
%Joints 5,...,11
% Joint 12 alpha f20+f21=0;
A(21,20)=alpha;
A(21,21)=1;

x = A\b;
```

Figure 4.4 gives a plot of the truss with the force members.

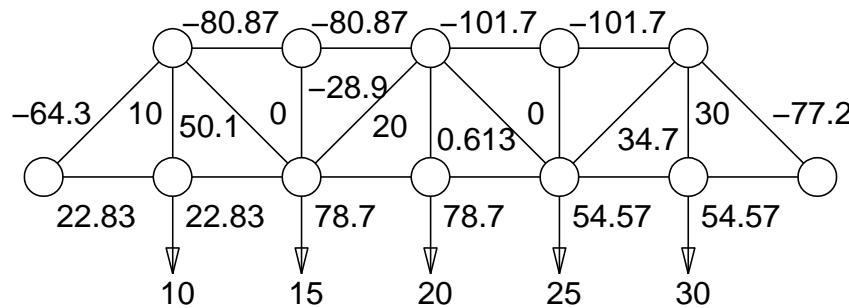


Figure 4.4: The solution of plane truss problem

## Problems

**Problem 4.1.** For a system with a tridiagonal matrix implement the following methods:

- (a) Gaussian elimination, with and without pivoting.
- (b) LU decomposition.
- (c) LUP decomposition.
- (d) Cholesky decomposition for a symmetric, positive definite matrix.
- (e) Jacobi method.
- (f) Gauss-Seidel method.
- (g) SOR method.

**Problem 4.2.** Implement Gaussian elimination with partial pivoting in two variants: with logical line permutation (using a permutation vector) and with physical line permutation. Compare execution time for various system matrix sizes. Do the same for LUP decomposition.

**Problem 4.3.** Modify the LUP decomposition function to return the determinant of the initial matrix.

**Problem 4.4.** Consider the system

$$\begin{aligned} 2x_1 - x_2 &= 1 \\ -x_{j-1} + 2x_j - x_{j+1} &= j, \quad j = \overline{2, n-1} \\ -x_{n-1} + 2x_n &= n. \end{aligned}$$

- (a) Generate its matrix using `diag`.
- (b) Solve it using LU decomposition.
- (c) Solve it using a suitable function in Problem 4.1.
- (d) Generate its matrix using `spdiags`, solve it using `\`, and compare the run time with the required run time for the same system, but with a dense matrix.
- (e) Estimate the condition number of the system using `condest`.

**Problem 4.5.** Modify Gaussian elimination and LUP decomposition to allow total pivoting.

**Problem 4.6.** Write a MATLAB function for the generation of random diagonal-dominant band matrix of given size. Test Jacobi and SOR methods on system having such matrices.

**Problem 4.7.** Apply the idea in Section 4.9.1 two solve the univariate Poisson equation

$$-\frac{d^2v(x)}{dx^2} = f, \quad 0 < x < 1,$$

with boundary conditions  $v(0) = v(1) = 0$ . Solve the system of discretized problem by Cholesky and SOR method.

**Problem 4.8.** Find the Gauss-Seidel method matrix corresponding to the matrix

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & \ddots & \ddots & \ddots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}.$$

**Problem 4.9.** A finite element analysis for the load of a structure yields the following system

$$\begin{bmatrix} \alpha & 0 & 0 & 0 & \beta & -\beta \\ 0 & \alpha & 0 & -\beta & 0 & -\beta \\ 0 & 0 & \alpha & \beta & \beta & 0 \\ 0 & -\beta & \beta & \gamma & 0 & 0 \\ \beta & 0 & \beta & 0 & \gamma & 0 \\ -\beta & -\beta & 0 & 0 & 0 & \gamma \end{bmatrix} x = \begin{bmatrix} 15 \\ 0 \\ -15 \\ 0 \\ 25 \\ 0 \end{bmatrix},$$

where  $\alpha = 482317$ ,  $\beta = 2196.05$  and  $\gamma = 6708.43$ . Here,  $x_1, x_2, x_3$  are the side displacements, and  $x_4, x_5, x_6$  are (tridimensional) rotational displacement corresponding to the applied force (the right-hand side).

- (a) Find  $x$ .
- (b) How accurate is the computing? Suppose first exact input data, then a relative error in input data of  $\|\Delta A\|/\|A\| = 5 \times 10^{-7}$ .

**Problem 4.10.** Consider the system

$$\begin{aligned} x_1 + x_2 &= 2 \\ 10x_1 + 10^{18}x_2 &= 10 + 10^{18}. \end{aligned}$$

- (a) Solve it by Gaussian elimination with partial pivoting.
- (b) Divide each line by maximum in modulus for that line, and then apply the Gaussian elimination.
- (c) Solve the system using Symbolic Math Toolbox.



# CHAPTER 5

---

## Function Approximation

---

The function to be approximated can be defined:

- On a continuum (typically a finite interval) – special functions (elementary or transcendental) that one wishes to evaluate as part of a subroutine.
- On a finite set of points – instance frequently encountered in the physical sciences when measurements are taken of a certain physical quantity as a function of other physical quantity (such as time).

In either case one wants to approximate the given function “as well as possible” in terms of other simpler functions. Since such an evaluation must be reduced to a finite number of arithmetical operations, the simpler functions should be polynomial or rational functions.

The general scheme of approximation can be described as:

- A given function  $f \in X$  to be approximated.
- A class  $\Phi$  of “approximations”.
- A “norm”  $\|\cdot\|$  measuring the overall magnitude of functions.

We are looking for an approximation  $\hat{\varphi} \in \Phi$  of  $f$  such that

$$\|f - \hat{\varphi}\| \leq \|f - \varphi\| \text{ for all } \varphi \in \Phi. \quad (5.0.1)$$

This problem is called *best approximation problem* of  $f$  from the class  $\Phi$ , and the function  $\hat{\varphi}$  is called *best approximation element* of  $f$ , relative to the norm  $\|\cdot\|$ .

Given a basis  $\{\pi_j\}_{j=1}^n$  of  $\Phi$ , we can express a  $\varphi \in \Phi$  and  $\Phi$  as

$$\Phi = \Phi_n = \left\{ \varphi : \varphi(t) = \sum_{j=1}^n c_j \pi_j(t), c_j \in \mathbb{R} \right\}. \quad (5.0.2)$$

$\Phi$  is a finite-dimensional linear space or a proper subset of it.

**Example 5.0.1.**  $\Phi = \mathbb{P}_m$  - the set of polynomials of degree at most  $m$ . A basis of this space is  $e_j(t) = t^j$ ,  $j = 0, 1, \dots, m$ . So,  $\dim \mathbb{P}_m = m + 1$ . Polynomials are the most frequently used “general-purpose” approximations for dealing with functions on bounded domains (finite intervals or finite set of points). One reason – Weierstrass’ theorem – any function from  $C[a, b]$  can be approximated on a finite interval as closely as one wishes by a polynomial of sufficiently high degree.  $\diamond$

**Example 5.0.2.**  $\Phi = \mathbb{S}_m^k(\Delta)$  the space of polynomial spline functions of degree  $m$  and smoothness class  $k$  on the subdivision

$$\Delta : a = t_1 < t_2 < t_3 < \dots < t_{N-1} < t_N = b$$

of the interval  $[a, b]$ . These are piecewise polynomials of degree  $\leq m$ , pieced together at the “joints”  $t_1, \dots, t_{N-1}$ , in such a way that all derivatives up to and including the  $k$ th are continuous on the whole interval  $[a, b]$  including the joints. We assume  $0 \leq k < m$ . For  $k = m$  this space equates  $\mathbb{P}_m$ . We set  $k = -1$  if we allow discontinuities at the joints.  $\diamond$

**Example 5.0.3.**  $\Phi = \mathbb{T}_m[0, 2\pi]$  the space of trigonometric polynomials of degree  $\leq m$  on  $[0, 2\pi]$ . This are linear combinations of the functions

$$\begin{aligned}\pi_k(t) &= \cos(k-1)t & k = \overline{1, m+1}, \\ \pi_{m+1-k}(t) &= \sin kt & k = \overline{1, m}.\end{aligned}$$

The dimension of this space is  $n = 2m + 1$ . Such approximations are natural choices when the function  $f$  to be approximated is periodic with period  $2\pi$ . (If  $f$  has period  $p$ , one makes a change of variables  $t \mapsto tp/2\pi$ ).  $\diamond$

The class of rational functions

$$\Phi = \mathbb{R}_{r,s} = \{\varphi : \varphi = p/q, p \in \mathbb{P}_r, q \in \mathbb{P}_s\},$$

is *not* a linear space.

Possible choice of norms – both for continuous and discrete functions – and the approximation they generate are summarized in Table 5.1. The continuous case involve an interval  $[a, b]$  and a weight function  $w(t)$  (possibly  $w(t) \equiv 1$ ) defined on  $[a, b]$  and positive except for isolate zeros. The discrete case involve a set of  $N$  distinct points  $t_1, t_2, \dots, t_N$  along with positive weight factors  $w_1, w_2, \dots, w_N$  (possibly  $w_i = 1, i = \overline{1, N}$ ). The interval  $[a, b]$  may be unbounded if the weight function  $w$  is such that the improper integral extended over  $[a, b]$ , which defines the norm makes sense.

Hence, we may take any one of the norms in Table 5.1 and combine it with any of the preceding linear spaces  $\Phi$  to arrive at a meaningful best approximation problem (5.0.1). In the continuous case, the given function  $f$  and the functions  $\varphi \in \Phi$  must be defined on  $[a, b]$  and such that the norm  $\|f - \varphi\|$  makes sense. Likewise,  $f$  and  $\varphi$  must be defined at the points  $t_i$  in the discrete case.

continuous norm	type	discrete norm
$\ u\ _\infty = \max_{a \leq t \leq b}  u(t) $	$L^\infty$	$\ u\ _\infty = \max_{1 \leq i \leq N}  u(t_i) $
$\ u\ _{1,w} = \int_a^b  u(t) w(t) dt$	$L_w^1$	$\ u\ _{1,w} = \sum_{i=1}^n w_i  u(t_i) $
$\ u\ _{2,w} = \left( \int_a^b  u(t) ^2 w(t) dt \right)^{1/2}$	$L_w^2$	$\ u\ _{2,w} = \left( \sum_{i=1}^N w_i  u(t_i) ^2 \right)^{1/2}$

Table 5.1: Types of approximations and associated norms

Note that if the best approximant  $\hat{\varphi}$  in the discrete case is such that  $\|f - \hat{\varphi}\| = 0$ , then  $\hat{\varphi}(t_i) = f(t_i)$ , for  $i = 1, 2, \dots, N$ . We then say that  $\hat{\varphi}$  interpolates  $f$  at the points  $t_i$  and we refer to this kind of approximation as an *interpolation problem*.

The simplest approximation problems are the least squares problem and the interpolation problem and the easiest space is the space of polynomials.

Before we start with the least square problem we introduce a notational device (as in [33]) that allows us to treat the continuous and the discrete case simultaneously. We define in the continuous case

$$\lambda(t) = \begin{cases} 0, & \text{if } t < a \text{ (when } -\infty < a), \\ \int_a^t w(\tau) d\tau, & \text{if } a \leq t \leq b, \\ \int_a^b w(\tau) d\tau, & \text{if } t > b \text{ (when } b < \infty). \end{cases} \quad (5.0.3)$$

then we can write, for any continuous function  $u$

$$\int_{\mathbb{R}} u(t) d\lambda(t) = \int_a^b u(t) w(t) dt, \quad (5.0.4)$$

since  $d\lambda(t) \equiv 0$  outside  $[a, b]$  and  $d\lambda(t) = w(t) dt$  inside. We call  $d\lambda$  a continuous (positive) measure. The discrete measure (also called “Dirac measure”) associated to the point set  $\{t_1, t_2, \dots, t_N\}$  is a measure  $d\lambda$  that is nonzero only at the points  $t_i$  and has the value  $w_i$  there. Thus in this case

$$\int_{\mathbb{R}} u(t) d\lambda(t) = \sum_{i=1}^N w_i u(t_i). \quad (5.0.5)$$

A more precise definition can be given in terms of Stieltjes integrals, if we define  $\lambda(t)$  to be a step function having the jump  $w_i$  at  $t_i$ . In particular, we can define the  $L^2$  norm as

$$\|u\|_{2,d\lambda} = \left( \int_{\mathbb{R}} |u(t)|^2 d\lambda(t) \right)^{\frac{1}{2}} \quad (5.0.6)$$

and obtain the continuous or the discrete norm depending on whether  $\lambda$  is taken to be as in (5.0.3) or a step function as in (5.0.5).

We call the *support* of  $d\lambda$  – denoted by  $\text{supp } d\lambda$  – the interval  $[a, b]$  in the continuous case (assuming  $w$  is positive on  $[a, b]$  except for isolated zeros) and the set  $\{t_1, t_2, \dots, t_N\}$

in the discrete case. We say that the set of functions  $\pi_j$  in (5.0.2) is linearly independent on  $\text{supp } d\lambda$  if

$$\forall t \in \text{supp } d\lambda \quad \sum_{j=1}^n c_j \pi_j(t) \equiv 0 \Rightarrow c_1 = c_2 = \dots = c_k = 0 \quad (5.0.7)$$

## 5.1 Least Squares approximation

We specialize the best approximation problem (5.0.1) by taking as norm the  $L^2$  norm

$$\|u\|_{2, d\lambda} = \left( \int_{\mathbb{R}} |u(t)|^2 d\lambda(t) \right)^{\frac{1}{2}}, \quad (5.1.1)$$

where  $d\lambda$  is either a continuous measure (cf. (5.0.3)) or a discrete measure (cf. (5.0.5)) and using approximants  $\varphi$  from an  $n$ -dimensional linear space

$$\Phi = \Phi_n = \left\{ \varphi : \varphi(t) = \sum_{j=1}^n c_j \pi_j(t), c_j \in \mathbb{R} \right\}. \quad (5.1.2)$$

$\pi_j$  linearly independent on  $\text{supp } d\lambda$ ; the integral in (5.1.1) is meaningful whenever  $u = \pi_j$ ,  $j = 1, \dots, n$  or  $u = f$ .

The specialized problem is called *least squares approximation problem* or *square mean approximation problem*. His solution (beginning of the 19th century) is due to Gauss and Legendre <sup>1</sup>.

### 5.1.1 Inner products

Given a discrete or continuous measure  $d\lambda$ , and given any two function  $u$  and  $v$  having a finite norm (5.0.1), we can define the inner (scalar) product

$$(u, v) = \int_{\mathbb{R}} u(t)v(t) d\lambda(t). \quad (5.1.3)$$

The Cauchy-Buniakovski-Schwarz inequality

$$\|(u, v)\| \leq \|u\|_{2, d\lambda} \|v\|_{2, d\lambda}$$

tells us that the integral in (5.1.3) is well defined.

A real inner product has the following properties:



Adrien Marie Legendre (1752-1833) was a French mathematicians active in Paris, best known for his treatise on elliptic integrals, but also famous for his work in number theory and geometry. He is considered the originator (in 1805) of the method of least squares, although Gauss had already used it in 1794, but published it only in 1809.

- (i) symmetry  $(u, v) = (v, u)$ ;
- (ii) homogeneity  $(\alpha u, v) = \alpha(u, v)$ ,  $\alpha \in \mathbb{R}$ ;
- (iii) additivity  $(u + v, w) = (u, w) + (v, w)$ ;
- (iv) positive definiteness  $(u, u) \geq 0$  and  $(u, u) = 0 \Leftrightarrow u \equiv 0$  on  $\text{supp } d\lambda$ .

(i)+(ii)  $\Rightarrow$  linearity

$$(\alpha_1 u_1 + \alpha_2 u_2, v) = \alpha_1(u_1, v) + \alpha_2(u_2, v) \quad (5.1.4)$$

(5.1.4) extends to finite linear combinations. Also

$$\|u\|_{2, d_\lambda}^2 = (u, u). \quad (5.1.5)$$

We say that  $u$  and  $v$  are *orthogonal* if

$$(u, v) = 0. \quad (5.1.6)$$

More generally, we may consider an *orthogonal system*  $\{u_k\}_{k=1}^n$ :

$$(u_i, u_j) = 0 \text{ if } i \neq j, \quad u_k \neq 0 \text{ on } \text{supp } d\lambda; \quad i, j = \overline{1, n}, \quad k = \overline{1, n}. \quad (5.1.7)$$

For such a system we have the Generalized Theorem of Pythagoras

$$\left\| \sum_{k=1}^n \alpha_k u_k \right\|^2 = \sum_{k=1}^n |\alpha_k|^2 \|u_k\|^2. \quad (5.1.8)$$

(5.1.8) implies that every orthogonal system is linearly independent on  $\text{supp } d\lambda$ . Indeed, if the left-hand side of (5.1.8) vanishes, then so does the right-hand side, and this, since  $\|u_k\|^2 > 0$ , by assumption, implies  $\alpha_1 = \alpha_2 = \dots = \alpha_n = 0$ .

## 5.1.2 The normal equations

By (5.1.5) we can write the square of  $L^2$  error in the form

$$E^2[\varphi] := \|\varphi - f\|^2 = (\varphi - f, \varphi - f) = (\varphi, \varphi) - 2(\varphi, f) + (f, f).$$

Inserting  $\varphi$  from (5.1.2) gives

$$\begin{aligned} E^2[\varphi] &= \int_{\mathbb{R}} \left( \sum_{j=1}^n c_j \pi_j(t) \right)^2 d\lambda(t) - 2 \int_{\mathbb{R}} \left( \sum_{j=1}^n c_j \pi_j(t) \right) f(t) d\lambda(t) + \\ &\quad + \int_{\mathbb{R}} f^2(t) d\lambda(t). \end{aligned} \quad (5.1.9)$$

The squared  $L^2$  error is a quadratic function of the coefficients  $c_1, \dots, c_n$  of  $\varphi$ . The problem of best  $L^2$  approximation thus amounts to minimizing this quadratic function; one solves it by vanishing the partial derivatives. One obtains

$$\frac{\partial}{\partial c_i} E^2[\varphi] = 2 \int_{\mathbb{R}} \left( \sum_{j=1}^n c_j \pi_j(t) \right) \pi_i(t) d\lambda(t) - 2 \int_{\mathbb{R}} \pi_i(t) f(t) d\lambda(t) = 0,$$

that is,

$$\sum_{j=1}^n (\pi_i, \pi_j) c_j = (\pi_i, f), \quad i = 1, 2, \dots, n. \quad (5.1.10)$$

These are called *normal equations* for the least squares problem. They form a system having the form

$$Ac = b, \quad (5.1.11)$$

where the matrix  $A$  and the vector  $b$  have elements

$$A = [a_{ij}], \quad a_{ij} = (\pi_i, \pi_j), \quad b = [b_i], \quad b_i = (\pi_i, f). \quad (5.1.12)$$

By symmetry of the inner product,  $A$  is a symmetric matrix. Moreover,  $A$  is positive definite; that is

$$x^T Ax = \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j > 0 \text{ if } x \neq [0, 0, \dots, 0]^T. \quad (5.1.13)$$

The quadratic function in (5.1.13) is called a *quadratic form* (since it is homogeneous of degree 2). The positive definiteness of  $A$  says that the quadratic form whose coefficients are the elements of  $A$  is always nonnegative, and it is zero only if all variables  $x_i$  vanish.

To prove (5.1.13), all we have to do is insert the definition of  $a_{ij}$  and to use the property (i)-(iv) of the inner product

$$x^T Ax = \sum_{i=1}^n \sum_{j=1}^n x_i x_j (\pi_i, \pi_j) = \sum_{i=1}^n \sum_{j=1}^n (x_i \pi_i, x_j \pi_j) = \left\| \sum_{i=1}^n x_i \pi_i \right\|^2.$$

This is clearly nonnegative. It is zero only if  $\sum_{i=1}^n x_i \pi_i \equiv 0$  on  $\text{supp } d\lambda$ , which, by the assumption of linear independence of the  $\pi_i$ , implies  $x_1 = x_2 = \dots = x_n = 0$ .

It is a well-known fact of linear algebra that a symmetric positive definite matrix  $A$  is nonsingular. Indeed, its determinant, as well as its leading principal minor determinants are strictly positive. It follows that the system (5.1.10) of normal equation has a unique solution. Does this solution correspond to a minimum of  $E[\varphi]$  in (5.1.9)? The hessian matrix  $H = [\partial^2 E^2 / \partial c_i \partial c_j]$  has to be positive definite. But  $H = 2A$ , since  $E^2$  is a quadratic function. Therefore,  $H$ , with  $A$ , is indeed positive definite, and the solution of the normal equations gives us the desired minimum. The least squares approximation problem thus has a unique solution, given by

$$\hat{\varphi}(t) = \sum_{j=1}^n \hat{c}_j \pi_j(t), \quad (5.1.14)$$

where  $\hat{c} = [\hat{c}_1, \hat{c}_2, \dots, \hat{c}_n]^T$  is the solution of the normal equation (5.1.10).

This completely settles the least square approximation problem in theory. How in practice? For a general set of linearly independent basis function, we can see the following difficulties.

(1) The system (5.1.10) may be *ill-conditioned*. A classical example is provided by  $\text{suppd}\lambda = [0, 1]$ ,  $d\lambda(t) = dt$  on  $[0, 1]$  and  $\pi_j(t) = t^{j-1}$ ,  $j = 1, 2, \dots, n$ . Then

$$(\pi_i, \pi_j) = \int_0^1 t^{i+j-2} dt = \frac{1}{i+j-1}, \quad i, j = 1, 2, \dots, n,$$

that is  $A$  is precisely the Hilbert matrix. The resulting severe ill-conditioning of the normal equations is entirely due to an unfortunate choice of the basis function. These become almost linearly dependent, as the exponent grows. Another source of degradation lies in the element  $b_j = \int_0^1 \pi_j(t)f(t)dt$  of the right-hand side vector. When  $j$  is large  $\pi_j(t) = t^{j-1}$  behaves on  $[0, 1]$  like a discontinuous function. A polynomial  $\pi_j$  that oscillates rapidly on  $[0, 1]$  would seem to be preferable from this point of view, since it would "engage" more vigorously the function  $f$  over all the interval  $[0, 1]$ , in contrast to a canonical monomial which shoots from almost zero to 1 at the right endpoint.

(2) The second disadvantage is that all the coefficients  $\hat{c}_j$  in (5.1.14) depends on  $n$ , i.e.  $\hat{c}_j = \hat{c}_j^{(n)}$ ,  $j = 1, 2, \dots, n$ . Increasing  $n$  will give an enlarged system of normal equations with a completely new solution vector. We refer to this as the *nonpermanence* of the coefficients  $\hat{c}_j$ .

Both defects (1) and (2) can be eliminated (or at least attenuated) by choosing for the basis functions  $\pi_j$  an orthogonal system,

$$(\pi_i, \pi_j) = 0 \text{ if } i \neq j \quad (\pi_i, \pi_j) = \|\pi_j\|^2 > 0 \quad (5.1.15)$$

Then the system of normal equations becomes diagonal and is solved immediately by

$$\hat{c}_j = \frac{(\pi_j, f)}{(\pi_i, \pi_j)}, \quad j = 1, 2, \dots, n. \quad (5.1.16)$$

Clearly, each of these coefficients  $\hat{c}_j$  are independent of  $n$  and once computed, they remain the same for any larger  $n$ . We now have *permanence* of the coefficients. We must not solve a system of normal equations, but instead we can use the formula (5.1.16) directly.

Any system  $\{\hat{\pi}_j\}$  that is linearly independent on  $\text{suppd}\lambda$  can be orthogonalized with respect to the measure  $d\lambda$  by the *Gram-Schmidt procedure*. One takes

$$\pi = \hat{\pi}_1$$

and then, for  $j = 2, 3, \dots$  recursively computes

$$\pi_j = \hat{\pi}_j - \sum_{k=1}^{j-1} c_k \pi_k, \quad c_k = \frac{(\hat{\pi}_j, \pi_k)}{(\pi_k, \pi_k)}, \quad k = 1, j-1.$$

Then each  $\pi_j$  so determined is orthogonal to all preceding ones.

### 5.1.3 Least squares error; convergence

We have seen that if  $\Phi = \Phi_n$  consists of  $n$  functions  $\pi_j$ ,  $j = 1, 2, \dots, n$  which are linearly independent on  $\text{supp } d\lambda$ , then the least squares problem for  $d\lambda$

$$\min_{\varphi \in \Phi_n} \|f - \varphi\|_{2,d\lambda} = \|f - \hat{\varphi}\|_{2,d\lambda} \quad (5.1.17)$$

has a unique solution  $\hat{\varphi} = \hat{\varphi}_n$ , given by (5.1.14). There are many ways to select a basis  $\{\pi_j\}$  in  $\Phi_n$  and, therefore, many ways the solution  $\hat{\varphi}_n$  be represented. Nevertheless, is always one and the same function. The least squares error – the quantity on the right of (5.1.17) – is independent of the choice of basis functions (although the calculation of the least square solution, as mentioned previously, is not). In studying this error we may assume, without restricting generality, that the basis  $\pi_j$  is an orthogonal system. (Every linear independent system can be orthogonalized by the Gram-Schmidt orthogonalization procedure). Then we have (cf. (5.1.16))

$$\hat{\varphi}_n(t) = \sum_{j=1}^n \hat{c}_j \pi_j(t), \quad \hat{c}_j = \frac{(\pi_j, f)}{(\pi_j, \pi_j)}. \quad (5.1.18)$$

We first note that the error  $f - \varphi_n$  is orthogonal to the space  $\Phi_n$ ; that is

$$(f - \hat{\varphi}_n, \varphi) = 0, \quad \forall \varphi \in \Phi_n \quad (5.1.19)$$

where the inner product is the one in (5.1.3). Since  $\varphi$  is a linear combination of the  $\pi_k$ , it suffices to show (5.1.19) for each  $\varphi = \pi_k$ ,  $k = 1, 2, \dots, n$ . Inserting  $\hat{\varphi}_n$  from (5.1.18) in the left of (5.1.19), we find indeed

$$(f - \hat{\varphi}_n, \pi_k) = \left( f - \sum_{j=1}^n \hat{c}_j \pi_j, \pi_k \right) = (f, \pi_k) - \hat{c}_k (\pi_k, \pi_k) = 0,$$

the last equation following from the formula for  $\hat{c}_k$  in (5.1.18). The result (5.1.19) has a simple geometric interpretation. If we picture functions as vectors, and the space  $\Phi_n$  as a plane, then for any function  $f$  that “sticks out” of the plane  $\Phi_n$ , the least square approximant  $\hat{\varphi}_n$  is the *orthogonal projection* of  $f$  onto  $\Phi_n$ ; see Figure 5.1.

In particular, choosing  $\varphi = \hat{\varphi}_n$  in (5.1.19), we get

$$(f - \hat{\varphi}_n, \hat{\varphi}_n) = 0$$

and, therefore, since  $f = (f - \hat{\varphi}) + \hat{\varphi}$ , by the theorem of Pythagoras and its generalization (5.1.8)

$$\begin{aligned} \|f\|^2 &= \|f - \hat{\varphi}\|^2 + \|\hat{\varphi}\|^2 = \|f - \hat{\varphi}_n\|^2 + \left\| \sum_{j=1}^n \hat{c}_j \pi_j \right\|^2 \\ &= \|f - \hat{\varphi}_n\|^2 + \sum_{j=1}^n |\hat{c}_j|^2 \|\pi_j\|^2. \end{aligned}$$

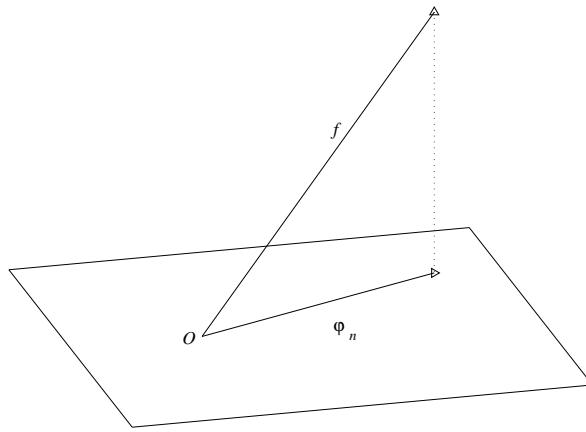


Figure 5.1: Least square approximation as orthogonal projection

Solving for the first term on the right, we get

$$\|f - \hat{\varphi}_n\| = \left\{ \|f\|^2 - \sum_{j=1}^n |\hat{c}_j| \|\pi_j\|^2 \right\}^{1/2}, \quad \hat{c}_j = \frac{(\pi_j, f)}{(\pi_j, \pi_j)}. \quad (5.1.20)$$

Note that the expression in braces must necessarily be nonnegative.

The formula (5.1.20) is interesting theoretically, but for limited practical use. Note, indeed, that as the error approaches the level of the machine precision  $\text{eps}$ , computing the error from the right-hand side of (5.1.20) cannot produce anything smaller than  $\sqrt{\text{eps}}$  because of inevitable rounding errors committed during the subtraction in the radicand. (They may even produce a negative result for the radicand.) Using instead the definition,

$$\|f - \hat{\varphi}_n\| = \left\{ \int_{\mathbb{R}} [f(t) - \hat{\varphi}_n(t)]^2 d\lambda(t) \right\}^{1/2},$$

along, perhaps, with a suitable (positive) quadrature rule, it is guaranteed to produce a non-negative result that may potentially be as small as  $O(\text{eps})$ .

If now we are given a sequence of linear spaces  $\Phi_n$ ,  $n = 1, 2, 3, \dots$ , then clearly

$$\|f - \hat{\varphi}_1\| \geq \|f - \hat{\varphi}_2\| \geq \|f - \hat{\varphi}_3\| \geq \dots,$$

which follows not only from (5.1.20), but more directly from the fact that

$$\Phi_1 \subset \Phi_2 \subset \Phi_3 \subset \dots$$

If there are infinitely many such spaces, then the sequence of  $L^2$  errors, being monotonically decreasing, must converge to a limit. Is this limit zero? If so, we say that the least square

approximation process converges (in the mean) as  $n \rightarrow \infty$ . It is obvious from (5.1.20) that a necessary and sufficient condition for this is

$$\sum_{j=1}^{\infty} |\hat{c}_j|^2 \|\pi_j\|^2 = \|f\|^2. \quad (5.1.21)$$

An equivalent way of stating convergence is as follows: given any  $f$  with  $\|f\| < \infty$ , that is  $\forall f \in L^2_{d\lambda}$  and given any  $\varepsilon > 0$  no matter how small, there exists an integer  $n = n_\varepsilon$  and a function  $\varphi^* \in \Phi_n$  such that  $\|f - \varphi^*\| \leq \varepsilon$ , for all  $n > n_\varepsilon$ . A class of spaces  $\Phi_n$  having this property is said to be *complete* with respect to the norm  $\|\cdot\| = \|\cdot\|_{2,d\lambda}$ . One therefore calls the relation (5.1.21) the *completeness relation* or *Parseval-Liapunov relation*.

## 5.2 Examples of orthogonal systems

The prototype of all orthogonal systems is the system of trigonometric functions known from Fourier analysis. Other widely used systems involve orthogonal algebraic polynomials.

(1) **The trigonometric system** consists of the functions

$$1, \cos t, \cos 2t, \cos 3t, \dots, \sin t, \sin 2t, \sin 3t, \dots$$

It is orthogonal on  $[0, 2\pi]$  with respect to the equally weighted measure

$$d\lambda(t) = \begin{cases} dt & \text{on } [0, 2\pi], \\ 0 & \text{otherwise.} \end{cases}$$

We have

$$\int_0^{2\pi} \sin kt \sin \ell t \, dt = \begin{cases} 0, & \text{if } k \neq \ell \\ \pi, & \text{if } k = \ell \end{cases} \quad k, \ell = 1, 2, 3, \dots$$

$$\int_0^{2\pi} \cos kt \cos \ell t \, dt = \begin{cases} 0, & \text{if } k \neq \ell \\ 2\pi, & \text{if } k = \ell = 0 \\ \pi, & \text{if } k = \ell > 0 \end{cases} \quad k, \ell = 0, 1, 2, \dots$$

$$\int_0^{2\pi} \sin kt \cos \ell t \, dt = 0, \quad k = 1, 2, 3, \dots, \quad \ell = 0, 1, 2, \dots$$

The form of approximation is

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k \cos kt + b_k \sin kt). \quad (5.2.1)$$

Using (5.1.16) we get

$$a_k = \frac{1}{\pi} \int_0^{2\pi} f(t) \cos kt \, dt, \quad k = 1, 2, \dots$$

$$b_k = \frac{1}{\pi} \int_0^{2\pi} f(t) \sin kt dt, \quad k = 1, 2, \dots \quad (5.2.2)$$

which are known as *Fourier coefficients* of  $f$ . They are precisely the coefficients (5.1.16) for the trigonometric system. By extension, the coefficients (5.1.16) for any orthogonal system  $(\pi_j)$  will be called Fourier coefficients of  $f$  relative to this system. In particular, we recognize the truncated Fourier series at  $k = m$  the best approximation of  $f$  from the class of trigonometric polynomials of degree  $\leq n$  relative to the norm

$$\|u\|_2 = \left( \int_0^{2\pi} |u(t)|^2 dt \right)^{1/2}.$$

**(2) Orthogonal polynomials.** Given a measure  $d\lambda$ , we know that any finite number of consecutive powers  $1, t, t^2, \dots$  are linearly independent on  $[a, b]$ , if  $\text{supp } d\lambda = [a, b]$ , whereas the finite set  $1, t, \dots, t^{n-1}$  is linearly independent on  $\text{supp } d\lambda = \{t_1, t_2, \dots, t_N\}$ . Since a linearly independent set can be orthogonalized by Gram-Schmidt procedure, any measure  $d\lambda$  of the type considered generates a unique set of monic<sup>2</sup> polynomials  $\pi_j(t, d\lambda)$ ,  $j = 0, 1, 2, \dots$  satisfying

$$\begin{aligned} \text{degree } \pi_j &= j, \quad j = 0, 1, 2, \dots \\ \int_{\mathbb{R}} \pi_k(t) \pi_{\ell}(t) d\lambda(t) &= 0, \text{ if } k \neq \ell \end{aligned} \quad (5.2.3)$$

These are called *orthogonal polynomials* relative to the measure  $d\lambda$ . Let the index  $j$  start from zero. The set  $\{\pi_j\}$  is infinite if  $\text{supp } d\lambda = [a, b]$ , and consists of exactly  $N$  polynomials  $\pi_0, \pi_1, \dots, \pi_{N-1}$  if  $\text{supp } d\lambda = \{t_1, \dots, t_N\}$ . The latter are referred to as *discrete orthogonal polynomials*.

Three consecutive orthogonal polynomials are linearly related. Specifically, there exists real constants  $\alpha_k = \alpha_k(d\lambda)$  and  $\beta_k = \beta_k(d\lambda) > 0$  (depending on the measure  $d\lambda$ ) such that

$$\begin{aligned} \pi_{k+1}(t) &= (t - \alpha_k) \pi_k(t) - \beta_k \pi_{k-1}(t), \quad k = 0, 1, 2, \dots \\ \pi_{-1}(t) &= 0, \quad \pi_0(t) = 1. \end{aligned} \quad (5.2.4)$$

(It is understood that (5.2.4) holds for all  $k \in \mathbb{N}$  if  $\text{supp } d\lambda = [a, b]$  and only for  $k = \overline{0, N-2}$  if  $\text{supp } d\lambda = \{t_1, t_2, \dots, t_N\}$ ).

To prove (5.2.4) and, at the same time identify the coefficients  $\alpha_k, \beta_k$  we note that

$$\pi_{k+1}(t) - t\pi_k(t)$$

is a polynomial of degree  $\leq k$ , and it can be expressed as a linear combination of  $\pi_0, \pi_1, \dots, \pi_k$ . We write this linear combination in the form

$$\pi_{k+1} - t\pi_k(t) = -\alpha_k \pi_k(t) - \beta_k \pi_{k-1}(t) + \sum_{j=0}^{k-2} \gamma_{k,j} \pi_j(t) \quad (5.2.5)$$

---

<sup>2</sup>A polynomial is called *monic* if its leading coefficient is equal to 1.

(with the understanding that empty sums are zero). Now, multiplying both sides of (5.2.5) by  $\pi_k$  in the sense of inner product defined in (5.1.3), we get

$$(-t\pi_k, \pi_k) = -\alpha_k(\pi_k, \pi_k);$$

that is

$$\alpha_k = \frac{(\pi_k, \pi_k)}{(\pi_k, \pi_k)}, \quad k = 0, 1, 2, \dots \quad (5.2.6)$$

Similarly, forming the inner product of (5.2.5) with  $\pi_{k-1}$  gives

$$(-t\pi_k, \pi_{k-1}) = -\beta_k(\pi_{k-1}, \pi_{k-1}).$$

Since  $(t\pi_k, \pi_{k-1}) = (\pi_k, t\pi_{k-1})$  and  $t\pi_{k-1}$  differs from  $\pi_k$  by a polynomial of degree  $< k$ , we obtain by orthogonality  $(t\pi_k, \pi_{k-1}) = (\pi_k, \pi_k)$ ; hence

$$\beta_k = \frac{(\pi_k, \pi_k)}{(\pi_{k-1}, \pi_{k-1})}, \quad k = 1, 2, \dots \quad (5.2.7)$$

Multiplication of (5.2.5) by  $\pi_\ell$ ,  $\ell < k - 1$ , yields

$$\gamma_{k,\ell} = 0, \quad \ell = 0, 1, \dots, k - 1 \quad (5.2.8)$$

The recursion (5.2.4) provides us with a practical scheme of generating orthogonal polynomials. Since  $\pi_0 = 1$ , we can compute  $\alpha_0$  by (5.2.6) with  $k = 0$ . This allows us to compute  $\pi_1$ , using (5.2.4), with  $k = 0$ . Knowing  $\pi_0, \pi_1$  we can go back to (5.2.6) and (5.2.7) and compute  $\alpha_1$  and  $\beta_1$ , respectively. This allow us to compute  $\pi_2$  via (5.2.4) with  $k = 1$ . Proceeding in this fashion, using alternatively (5.2.6), (5.2.7) and (5.2.4), we can generate as many orthogonal polynomials as are desired. This procedure, called Stieltjes's <sup>3</sup> procedure – is particularly well suited for discrete orthogonal polynomials, since the inner product is then a finite sum. In the continuous case, the computation of the inner product requires integration, which complicates matters. Fortunately, for some important special measures  $d\lambda(t) = w(t)$  the recursion coefficients are explicitly known.

The special case of *symmetry* (i.e.  $d\lambda(t) = w(t)$  with  $w(-t) = w(t)$  and  $\text{supp } d\lambda$  is symmetric with respect to the origin) deserves special attention. In this case  $\alpha_k = 0, \forall k \in \mathbb{N}$ , due to (5.2.1) since

---


$$(t\pi_k, \pi_k) = \int_{\mathbb{R}} t\pi_k^2(t) d\lambda(t) = \int_a^b w(t)t\pi_k^2(t) dt = 0,$$

<sup>3</sup> Thomas Joannes Stieltjes (1856-1894), born in the Netherlands, studied at the Technical Institute of Delft, but never finished to get his degree because of a deep-seated aversion to examinations. He nevertheless got a job at the Observatory of Leiden as a “computer assistant for astronomical calculation”. His early publication caught the attention of Hermite, who was able to eventually secure a university position for Stieltjes in Toulouse. A life-long friendship evolved between these two great men, of which two volumes of their correspondence gives vivid testimony (and still makes fascinating reading). Stieltjes is best known for his work on continued fractions and moment problem, which, among other things, led him to invent a new concept of integral which now bears his name. He died very young of tuberculosis at age of 38.



because the integrand is an odd function and the domain is symmetric with respect to the origin.

## 5.3 Examples of orthogonal polynomials

### 5.3.1 Legendre polynomials

They are defined by means of the so-called Rodrigues's formula

$$\pi_k(t) = \frac{k!}{(2k)!} \frac{d^k}{dt^k} (t^2 - 1)^k. \quad (5.3.1)$$

Let us check first the orthogonality on  $[-1, 1]$  relative to the measure  $d\lambda(t) = dt$ . For any  $0 \leq \ell < k$ , repeated integration by parts gives

$$\begin{aligned} \int_{-1}^1 t^\ell \frac{d^k}{dt^k} (t^2 - 1)^k = \\ \sum_{m=0}^{\ell} (-1)^\ell \ell(\ell-1)\dots(\ell-m+1) t^{\ell-m} \frac{d^{k-m-1}}{dt^{k-m-1}} (t^2 - 1)^k \Big|_{-1}^1 = 0, \end{aligned} \quad (5.3.2)$$

the last relation since  $0 \leq k - m - 1 < k$ . Thus,

$$(\pi_k, p) = 0, \quad \forall p \in \mathbb{P}_{k-1},$$

proving orthogonality. Writing (by symmetry)

$$\pi_k(t) = t^k + \mu_k t^{k-2} + \dots, \quad k \geq 2$$

and noting (again by symmetry) that the recurrence relation has the form

$$\pi_{k+1}(t) = t\pi_k(t) - \beta_k \pi_{k-1}(t),$$

we obtain

$$\beta_k = \frac{t\pi_k(t) - \pi_{k+1}(t)}{\pi_{k-1}(t)},$$

which is valid for all  $t$ . In particular as  $t \rightarrow \infty$ ,

$$\beta_k = \lim_{t \rightarrow \infty} \frac{t\pi_k(t) - \pi_{k+1}(t)}{\pi_{k-1}(t)} = \lim_{t \rightarrow \infty} \frac{(\mu_k - \mu_{k+1})t^{k-1} + \dots}{t^{k-1} + \dots} = \mu_k - \mu_{k+1}.$$

(If  $k = 1$ , set  $\mu_1 = 0$ .)

From Rodrigues's formula we find

$$\begin{aligned} \pi_k(t) &= \frac{k!}{(2k)!} \frac{d^k}{dt^k} (t^{2k} - kt^{2k-2} + \dots) \\ &= \frac{k!}{(2k)!} (2k(2k-1)\dots(k+1)t^k - k(2k-2)(2k-3)\dots(k-1)t^{k-1} + \dots) \\ &= t^k - \frac{k(k-1)}{2(2k-1)} t^{k-2} + \dots, \end{aligned}$$

so that

$$\mu_k = \frac{k(k-1)}{2(2k-1)}, \quad k \geq 2.$$

Therefore,

$$\beta_k = \mu_k - \mu_{k+1} = \frac{k^2}{(2k-1)(2k+1)}$$

that is, since  $\mu_1 = 0$ ,

$$\beta_k = \frac{1}{4-k^2}, \quad k \geq 1. \quad (5.3.3)$$

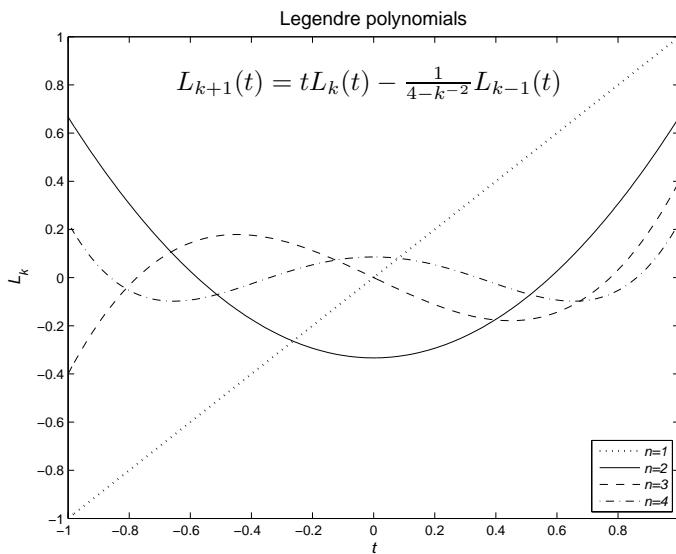


Figure 5.2: Legendre polynomials

MATLAB Source 5.3 computes the  $n$ th degree least squares Legendre approximation. It computes coefficients using formula (5.1.16) and then evaluates the approximation. The function vLegendre (MATLAB Source 5.1) computes the values of a Legendre polynomials with given degree on a given set of points. The inner products are computed via MATLAB function quadl.

Figure 5.2 gives the graphs of  $k$ th Legendre polynomials,  $k = \overline{1, 4}$ . They were generated using the MATLAB script graphLegendre.m:

```
%graphs for Legendre polynomials
n=4; clf
t=(-1:0.01:1)';
s=[];
ls={' ':'--','-'};
lw=[1.5,0.5,0.5,0.5];
for k=1:n
```

```

y=vLegendre(t,k);
s=[s;strcat('\itn=','int2str(k))];
plot(t,y,'LineStyle',ls{k}, 'Linewidth',lw(k), 'Color','k');
hold on
end
legend(s,4)
xlabel('t','FontSize',12,'FontAngle','italic')
ylabel('L_k','FontSize',12,'FontAngle','italic')
title('polinoame Legendre','Fontsize',14);
text(-0.65,0.8,...
'${L_{k+1}}(t)={L_k}(t)-\frac{1}{4-k^2}{L_{k-1}}(t)$',...
'FontSize',14,'FontAngle','italic','Interpreter','LaTeX')

```

We used L<sup>A</sup>T<sub>E</sub>X commands within `text` for a more pleasant looking of the recurrence relation.

---

**MATLAB Source 5.1** Compute Legendre polynomials using recurrence relation

---

```

function vl=vLegendre(x,n)
%VLEGENDRE - value of Legendre polynomial
%call vl=vLegendre(x,n)
%x - points
%n - degree
%vl - value

pnml = ones(size(x));
if n==0, vl=pnml; return; end
pn = x;
if n==1, vl=pn; return; end
for k=2:n
    vl=x.*pn-1/(4-(k-1)^(-2)).*pnml;
    pnml=pn; pn=vl;
end

```

---

---

**MATLAB Source 5.2 Compute Legendre coefficients**

---

```
function c=Legendrecoeff(f,n)
%LEGENDRECOEFF - coefficients of least squares Legendre
%                           approximation
%call c=Legendrecoeff(f,n)
%f - function
%n - degree

n3=2;
for k=0:n
    if k>0, n3=n3*k^2/(2*k-1)/(2*k+1); end
    c(k+1)=quadl(@fleg,-1,1,1e-12,0,f,k)/n3;
end
%subfunction
function y=fleg(x,f,k)
y=f(x).*vLegendre(x,k);
```

---



---

**MATLAB Source 5.3 Least square approximation via Legendre polynomials**

---

```
function y=Legendreapprox(f,x,n)
%LEGENDREAPPROX - continuous least squares Legendre
%                           approximation
%call y=Legendreapprox(f,x,n)
%f - function
%x - points
%n - degree

c=Legendrecoeff(f,n);
y=evalLegendreapprox(c,x);
```

---



---

```
function y=evalLegendreapprox(c,x)
%EVALLEGENDREAPPROX - evaluate least squares Legendre
%                           approximation

y=zeros(size(x));
for k=1:length(c)
    y=y+c(k)*vLegendre(x,k-1);
end
```

---

### 5.3.2 First kind Chebyshev polynomials

The Chebyshev<sup>4</sup> #1 polynomials can be defined by formula

$$T_n(x) = \cos(n \arccos x), \quad n \in \mathbb{N}. \quad (5.3.4)$$

The trigonometric identity

$$\cos(k+1)\theta + \cos(k-1)\theta = 2 \cos \theta \cos k\theta$$

and (5.3.4), by setting  $\theta = \arccos x$  give us

$$\begin{aligned} T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x) \quad k = 1, 2, 3, \dots \\ T_0(x) &= 1, \quad T_1(x) = x. \end{aligned} \quad (5.3.5)$$

For example,

$$\begin{aligned} T_2(x) &= 2x^2 - 1, \\ T_3(x) &= 4x^3 - 3x, \\ T_4(x) &= 8x^4 - 8x^2 + 1, \end{aligned}$$

and so on.

It is evident from (5.3.5) that the leading coefficient of  $T_n$  is  $2^{n-1}$  (if  $n \geq 1$ ); the first kind monic Chebyshev polynomial is

$$\overset{\circ}{T}_n(x) = \frac{1}{2^{n-1}} T_n(x), \quad n \geq 0, \quad \overset{\circ}{T}_0 = T_0. \quad (5.3.6)$$

From (5.3.4) we obtain immediately the zeros of  $T_n$

$$x_k^{(n)} = \cos \theta_k^{(n)}, \quad \theta_k^{(n)} = \frac{2k-1}{2n}\pi, \quad k = \overline{1, n}. \quad (5.3.7)$$

They are the projections onto the real line of equally spaced points on the unit circle; see Figure 5.3 for  $n = 4$ .

On  $[-1, 1]$   $T_n$  oscillates from +1 to -1, attaining this extreme values at

$$y_k^{(n)} = \cos \eta_k^{(n)}, \quad \eta_k^{(n)} = \frac{k\pi}{n}, \quad k = \overline{0, n}.$$

Figure 5.4 give the graphs of some first kind Chebyshev polynomials.

Pafnuty Levovich Cebyshev (1821-1894) was the most prominent of the St. Petersburg school of mathematics. He made pioneering contributions to number theory, probability theory, and approximation theory. He is regarded as the founder of the constructive function theory, but also worked in mechanics, notably the theory of mechanisms, and in ballistics.



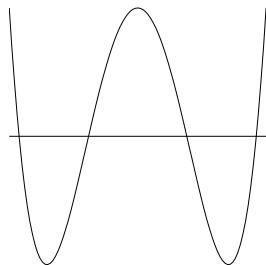
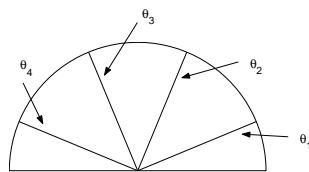


Figure 5.3: The Chebyshev polynomial  $T_4$  and its root

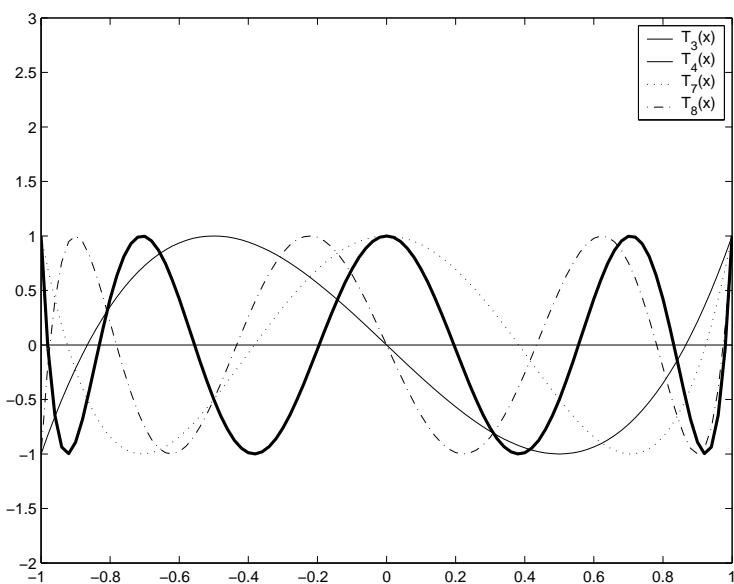


Figure 5.4: The Chebyshev #1 polynomials  $T_3$ ,  $T_4$ ,  $T_7$ ,  $T_8$  on  $[-1, 1]$

First kind Chebyshev polynomials are orthogonal relative to the measure

$$d\lambda(x) = \frac{dx}{\sqrt{1-x^2}}, \quad \text{on } [-1, 1].$$

One easily checks from (5.3.4) that

$$\begin{aligned} \int_{-1}^1 T_k(x)T_\ell(x) \frac{dx}{\sqrt{1-x^2}} &= \int_0^\pi T_k(\cos \theta)T_\ell(\cos \theta) d\theta \\ &= \int_0^\pi \cos k\theta \cos \ell\theta d\theta = \begin{cases} 0 & \text{if } k \neq \ell \\ \pi & \text{if } k = \ell = 0 \\ \pi/2 & \text{if } k = \ell \neq 0 \end{cases} \end{aligned} \quad (5.3.8)$$

The Fourier expansion in Chebyshev polynomials (essentially the Fourier cosine expansion) is given by

$$f(x) = \sum_{j=0}^{\infty}' c_j T_j(x) := \frac{1}{2}c_0 + \sum_{j=1}^{\infty} c_j T_j(x), \quad (5.3.9)$$

where

$$c_j = \frac{2}{\pi} \int_{-1}^1 f(x)T_j(x) \frac{dx}{\sqrt{1-x^2}}, \quad j \in \mathbb{N}.$$

Truncating (5.3.9) with the term of degree  $n$  gives a useful polynomial approximation of degree  $n$

$$\tau_n(x) = \sum_{j=0}^n' c_j T_j(x) := \frac{c_0}{2} + \sum_{j=1}^n c_j T_j(x), \quad (5.3.10)$$

having an error

$$f(x) - \tau_n(x) = \sum_{j=n+1}^{\infty} c_j T_j(x) \approx c_{n+1} T_{n+1}(x). \quad (5.3.11)$$

The approximation on the far right is better the faster the Fourier coefficients  $c_j$  tend to zero. The error (5.3.11), essentially oscillates between  $+c_{n+1}$  and  $-c_{n+1}$  and thus is of “uniform” size. This is in stark contrast to Taylor’s expansion at  $x = 0$ , where the  $n$ th degree polynomial partial sum has an error proportional to  $x^{n+1}$  on  $[-1, 1]$ .

With respect to the inner product

$$(f, g)_T := \sum_{k=1}^{n+1} f(\xi_k)g(\xi_k), \quad (5.3.12)$$

where  $\{\xi_1, \dots, \xi_{n+1}\}$  is the set of zeros of  $T_{n+1}$ , the following discrete orthogonality property holds

$$(T_i, T_j)_T = \begin{cases} 0, & i \neq j \\ \frac{n+1}{2}, & i = j \neq 0 \\ n+1, & i = j = 0 \end{cases} .$$

Indeed, we have  $\arccos \xi_k = \frac{2k-1}{2n+2}\pi$ ,  $k = \overline{1, n+1}$ . Let us compute now the inner product:

$$\begin{aligned}
(T_i, T_j)_T &= (\cos i \arccos t, \cos j \arccos t)_T = \\
&= \sum_{k=1}^{n+1} \cos(i \arccos \xi_k) \cos(j \arccos \xi_k) = \\
&= \sum_{k=1}^{n+1} \cos\left(i \frac{2k-1}{2(n+1)}\pi\right) \cos\left(j \frac{2k-1}{2(n+1)}\pi\right) = \\
&= \frac{1}{2} \sum_{k=1}^{n+1} \left[ \cos(i+j) \frac{2k-1}{2(n+1)}\pi + \cos(i-j) \frac{2k-1}{2(n+1)}\pi \right] = \\
&= \frac{1}{2} \sum_{k=1}^{n+1} \cos(2k-1) \frac{i+j}{2(n+1)}\pi + \frac{1}{2} \sum_{k=1}^{n+1} \cos(2k-1) \frac{i-j}{2(n+1)}\pi.
\end{aligned}$$

One introduces the notations  $\alpha := \frac{i+j}{2(n+1)}\pi$ ,  $\beta := \frac{i-j}{2(n+1)}\pi$  and

$$\begin{aligned}
S_1 &:= \frac{1}{2} \sum_{k=1}^{n+1} \cos(2k-1)\alpha, \\
S_2 &:= \frac{1}{2} \sum_{k=1}^{n+1} \cos(2k-1)\beta.
\end{aligned}$$

Since

$$\begin{aligned}
2 \sin \alpha S_1 &= \sin 2(n+1)\alpha, \\
2 \sin \beta S_2 &= \sin 2(n+1)\beta,
\end{aligned}$$

one obtains  $S_1 = 0$  and  $S_2 = 0$ .

With respect to the inner product

$$\begin{aligned}
(f, g)_U &:= \frac{1}{2} f(\eta_0)g(\eta_0) + f(\eta_1)g(\eta_1) + \cdots + f(\eta_{n-1})g(\eta_{n-1}) + \frac{1}{2} f(\eta_n)g(\eta_n) \\
&= \sum_{k=0}^n f(\eta_k)g(\eta_k),
\end{aligned} \tag{5.3.13}$$

where  $\{\eta_0, \dots, \eta_n\}$  is the set of extremal points of  $T_n$ , a similar property holds

$$(T_i, T_j)_U = \begin{cases} 0, & i \neq j \\ \frac{n}{2}, & i = j \neq 0 \\ n, & i = j = 0 \end{cases}.$$

MATLAB Source 5.4 computes continuous  $n$ th degree least squares approximation based on first kind Chebyshev polynomials. The method is analogous to Legendre approximation. In order to avoid the computation of improper integrals of the form

$$c_k = \frac{2}{\pi} \int_{-1}^1 \frac{1}{\sqrt{1-x^2}} f(x) \cos(k \arccos x) dx,$$

for the coefficients  $c_k = (f, T_k)$ , one performed the change of variable  $u = \arccos x$ . Thus, the formula for  $c_k$  becomes

$$c_k = \frac{2}{\pi} \int_0^\pi f(\cos u) \cos ku \, du.$$

MATLAB Source 5.5 computes the value of first kind Chebyshev polynomials of degree  $n$

---

**MATLAB Source 5.4** Least squares continuous approximation with Chebyshev # 1 polynomials

---

```
function y=Chebyshevapprox(f,x,n)
%CHEBYSHEVAPPROX - continuous least square Chebyshev #1 approx
%call y=Chebyshevapprox(f,x,n)
%f - function
%x - points
%n - degree
%y - approximation value

c=Chebyshevcoeff(f,n);
y=evalChebyshev(c,x);

function y=evalChebyshev(c,x)
%EVALCHEBYSHEV - evaluate least square Chebyshev approximation

y=c(1)/2*ones(size(x));
for k=1:length(c)-1
    y=y+c(k+1)*vChebyshev(x,k);
end
```

---

on a set of given points, and MATLAB Source 5.6 find Fourier coefficients of a Fourier series of Chebyshev polynomials.

The function that computes the discrete approximation corresponding to the inner product 5.3.12 is given in MATLAB Source 5.7. Such an approximation is useful in practical problems, and the precision is not much smaller than that of continuous approximation; moreover is simpler to compute, since it does not require to approximate integrals (the inner products are finite sums, see MATLAB Source 5.8).

The polynomial  $\overset{\circ}{T}_n$  has the least uniform norm in the set of  $n$ -th monic polynomials.

**Theorem 5.3.1 (Chebyshev).** *For an arbitrary monic polynomial  $\overset{\circ}{p}_n$  of degree  $n$ , there holds*

$$\max_{-1 \leq x \leq 1} \left| \overset{\circ}{p}_n(x) \right| \geq \max_{-1 \leq x \leq 1} \left| \overset{\circ}{T}_n(x) \right| = \frac{1}{2^{n-1}}, \quad n \geq 1, \quad (5.3.14)$$

where  $\overset{\circ}{T}_n(x)$  is the monic Chebyshev polynomial (5.3.6) of degree  $n$ .

*Proof.* (by contradiction) Assume contrary to (5.3.14), that

$$\max_{-1 \leq x \leq 1} \left| \overset{\circ}{p}_n(x) \right| < \frac{1}{2^{n-1}}. \quad (5.3.15)$$

**MATLAB Source 5.5 Compute Chebyshev # 1 polynomials by mean of recurrence relation**

---

```

function y=vChebyshev(x,n)
%VCHEBYSHEV - values of Chebyshev #1 polynomial
%call y=vChebyshev(x,n)
%x - points
%n - degree
%y - values of Chebyshev polynomial

pnml=ones(size(x));
if n==0, y=pnml; return; end
pn=x;
if n==1, y=pn; return; end
for k=2:n
    y=2*x.*pn-pnml;
    pnml=pn;
    pn=y;
end

```

---

**MATLAB Source 5.6 Least squares approximation with Chebyshev # 1 polynomials – continuation: computing Fourier coefficients**

---

```

function c=Chebyshevcoeff(f,n)
%CHEBYSHEVCOEFF - least square Cebyshев coefficients.
%call c=Chebyshevcoeff(f,n)
%f - function
%n - degree
%c - coefficients

for k=0:n
    c(k+1)=2/pi*quadl(@fceb,0,pi,1e-12,0,f,k);
end
%subfunction
function y=fceb(x,f,k)
y=cos(k*x).*feval(f,cos(x));

```

---

**MATLAB Source 5.7 Discrete Chebyshev least squares approximation**

---

```

function y=discrChebyshevapprox(f,x,n)
%DISCRCHEBYSHEVAPPROX - discrete least square Chebyshev #1
%call y=discrChebyshevapprox(f,x,n)
%f - function
%x - points
%n - degree

c=discrChebyshevcoeff(f,n);
y=evalChebyshev(c,x);

```

---

**MATLAB Source 5.8** The coefficients of discrete Chebyshev least squares approximation

---

```

function c=discrChebyshevcoeff(f,n)
%DISCRCHEBYSHEVCOEFF - discrete least squares Chebyshev
%
%coefficients
%call c=discrChebyshevcoeff(f,n)
%f - function
%n - degree

xi=cos((2*[1:n+1]-1)*pi/(2*n+2));
y=f(xi)';
for k=1:n+1
    c(k)=2/(n+1)*vChebyshev(xi,k-1)*y;
end

```

---

Then the polynomial  $d_n(x) = \overset{\circ}{T}_n(x) - \overset{\circ}{p}_n(x)$  (of degree  $\leq n - 1$ ) satisfies

$$d_n(y_0^{(n)}) > 0, d_n(y_1^{(n)}) < 0, d_n(y_2^{(n)}) > 0, \dots, (-1)^n d_n(y_n^{(n)}) > 0. \quad (5.3.16)$$

Since  $d_n$  change sign at least  $n$  times, it must vanish identically; this contradicts (5.3.16); thus (5.3.15) cannot be true.  $\square$

The result (5.3.14) can be given the following interesting interpretation: the best uniform approximation on  $[-1, 1]$  to  $f(x) = x^n$  from  $\mathbb{P}_{n-1}$  is given by  $x^n - \overset{\circ}{T}_n(x)$ , that is, by the aggregate of terms of degree  $\leq n - 1$  in  $\overset{\circ}{T}_n$  taken with the minus sign. From the theory of uniform polynomial approximation it is known that the best approximant is unique. Therefore equality in (5.3.14) can hold only if  $\overset{\circ}{p}_n(x) = \overset{\circ}{T}_n(x)$ .

### 5.3.3 Second kind Chebyshev polynomials

Chebyshev #2 polynomials are defined by

$$Q_n(t) = \frac{\sin[(n+1)\arccos t]}{\sqrt{1-t^2}}, \quad t \in [-1, 1]$$

They are orthogonal on  $[-1, 1]$  relative to the measure  $d\lambda(t) = w(t)dt$ ,  $w(t) = \sqrt{1-t^2}$ .

The recurrence relation is

$$Q_{n+1}(t) = 2tQ_n(t) - Q_{n-1}(t), \quad Q_0(t) = 1, \quad Q_1(t) = 2t.$$

### 5.3.4 Laguerre polynomials

This Laguerre<sup>5</sup> polynomials are orthogonal on  $[0, \infty)$  with respect to the weight  $w(t) = t^\alpha e^{-t}$ . They are defined by

$$l_n^\alpha(t) = \frac{e^t t^{-\alpha}}{n!} \frac{d^n}{dt^n}(t^{n+\alpha} e^{-t}) \text{ for } \alpha > 1$$

The recurrence relation for monic polynomials  $\tilde{l}_n^\alpha$  is

$$\tilde{l}_{n+1}^\alpha(t) = (t - \alpha_n) \tilde{l}_n^\alpha(t) - (2n + \alpha + 1) \tilde{l}_{n-1}^\alpha(t),$$

where  $\alpha_0 = \Gamma(1 + \alpha)$  and  $\alpha_k = k(k + \alpha)$ , for  $k > 0$ .

### 5.3.5 Hermite polynomials

Hermite polynomials are defined by

$$H_n(t) = (-1)^n e^{t^2} \frac{d^n}{dt^n}(e^{-t^2}).$$

They are orthogonal on  $(-\infty, \infty)$  with respect to the weight  $w(t) = e^{-t^2}$  and the recurrence relation is for monic polynomials  $\tilde{H}_n(t)$  is

$$\tilde{H}_{n+1}(t) = t \tilde{H}_n(t) - \beta_n \tilde{H}_{n-1}(t),$$

where  $\beta_0 = \sqrt{\pi}$  and  $\beta_k = k/2$ , for  $k > 0$ .

### 5.3.6 Jacobi polynomials

They are orthogonal on  $[-1, 1]$  relative to the weight

$$w(t) = (1 - t)^\alpha (1 + t)^\beta.$$

Jacobi polynomials are generalizations of other orthogonal polynomials:

- For  $\alpha = \beta = 0$  we obtain Legendre polynomials.
- For  $\alpha = \beta = -1/2$  we obtain Chebyshev #1 polynomials.

Edmond Laguerre (1834-1886) was a French mathematician active<sup>5</sup> in Paris, who made essential contributions to geometry, algebra, and analysis.



- For  $\alpha = \beta = 1/2$  we obtain Chebyshev #2 polynomials.

**Remark 5.3.2.** For Jacobi polynomials we have

$$\alpha_k = \frac{\beta^2 - \alpha^2}{(2k + \alpha + \beta)(2k + \alpha + \beta + 2)}$$

and

$$\begin{aligned} \beta_0 &= 2^{\alpha+\beta+1} B(\alpha + 1, \beta + 1), \\ \beta_k &= \frac{4k(k+\alpha)(k+\alpha+\beta)(k+\beta)}{(2k+\alpha+\beta-1)(2k+\alpha+\beta)^2(2k+\alpha+\beta+1)}, \quad k > 0. \end{aligned} \quad \diamond$$

We conclude this section with a table of some classical weight functions, their corresponding orthogonal polynomials, and the recursion coefficients  $\alpha_k, \beta_k$  for generating orthogonal polynomials (see Table 5.2).

Polynomials	Notation	Weight	interval	$\alpha_k$	$\beta_k$
Legendre	$P_n(l_n)$	1	$[-1,1]$	0	$2 \ (k=0)$ $(4-k^2)^{-1} \ (k>0)$
Chebyshev #1	$T_n$	$(1-t^2)^{-\frac{1}{2}}$	$[-1,1]$	0	$\pi \ (k=0)$ $\frac{1}{2}\pi \ (k=1)$ $\frac{1}{4} \ (k>0)$
Chebyshev #2	$u_n(Q_n)$	$(1-t^2)^{\frac{1}{2}}$	$[-1,1]$	0	$\frac{1}{2}\pi \ (k=0)$ $\frac{1}{4} \ (k>0)$
Laguerre	$L_n^{(\alpha)}$	$t^\alpha e^{-t} \ \alpha > -1$	$[0,\infty)$	$2k+\alpha+1$	$\Gamma(1+\alpha) \ (k=0)$ $k(k+\alpha) \ (k>0)$
Hermite	$H_n$	$e^{-t^2}$	$\mathbb{R}$	0	$\sqrt{\pi} \ (k=0)$ $\frac{1}{2}k \ (k>0)$
Jacobi	$P_n^{(\alpha,\beta)}$	$(1-t)^\alpha (1-t)^\beta$ $\alpha > -1, \beta > -1$	$[-1,1]$	See Remark 5.3.2 page 175	

Table 5.2: Orthogonal Polynomials

### 5.3.7 A MATLAB example

Consider the function  $f : [-1, 1] \rightarrow \mathbb{R}$ ,  $f(x) = x + \sin \pi x^2$ . We shall study experimentally the following least squares approximations: Legendre, continuous Chebyshev #1 and discrete Chebyshev #1. First, we shall try to find the degree of approximation such that the error stays within a given tolerance. The idea is as follows: we shall evaluate the function and the approximation on a large number of point and we shall see if the Chebyshev norm,  $\|\cdot\|_\infty$ , of the difference vector is lower than the prescribed error. If true, one returns the degree, otherwise the computing proceeds with a larger  $n$ . MATLAB Source 5.9 returns the degree and the actual error. For example, for a tolerance equal to  $10^{-3}$ , one obtains the following results

**MATLAB Source 5.9 Test for least squares approximations**

---

```

function [n,erref]=excebc(f,err,proc)
%f - function
%err - error
%proc - approximation method (Legendre, continuous
%       Chebyshev, discrete Chebyshev

x=linspace(-1,1,100); %abscissas
y=f(x); %function values
n=1;
while 1
    ycc=proc(f,x,n); %approximation values
    erref=norm(y-ycc,inf); %error
    if norm(y-ycc,inf)<err %success
        return
    end
    n=n+1;
end

```

---

```

>> fp=@(x)x+sin(pi*x.^2);
>> [n,er]=excebc(fp,1e-3,@approxLegendre)
n =
     8
er =
   9.5931e-004
>> [n,er]=excebc(fp,1e-3,@approxChebyshev)
n =
     8
er =
   5.9801e-004
>> [n,er]=excebc(fp,1e-3,@approxChebyshevdiscr)
n =
    11
er =
   6.0161e-004

```

The next program (exorthpol.m) computes the coefficients and plots the three types of approximations for a given degree:

```

k=input('k=');
fp=inline('x+sin(pi*x.^2)');
x=linspace(-1,1,100);
y=fp(x);
yle=Legendreapprox(fp,x,k);
ycc=Chebyshevapprox(fp,x,k);
ycd=discrChebyshevapprox(fp,x,k);
plot(x,y,':', x,yle,'--', x,ycc,'-.',x,ycd,'-');
legend('f','Legendre', 'Continuous Chebyshev', 'Discrete Chebyshev',4)

```

```

title(['k=',int2str(k)],'Fontsize',14);
cl=Legendrecoeff(fp,k)
ccc=Chebyshevcoeff(fp,k)
ccd=discrChebyshevcoeff(fp,k)

```

For  $k=3$  and  $k=4$ , one obtains the following values for the coefficients:

```

k=3 cl =
    Columns 1 through 3
    0.50485459411369    1.000000000000000    0.56690206826580
    Column 4
    0.000000000000000

ccc =
    Columns 1 through 3
    0.94400243153647    1.00000000000114    0.000000000000000
    Column 4
    -0.000000000000000

ccd =
    Columns 1 through 3
    0.88803168065243    1.000000000000000         0
    Column 4
    -0.000000000000000

k=4 cl =
    Columns 1 through 3
    0.50485459411369    1.000000000000000    0.56690206826580
    Columns 4 through 5
    0.000000000000000   -4.02634086092250

ccc =
    Columns 1 through 3
    0.94400243153647    1.00000000000114    0.000000000000000
    Columns 4 through 5
    -0.000000000000000   -0.49940325827041

ccd =
    Columns 1 through 3
    0.94400233847188    1.000000000000000   -0.02739538442025
    Columns 4 through 5
    -0.000000000000000   -0.49939655365619

```

The graphs are given in figure 5.5.

## 5.4 Polynomials and data fitting in MATLAB

MATLAB represents a polynomial

$$p(x) = p_1x^n + p_2x^{n-1} + p_nx + p_{n+1}$$

by a row vector  $p=[p(1) \ p(2) \ \dots \ p(n+1)]$  of coefficients, ordered in decreasing order of variable powers.

We shall consider three problems related to polynomials:

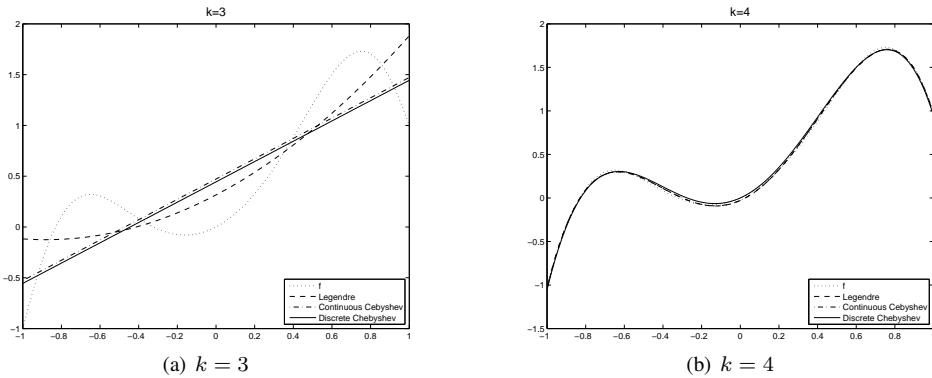


Figure 5.5: Least squares approximation of degree  $k$  for the function  $f : [-1, 1] \rightarrow \mathbb{R}$ ,  $f(x) = x + \sin \pi x^2$

- *Evaluation* — given the coefficients compute the value of the polynomial at one or more points.
- *Root finding* — given the coefficients find the roots.
- *Data fitting* — given a data set  $(x_i, y_i)_{i=1}^m$ , find a polynomial (or another combination of basic functions) that “fits” the data (i.e. a least squares approximation).

The evaluation uses Horner’s scheme, implemented in MATLAB by `polyval` function. In the command `y=polyval(p, x)`  $x$  can be a matrix, in which case the polynomial is evaluated at each element of the matrix (that is, in the array sense). Evaluation of the polynomial in matrix sense, that is the computation of the matrix

$$p(X) = p_1 X^n + p_2 X^{n-1} + \dots + p_n X + p_{n+1},$$

where  $X$  is a square matrix is carried out through the command `Y = polyvalm(p, X)`.

The command `z = roots(p)` computes the (real and complex) roots of  $p$ . The function `poly` carries out the converse operation, that is it constructs the polynomial given the roots. Thus if  $z$  is an  $n$ -vector then `p = poly(z)` gives the coefficients of the polynomial  $(x - z_1)(x - z_2) \dots (x - z_n)$ . It also accepts a square matrix argument: in this case `p=poly(A)` computes the coefficients of the characteristic polynomial of  $A$ ,  $\det(xI - A)$ .

The `polyder` function computes the coefficients of the derivative of a polynomial.

As an example, consider the quadratic polynomial  $p(x) = x^2 + x + 1$ . First, we find its roots:

```
>> p = [1 1 1]; z = roots(p)
z =
-0.5000 + 0.8660i
-0.5000 - 0.8660i
```

We check them, up to roundoff:

```
>> polyval(p, z)
ans =
1.0e-015 *
0.3331
0.3331
```

This is the characteristic polynomial of a certain  $2 \times 2$  matrix

```
>> A=[0, 1; -1,-1]; chp = poly(A)
chp =
1.0000    1.0000    1.0000
```

The Cayley-Hamilton theorem says that every matrix satisfies its own characteristic polynomial. We check this, modulo roundoff, for our matrix:

```
>> polyvalm(chp,A)
ans =
1.0e-015 *
-0.3331      0
0     -0.3331
```

For polynomial multiplication and division we can use `conv` and `deconv`, respectively. The `deconv` syntax is `[q, r]=deconv(g, h)`, where `g` is the dividend, `h` is the divisor, `q` the quotient and `r` the remainder. In the following example we divide  $x^3 - 2x^2 - x + 2$  by  $x - 2$ , obtaining quotient  $x^2 - 1$  and zero remainder. Then we reproduce the original polynomial using `conv`.

```
>> g = [1 -2 -1 2]; h=[1 -2];
>> [q,r] = deconv(g,h)
q =
1     0     -1
r =
0     0     0     0
>> conv(h,q)+r
ans =
1     -2     -1     2
```

Consider now the problem of data fitting. Data fitting is a very common source of least squares problems. Let  $t$  be the independent variable and let  $y(t)$  denote an unknown function of  $t$  that we want to approximate. Assume there are  $m$  observations,  $(y_i)$ , measured at specified values  $(t_i)$ :

$$y_i = y(t_i), \quad i = \overline{1, m}.$$

Our *model* for  $y$  is a combination of  $n$  basis functions  $(\pi_i)$

$$y(t) \approx c_1\pi_1(t, \alpha) + \cdots + c_n\pi_n(t, \alpha).$$

The *design matrix*  $A(\alpha)$  is a rectangular  $m \times n$  matrix with elements

$$a_{i,j} = \pi_j(t_i, \alpha).$$

They may depend upon  $\alpha$ . In matrix notation, we can express our model as:

$$y \approx A(\alpha)c.$$

The *residuals* are the differences between the observations and the model:

$$r_i = y_i - \sum_{j=1}^n c_j \pi_j(t_i, \alpha)$$

or in matrix notation

$$r = y - A(\alpha)c.$$

We wish to minimize a certain norm of the residuals. The most frequent choices are

$$\|r\|_2^2 = \sum_{i=1}^m r_i^2$$

or

$$\|r\|_{2,w}^2 = \sum_{i=1}^m w_i r_i^2.$$

A physical intuitive explanation of the second choice is: If some observations are more important or more accurate than others, then we might associate different weights,  $w_i$ , with different observations. For example, if the error in the  $i$ th observation is approximately  $e_i$ , then choose  $w_i = 1/e_i$ . Thus, we have a discrete least-squares approximation problem. This problem is linear if it does not depend on  $\alpha$  and nonlinear otherwise.

Any algorithm for solving an unweighted least squares problem can be used to solve a weighted problem by scaling the observations and design matrix. We simply multiply both  $y$  and the  $i$ th row of  $A$  by  $w_i$ . In MATLAB, this can be accomplished with

```
A=diag(w)*A
y=diag(w)*y
```

The MATLAB Optimization and Curve Fitting Toolboxes include functions for one-norm and infinity-norm problems. We will limit ourselves here to least squares.

If the problem is linear and we have more observation than base functions, we obtain an overdetermined system (see Section 4.6.2)

$$Ac \approx y.$$

We solve it in least squares sense

$$c = A \setminus y.$$

The theoretical approach is based on normal equations

$$A^T Ac = A^T y.$$

If the base functions are linear independent (and hence,  $A^T A$  nonsingular), the solution is

$$c = (A^T A)^{-1} A^T y,$$

or

$$c = A^+y,$$

where  $A^+$  is the pseudoinverse of  $A$ . MATLAB function `pinv` computes it.

Let  $Ax = b$  be an arbitrary system. If  $A$  is an  $m \times n$  with  $m > n$  and its rank is  $n$ , then any of the three following MATLAB instructions

```
x=A\b
x=pinv(A)*b
x=inv(A'*A)*A'*b
```

calculates the same least squares solution. Nevertheless, `\` operator does it faster.

If  $A$  is not full rank, the least squares solution is not unique. There exist several vectors which minimize the norm  $\|Ax - b\|_2$ . The solution computed with `x=A\b` is a *basic solution*; it has at most  $r$  nonzero components, where  $r$  is the rank of  $A$ . The solution computed with `x=pinv(A)*b` is the minimum norm solution (it minimizes `norm(x)`). The attempt to find the solution by using `x=inv(A'*A)*A'*b` fails, since  $A'*A$  is singular. Here is an example that illustrates various solutions. The matrix

`A=[1,2,3; 4,5,6; 7,8,9; 10,11,12];`  
is rank deficient. If `b=A(:,2)`, then an obvious solution of `A*x=b` is `x=[0, 1, 0]'`. None of the previous approaches compute `x`. The `\` operator yields

```
>> x=A\b
Warning: Rank deficient, rank = 2 tol = 1.4594e-014.
x =
    0.5000
        0
    0.5000
```

This solution has two nonzero components. The variant with pseudoinverse gives us

```
>> y=pinv(A)*b
y =
    0.3333
    0.3333
    0.3333
```

We see that `norm(y)=0.5774` < `norm(x)=0.7071`. The third variant fails:

```
>> z=inv(A'*A)*A'*b
Warning: Matrix is singular to working precision.
z =
    Inf
    Inf
    Inf
```

The normal equation approach has several drawbacks. The corresponding matrix is always more ill-conditioned than the initial overdetermined system. The condition number is in fact squared<sup>6</sup>:

$$\text{cond}(A^T A) = \text{cond}(A)^2.$$

---

<sup>6</sup>For a rectangular matrix  $X$ , the condition number can be defined by  $\text{cond}(X) = \|X\| \|X^+\|$

In floating point representation, even if the columns of  $A$  are linear independent,  $(A^T A)^{-1}$  can be almost singular (very close to a singular matrix).

MATLAB avoids normal equations. \ operator uses internally QR factorization. We can find the solution using  $c=R\backslash(Q' * y)$ .

If our base is  $1, t, \dots, t^n$ , one can use MATLAB `polyfit` function. The command `p=polyfit(x, y, n)` finds the coefficients of the degree  $n$  discrete least squares approximation polynomial for data  $x$  and  $y$ . If  $n \geq m$ , it returns the coefficients of the interpolation polynomial.

**Example 5.4.1.** A quantity  $y$  is measured at various time moments,  $t$ , to produce the observations given in Table 5.3. We introduce it in MATLAB using

t	y
0.0	0.82
0.3	0.72
0.8	0.63
1.1	0.60
1.6	0.55
2.3	0.50

Table 5.3: Input data for example 5.4.1

```
t=[0,0.3,0.8,1.1,1.6,2.3]';
y=[0.82,0.72,0.63,0.60,0.55,0.50]';
```

We shall try to model these data by the function

$$y(t) = c_1 + c_2 e^{-t}.$$

One computes the unknown coefficients with least squares methods. We have 6 equations and two unknowns, represented by a  $6 \times 2$  matrix

```
>> E=[ones(size(t)),exp(-t)]
E =
    1.0000    1.0000
    1.0000    0.7408
    1.0000    0.4493
    1.0000    0.3329
    1.0000    0.2019
    1.0000    0.1003
```

Using \ operator, we find:

```
c=E\y
c =
    0.4760
    0.3413
```

We plot on the same graph the function and the original data:

```
T=[0:0.1:2.5]';
Y=[ones(size(T)),exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
xlabel('t'); ylabel('y');
```

We observe that  $E \cdot c \neq y$ , but the Euclidian norm of the residual is minimized (Figure 5.6).

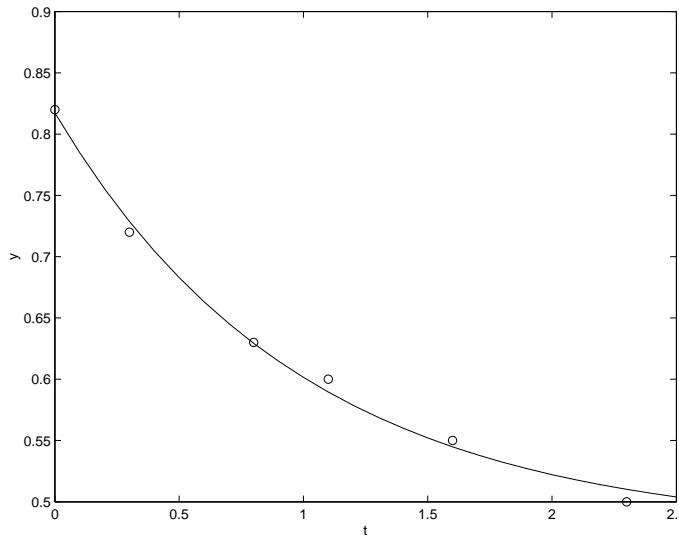


Figure 5.6: Illustration of data fitting

If  $A$  is rank deficient (i.e it has not linear independent columns), the operator  $\backslash$  gives a warning message and produce a solution with a minimum number of nonzero elements.

#### 5.4.1 An application — Census Data

In this example, the data are the total population of the United States, as determined by the U.S. Census, for the years 1900 to 2000. The units are millions of people [66]:

t	y
1900	75.995
1910	91.972
1920	105.711
1930	123.203
1940	131.669
1950	150.697
1960	179.323
1970	203.212
1980	226.505
1990	249.633
2000	281.422

The task is to model the population growth by a third order polynomial

$$y(t) = c_1 t^3 + c_2 t^2 + c_3 t + c_4$$

and predict the population when  $t = 2010$ .

If we try to fit our data with `c=polyfit(t,y,3)`, the design matrix will be ill-conditioned and badly scaled and its columns are nearly linearly dependent. We shall obtain the message

```
Warning: Polynomial is badly conditioned. Remove repeated
        data points or try centering and scaling as
        described in HELP POLYFIT.
```

A much better basis is provided by powers of a translated and scaled  $t$

$$s = (t - 1950)/50.$$

This new variable is in the interval  $[-1, 1]$  and the resulting design matrix is well conditioned. The MATLAB script 5.10, `census.m`, finds the coefficients, plot the data and the polynomial, and estimates the population in 2010. The estimation is given explicitly and marked by an asterisk (see Figure 5.7).

The reader is strongly encouraged to try other models.

## 5.5 The Space $H^n[a, b]$

For  $n \in \mathbb{N}^*$ , we define

$$H^n[a, b] = \{f : [a, b] \rightarrow \mathbb{R} : f \in C^{n-1}[a, b], f^{(n-1)} \text{ absolute continuous on } [a, b]\}. \quad (5.5.1)$$

Each function  $f \in H^n[a, b]$  admits a Taylor-type representation with the remainder in integral form

$$f(x) = \sum_{k=0}^{n-1} \frac{(x-a)^k}{k!} f^{(k)}(a) + \int_a^x \frac{(x-t)^{n-1}}{(n-1)!} f^{(n)}(t) dt. \quad (5.5.2)$$

$H^n[a, b]$  is a linear space.

---

**MATLAB Source 5.10** An example of least squares approximation

---

```
%CENSUS - example with polynomial fit

%data
y = [ 75.995 91.972 105.711 123.203 131.669 150.697 ...
       179.323 203.212 226.505 249.633 281.422]';
t = (1900:10:2000)'; % census years
x = (1890:1:2019)'; % evaluation years
w = 2010; % prediction year

s=(t-1950)/50;
xs=(x-1950)/50;
cs=polyfit(s,y,3);
zs=polyval(cs,xs);
est=polyval(cs,(2010-1950)/50);
plot(t,y,'o',x,zs,'-',w,est,'*')
text(1990,est,num2str(est))
title('U.S. Population', 'FontSize', 14)
xlabel('year', 'FontSize', 12)
ylabel('Millions', 'FontSize', 12)
```

---

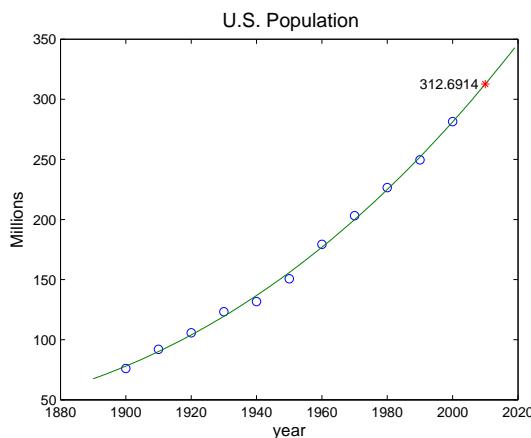


Figure 5.7: An illustration of census example

**Remark 5.5.1.** A function  $f : I \rightarrow \mathbb{R}$ ,  $I$  interval, is called *absolute continuous* on  $I$  if  $\forall \varepsilon > 0 \exists \delta > 0$  such that for each finite system of disjoint subintervals in  $I$   $\{(a_k, b_k)\}_{k=1}^n$  having the property  $\sum_{k=1}^n (b_k - a_k) < \delta$  it holds

$$\sum_{k=1}^n |f(b_k) - f(a_k)| < \varepsilon.$$

◇

The next theorem, due to Peano<sup>7</sup>, extremely important for Numerical Analysis, gives a representation of real linear functionals, defined on  $H^n[a, b]$ .

**Theorem 5.5.2 (Peano).** Let  $L$  be a real continuous linear functional, defined on  $H^n[a, b]$ . If  $Ker L = \mathbb{P}_{n-1}$  then

$$Lf = \int_a^b K(t) f^{(n)}(t) dt, \quad (5.5.3)$$

where

$$K(t) = \frac{1}{(n-1)!} L[(\cdot - t)_+^{n-1}] \quad (\text{Peano kernel}). \quad (5.5.4)$$

**Remark 5.5.3.** The function

$$z_+ = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

is called *positive part*, and  $z_+^n$  is called *truncated power*.

◇

*Proof.*  $f$  admits a Taylor representation with the remainder in integral form

$$f(x) = T_{n-1}(x) + R_{n-1}(x)$$

where

$$R_{n-1}(x) = \int_a^x \frac{(x-t)^{n-1}}{(n-1)!} f^{(n)}(t) dt = \frac{1}{(n-1)!} \int_a^b (x-t)_+^{n-1} f^{(n)}(t) dt$$

By applying  $L$  to both sides we get

$$Lf = \underbrace{LT_{n-1}}_0 + LR_{n-1} \Rightarrow Lf = \frac{1}{(n-1)!} L \left( \int_a^b (\cdot - t)_+^{n-1} f^{(n)}(t) dt \right) =$$



7

Giuseppe Peano (1858-1932), an Italian mathematician active in Turin, made fundamental contributions to mathematical logic, set theory, and the foundations of mathematics. General existence theorems in ordinary differential equations also bear his name. He created his own mathematical language, using symbols of the algebra and logic, and even promoted (and used) a simplified Latin (his “latino”) as a world language for scientific publication.

$$\stackrel{\text{cont}}{=} \frac{1}{(n-1)!} \int_a^b L(\cdot - t)_+^{n-1} f^{(n)}(t) dt.$$

□

**Remark 5.5.4.** The conclusion of the theorem remains valid if  $L$  is not continuous, but it has the form

$$Lf = \sum_{i=0}^{n-1} \int_a^b f^{(i)}(x) d\mu_i(x), \quad \mu_i \in BV[a, b].$$

◇

**Corollary 5.5.5.** If  $K$  does not change sign on  $[a, b]$  and  $f^{(n)}$  is continuous on  $[a, b]$ , then there exists  $\xi \in [a, b]$  such that

$$Lf = \frac{1}{n!} f^{(n)}(\xi) L e_n, \quad (5.5.5)$$

where  $e_k(x) = x^k$ ,  $k \in \mathbb{N}$ .

*Proof.* Since  $K$  does not change sign we may apply in (5.5.3) the second mean value theorem of integral calculus

$$Lf = f^{(n)}(\xi) \int_a^b K_n(t) dt, \quad \xi \in [a, b].$$

Setting  $f = e_n$  we get precise (5.5.5). □

## 5.6 Polynomial Interpolation

We now wish to approximate functions by matching their values at given points.

**Problem 5.1.** Given  $m + 1$  distinct points  $x_0, x_1, \dots, x_m$  and values  $f_i = f(x_i)$  of some function  $f \in X$  at these points, find a function  $\varphi \in \Phi$  such that

$$\varphi(x_i) = f_i, \quad i = \overline{1, m}.$$

Suppose  $\Phi$  is a  $(m + 1)$ -dimensional linear space. Since we have to satisfy  $m + 1$  conditions, and have at our disposal  $m + 1$  degrees of freedom – the coefficients of  $\varphi$  relative to a base of  $\Phi$  – we expect the problem to have a unique solution. Other questions of interest, in addition to existence and uniqueness, are different ways of representing and computing  $\varphi$ , what can be said about the error  $e(x) = f(x) - p(x)$  when  $x \neq x_i$ ,  $i = \overline{1, m}$  and the quality of approximation  $f(x) \approx \varphi(x)$  when the number of points, and hence the “degree” of  $\varphi$ , is allowed to increase indefinitely. Although these questions are not of the utmost interest in themselves, the results discussed in the sequel are widely used in the development of approximate methods for important practical tasks (numerical integration, equation solving and so on).

Interpolation to function values is referred to as *Lagrange-type interpolation*. More generally, we may wish to interpolate to function and derivative values of some function. This is called *Hermite-type interpolation*.

When  $\Phi = \mathbb{P}_n$  we have to deal with *polynomial interpolation*. In this case interpolation problem is called *Lagrange interpolation* and *Hermite interpolation*, respectively. For example, the Lagrange interpolation problem is stated as follows.

**Problem 5.2.** Given  $m + 1$  distinct points  $x_0, x_1, \dots, x_m$  and values  $f_i = f(x_i)$  of some function  $f \in X$  at these points, find a polynomial  $\varphi$  of minimum degree such that

$$\varphi(x_i) = f_i, \quad i = \overline{1, m}.$$

### 5.6.1 Lagrange interpolation

Let  $[a, b] \subset \mathbb{R}$  a closed interval, a set of  $m + 1$  distinct points  $\{x_0, x_1, \dots, x_m\} \subset [a, b]$  and a function  $f : [a, b] \mapsto \mathbb{R}$ . We wish to determine a polynomial of minimum degree reproducing the values of  $f$  at  $x_k$ ,  $k = \overline{0, m}$ .

**Theorem 5.6.1.** *There exists one polynomial and only one  $L_m f \in \mathbb{P}_m$  such that*

$$\forall i = 0, 1, \dots, m, \quad (L_m f)(x_i) = f(x_i); \quad (5.6.1)$$

*this polynomial can be written in the form*

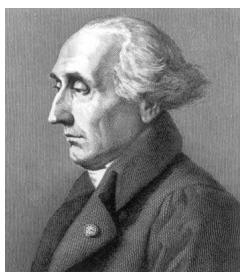
$$(L_m f)(x) = \sum_{i=0}^m f(x_i) \ell_i(x), \quad (5.6.2)$$

*where*

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^m \frac{x - x_j}{x_i - x_j}. \quad (5.6.3)$$

**Definition 5.6.2.** *The polynomial  $L_m f$  defined in Theorem 5.6.1 is called Lagrange<sup>8</sup> interpolation polynomial of  $f$  relative to the points  $x_0, x_1, \dots, x_m$ , and the functions  $\ell_i(x)$ ,  $i = \overline{0, m}$ , are called elementary (fundamental, basic) Lagrange polynomials associated to those points.*

8



Joseph Louis Lagrange (1736-1813), born in Turin, became, through correspondence with Euler, his protégé. In 1766 he indeed succeeded Euler in Berlin. He returned to Paris in 1787. Clairaut wrote of the young Lagrange: "... a Young man, no less remarkable for his talents than for his modesty; his temperament is mild and melancholic; he knows no other pleasure than study". Lagrange made fundamental contributions to the calculus of variations and to number theory, but worked also on many problems in analysis. He is widely known for his representation of the remainder term in Taylor's formula. The interpolation formula appeared in 1794. His *Mécanique Analytique*, published in 1788, made him one of the founders of analytic mechanics.

*Proof.* One proves immediately that  $\ell_i \in \mathbb{P}_i$  and that  $\ell_i(x_j) = \delta_{ij}$  (Krönecker's symbol); it results that the polynomial  $L_m f$  defined by (5.6.1) is of degree at most  $m$  and it satisfies (5.6.2). Suppose that there is another polynomial  $p_m^* \in \mathbb{P}_m$  which also verifies (5.6.2) and we set  $q_m = L_m - p_m^*$ ; we have  $q_m \in \mathbb{P}_m$  and  $\forall i = 0, m$ ,  $q_m(x_i) = 0$ ; so  $q_m$ , having  $(m+1)$  distinct roots vanishes identically, therefore the uniqueness result.  $\square$

The M-file `lagr.m` (MATLAB Source 5.11) gives the code for Lagrange interpolation using the formulas (5.6.2) and (5.6.3). The `lagr` function works also for symbolic variables.

---

### MATLAB Source 5.11 Lagrange Interpolation

---

```
function fi=lagr(x,y,xi)
%LAGR - computes Lagrange interpolation polynomial
% x,y - coordinates of nodes
% xi - evaluation points

if nargin ~=3
    error('illegal no. of arguments')
end
[mu,nu]=size(xi);
fi=zeros(mu,nu);
np1=length(y);
for i=1:np1
    z=ones(mu,nu);
    for j=[1:i-1,i+1:np1]
        z=z.*((xi-x(j))/(x(i)-x(j)));
    end;
    fi=fi+z*y(i);
end
```

---

Example:

```
>> a = 1:3; b = (a-2).^3; syms x;
>> P = lagr(a,b,x); pretty(P);
1/2 (-x + 2) (x - 3) + (1/2 x - 1/2) (x - 2)
```

**Remark 5.6.3.** The basic polynomial  $\ell_i$  is thus the unique polynomial satisfying

$$\ell_i \in \mathbb{P}_m \text{ and } \forall j = 0, 1, \dots, m, \quad \ell_i(x_j) = \delta_{ij}$$

Setting

$$u(x) = \prod_{j=0}^m (x - x_j)$$

from (5.6.3) we obtain that  $\forall x \neq x_i$ ,  $\ell_i(x) = \frac{u(x)}{(x-x_i)u'(x_i)}$ .  $\diamond$

The Figure 5.8 shows the graphs of third degree basic Lagrange polynomials for the nodes  $x_k = k$ ,  $k = \overline{0, 3}$ .

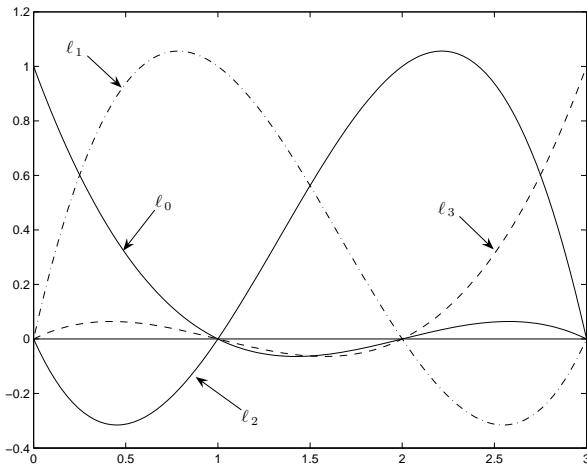


Figure 5.8: Basic Lagrange polynomials for the nodes  $x = 0, 1, 2, 3$

---

**MATLAB Source 5.12** Find basic Lagrange interpolation polynomials using MATLAB facilities.

---

```

function Z=pfl2(x,t)
%PFL2 - computes basic Lagrange polynomials
%call Z=pfl2(x,t)
%x - interpolation nodes
%t - evaluation points

%Return the result in a matrix: each line corresponds to a basic
%polynomial, and columns correspond to evaluation points
m=length(x);
n=length(t);
[T,X]=meshgrid(t,x);
TT=T-X;
Z=zeros(m,n);
TX=zeros(m,m);
[U,V]=meshgrid(x,x);
XX=U-V;
for i=1:m
    TX(i)=prod(XX([1:i-1,i+1:m],i));
    Z(i,:)=prod(TT([1:i-1,i+1:m],:))/TX(i);
end

```

---

The M-file `pfl2b.m` (MATLAB Source 5.12) computes basic Lagrange polynomials for a given set of nodes and a given set of evaluation points.

The proof of 5.6.1 proves in fact the existence and the uniqueness of the solution of general Lagrange interpolation problem:

(PGIL) Given the data  $b_0, b_1, \dots, b_m \in \mathbb{R}$ , determine

$$p_m \in \mathbb{P}_m \text{ such that } \forall i = 0, 1, \dots, n, \quad p_m(x_i) = b_i. \quad (5.6.4)$$

Problem (5.6.4) leads us to a linear system of  $(m + 1)$  equations with  $(m + 1)$  unknowns (the coefficients of  $p_m$ ).

It is a well-known result from linear algebra

$$\{\text{Existence of a solution } \forall b_0, b_1, \dots, b_m\} \Leftrightarrow \{\text{uniqueness of the solution}\} \Leftrightarrow$$

$$\{(b_0 = b_1 = \dots = b_m = 0) \Rightarrow p_m \equiv 0\}$$

We set  $p_m = a_0 + a_1x + \dots + a_mx^m$

$$a = (a_0, a_1, \dots, a_m)^T, \quad b = (b_0, b_1, \dots, b_m)^T$$

and let  $V = (v_{ij})$  be the  $m + 1$  by  $m + 1$  square matrix with elements  $v_{ij} = x_i^j$ . The equation (5.6.4) can be rewritten in the form

$$Va = b$$

The matrix  $V$  is invertible (it is a Vandermonde matrix); one can prove that  $V^{-1} = U^T$  where  $U = (u_{ij})$  with  $\ell_i(x) = \sum_{k=0}^m u_{ik}x^k$ ; in this way we obtain a not so expensive procedure to invert the Vandermonde matrix and thus to solve the system (5.6.4).

**Example 5.6.4.** The Lagrange interpolation polynomial of a function  $f$  relative to the nodes  $x_0$  and  $x_1$  is

$$(L_1 f)(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1),$$

that is, the line passing through the points  $(x_0, f(x_0))$  and  $(x_1, f(x_1))$ . Analogously, the Lagrange interpolation polynomial of a function  $f$  relative to the nodes  $x_0, x_1$  and  $x_2$  is

$$\begin{aligned} (L_2 f)(x) &= \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1) + \\ &\quad \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2), \end{aligned}$$

that is, the parabola passing through the points of coordinates  $(x_0, f(x_0)), (x_1, f(x_1))$  and  $(x_2, f(x_2))$ . Their geometric interpretation is given in Figure 5.9.  $\diamond$

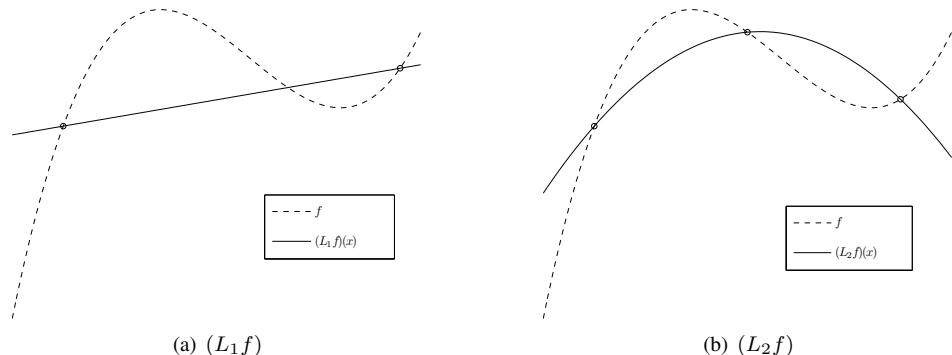


Figure 5.9: Geometric interpretation of  $L_1f$  (left) and  $L_2f$

### 5.6.2 Hermite Interpolation

Instead of making  $f$  and the interpolation polynomial to agree at points  $x_i$  in  $[a, b]$ , we could make that  $f$  and the interpolation polynomial to agree together with their derivatives up to the order  $r_i$  at points  $x_i$ . One obtains:

**Theorem 5.6.5.** Given  $(m + 1)$  distinct points  $x_0, x_1, \dots, x_m$  in  $[a, b]$  and  $(m + 1)$  natural numbers  $r_0, r_1, \dots, r_m$ , we set  $n = m + r_0 + r_1 + \dots + r_m$ . Then, given a function  $f$ , defined on  $[a, b]$  and having  $r_i$ th order derivative at point  $x_i$ , there exists one polynomial and only one  $H_n f$  of degree  $\leq n$  such that

$$\forall (i, \ell), 0 \leq i \leq m, 0 \leq \ell \leq r_i \quad (H_n f)^{(\ell)}(x_i) = f^{(\ell)}(x_i), \quad (5.6.5)$$

where  $f^{(\ell)}(x_i)$  is the  $\ell$ th order derivative of  $f$  at  $x_i$ .

**Definition 5.6.6.** The polynomial defined as above is called Hermite <sup>9</sup> interpolation polynomial of the function  $f$  relative to the points  $x_0, x_1, \dots, x_m$  and integers  $r_0, r_1, \dots, r_m$ .

*Proof.* Equation (5.6.5) leads us to a linear system having  $(n + 1)$  equations and  $(n + 1)$  unknowns (the coefficients of  $H_n f$ ), so it is sufficient to show that the corresponding homogeneous system has only the null solution, that is, the relations

$H_n f \in \mathbb{P}_n$  and  $\forall (i, \ell), 0 \leq i \leq k, 0 \leq \ell \leq r_i, (H_n f)^{(\ell)}(x_i) = 0$



9

Charles Hermite (1822-1901) was a leading French mathematician, Academician in Paris, known for his extensive work in number theory, algebra, and analysis. He is famous for his proof in 1873 of the transcendental nature of the number  $e$ .

guarantee us that for each  $i = 0, 1, \dots, m$   $x_i$  is a  $(r_i + 1)$ th order multiple root of  $H_n f$ ; therefore  $H_n f$  has the form

$$(H_n f)(x) = q(x) \prod_{i=0}^m (x - x_i)^{r_i+1},$$

where  $q$  is a polynomial. Since  $\sum_{i=0}^m (\alpha_i + 1) = n + 1$ , the above relation is incompatible to the membership of  $H_n$  to  $\mathbb{P}_n$ , excepting the situation when  $q \equiv 0$ , hence  $H_n \equiv 0$ .  $\square$

**Remark 5.6.7.** 1) Given the real numbers  $b_{i\ell}$ , for each pair  $(i, \ell)$  such that  $0 \leq i \leq k$  and  $0 \leq \ell \leq r_i$ , we proved that the general Hermite interpolation problem

$$\begin{aligned} & \text{determine } p_n \in \mathbb{P}_n \text{ such that } \forall (i, \ell) \ 0 \leq i \leq m \text{ and} \\ & 0 \leq \ell \leq r_i, \ p_n^{(\ell)}(x_i) = b_{i\ell} \end{aligned} \quad (5.6.6)$$

has a solution and only one. In particular, if we choose a given pair  $(i, \ell)$ ,  $b_{i\ell} = 1$  and  $b_{jn} = 0, \forall (j, m) \neq (i, \ell)$  one obtains a basic (fundamental) Hermite interpolation polynomial relative to the points  $x_0, x_1, \dots, x_m$  and integers  $r_0, r_1, \dots, r_m$ . The Hermite interpolation polynomial defined by (5.6.5) can be obtained using the basic polynomials

$$(H_n f)(x) = \sum_{i=0}^m \sum_{\ell=0}^{r_i} f^{(\ell)}(x) h_{i\ell}(x). \quad (5.6.7)$$

Setting

$$q_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^k \left( \frac{x - x_j}{x_i - x_j} \right)^{r_{j+1}}$$

one checks easily that the basic polynomials  $h_{i\ell}$  are defined by the recurrences

$$h_{ir_i}(x) = \frac{(x - x_i)^{r_i}}{r_i!} q_i(x)$$

and for  $\ell = r_{i-1}, r_{i-2}, \dots, 1, 0$

$$h_{i\ell}(x) = \frac{(x - x_i)^\ell}{\ell!} q_i(x) - \sum_{j=\ell+1}^{r_i} \binom{j}{\ell} q_i^{(j-\ell)}(x_i) h_{ij}(x).$$

- 2) The matrix  $V$  of the linear system (5.6.6) is called generalized Vandermonde matrix; it is invertible and the elements of its inverse are the coefficients of polynomials  $h_{il}$ .
- 3) Lagrange interpolation is a particular case of Hermite interpolation (for  $r_i = 0, i = 0, 1, \dots, m$ ); Taylor's polynomial is a particular case for  $m = 0$  and  $r_0 = n$ .  $\diamond$

We shall give a more convenient expression for Hermite basic polynomials due to Dimitrie D. Stancu [86]. They verify

$$\begin{aligned} h_{kj}^{(p)}(x_\nu) &= 0, \quad \nu \neq k, \ p = \overline{0, r_\nu} \\ h_{kj}^{(p)}(x_k) &= \delta_{jp}, \quad p = \overline{0, r_k} \end{aligned} \quad (5.6.8)$$

for  $j = \overline{0, r_k}$  and  $\nu, k = \overline{0, m}$ . Setting

$$u(x) = \prod_{k=0}^m (x - x_k)^{r_k+1}$$

and

$$u_k(x) = \frac{u(x)}{(x - x_k)^{r_k+1}},$$

it results from (5.6.8) that  $h_{kj}$  is of the form

$$h_{kj}(x) = u_k(x)(x - x_k)^j g_{kj}(x), \quad g_{kj} \in \mathbb{P}_{r_k-j}. \quad (5.6.9)$$

Applying Taylor's formula, we get

$$g_{kj}(x) = \sum_{\nu=0}^{r_k-j} \frac{(x - x_k)^\nu}{\nu!} g_{kj}^\nu(x_k); \quad (5.6.10)$$

now we must determine the values  $g_{kj}^\nu(x_k)$ ,  $\nu = \overline{0, r_k - j}$ . Rewriting (5.6.9) in the form

$$(x - x_k)^j g_{kj}(x) = h_{kj}(x) \frac{1}{u_k(x)},$$

and applying Leibnitz's formula for the  $(j + \nu)$ th order derivative of the product one gets

$$\sum_{s=0}^{j+\nu} \binom{j+\nu}{s} [(x - x_k)^j]^{(j+\nu-s)} g_{kj}^{(s)}(x) = \sum_{s=0}^{j+\nu} \binom{j+\nu}{s} h_{kj}^{(j+\nu-s)}(x) \left[ \frac{1}{u_k(x)} \right]^{(s)}.$$

Taking  $x = x_k$ , all terms in both sides, excepting those corresponding to  $s = \nu$  will vanish. Thus, we have

$$\binom{j+\nu}{\nu} j! g_{kj}^{(\nu)}(x_k) = \binom{j+\nu}{\nu} \left[ \frac{1}{u_k(x)} \right]_{x=x_k}^{(\nu)}, \quad \nu = \overline{0, r_k - j}.$$

We got

$$g_{kj}^{(\nu)}(x_k) = \frac{1}{j!} \left[ \frac{1}{u_k(x)} \right]_{x=x_k}^{(\nu)},$$

and from (5.6.10) and (5.6.9) we finally have

$$h_{kj}(x) = \frac{(x - x_k)^j}{j!} u_k(x) \sum_{\nu=0}^{r_k-j} \frac{(x - x_k)^\nu}{\nu!} \left[ \frac{1}{u_k(x)} \right]_{x=x_k}^{(\nu)}.$$

**Proposition 5.6.8.** *The operator  $H_n$  is a projector, i.e.*

- it is linear ( $H_n(\alpha f + \beta g) = \alpha H_n f + \beta H_n g$ );

- it is idempotent ( $H_n \circ H_n = H_n$ ).

*Proof.* Linearity results from (5.6.7). Due to the uniqueness of Hermite interpolation polynomial,  $H_n(H_n f) - H_n f$  vanishes identically, hence  $H_n(H_n f) = H_n f$ , that is, it is idempotent.  $\square$

**Example 5.6.9.** The Hermite interpolation polynomial corresponding to a function  $f$  and double nodes 0 and 1 is

$$(H_3 f)(x) = h_{00}(x)f(0) + h_{10}(x)f(1) + h_{01}(x)f'(0) + h_{11}(x)f'(1),$$

where

$$\begin{aligned} h_{00}(x) &= (x-1)^2(2x+1), \\ h_{01}(x) &= x(x-1)^2, \\ h_{10}(x) &= x^2(3-2x), \\ h_{11}(x) &= x^2(x-1). \end{aligned}$$

If we add the node  $x = \frac{1}{2}$ , then the quality of approximation increases (see Figure 5.10).  $\diamond$

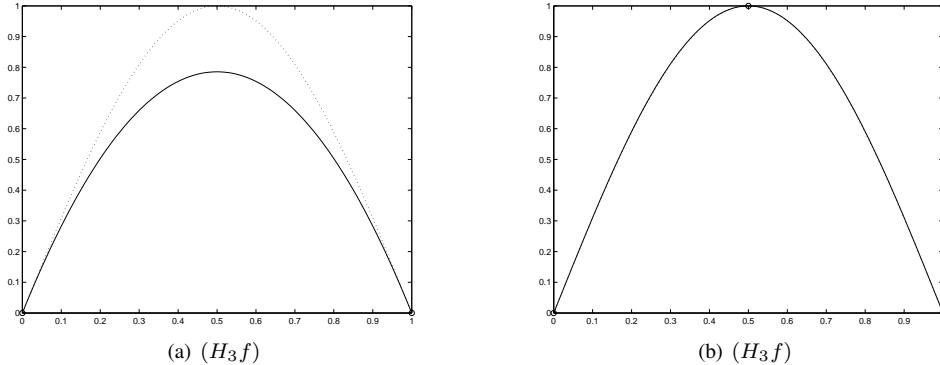


Figure 5.10: Hermite interpolation polynomial  $(H_3 f)$  (black) of the function  $f : [0, 1] \rightarrow \mathbb{R}$ ,  $f(x) = \sin \pi x$  (dotted) and double nodes  $x_0 = 0$  and  $x_1 = 1$  (left) and  $(H_5 f)$  of the function  $f : [0, 1] \rightarrow \mathbb{R}$ ,  $f(x) = \sin \pi x$  (dotted) and double nodes  $x_0 = 0$ ,  $x_1 = \frac{1}{2}$  and  $x_2 = 1$

### 5.6.3 Interpolation error

Recall that the norm of a linear operator,  $P_n$ , can be defined by

$$\|P_n\| = \max_{f \in C[a,b]} \frac{\|P_n f\|}{\|f\|}, \quad (5.6.11)$$

where in the right-hand side one chooses a convenient norm for functions. Taking the norm  $L^\infty$ , from Lagrange formula one obtains

$$\begin{aligned} \|(L_m f)(.)\|_\infty &= \max_{a \leq x \leq b} \left| \sum_{i=0}^m f(x_i) \ell_i(x) \right| \\ &\leq \|f\|_\infty \max_{a \leq x \leq b} \sum_{i=0}^m |\ell_i(x)|. \end{aligned} \quad (5.6.12)$$

Let  $\|\lambda_m\|_\infty = \lambda_m(x_\infty)$ . The equality holds for a function  $\varphi \in C[a, b]$ , piecewise linear, that verifies  $\varphi(x_i) = \text{sgn} \ell_i(x_\infty)$ ,  $i = \overline{0, m}$ . So,

$$\|P_n\|_\infty = \Lambda_m, \quad (5.6.13)$$

where

$$\Lambda_m = \|\lambda_m\|_\infty, \quad \lambda_m(x) = \sum_{i=0}^m |\ell_i(x)|. \quad (5.6.14)$$

The function  $\lambda_m(x)$  and its maximum  $\Lambda_m$  is called *Lebesgue function*<sup>10</sup> and *Lebesgue constant* for Lagrange interpolation. They provide a first estimation of interpolation error: let  $\mathcal{E}_m(f)$  be the error in best approximation of  $f$  by polynomials of degree  $\leq m$ ,

$$\mathcal{E}_m(f) = \min_{p \in \mathbb{P}_m} \|f - p\|_\infty = \|f - \hat{p}_n\|_\infty, \quad (5.6.15)$$

where  $\hat{p}_n$  is the  $m$ -th degree best approximation polynomial of  $f$ . Using the fact that  $L_m$  is a projector and formulas (5.6.12) and (5.6.14), one finds

$$\begin{aligned} \|f - L_m f\| &= \|f - \hat{p}_m - L_m(f - \hat{p}_m)\|_\infty \\ &\leq \|f - \hat{p}_m\|_\infty + \Lambda_m \|f - \hat{p}_m\|_\infty; \end{aligned}$$

that is,

$$\|f - L_m f\|_\infty \leq (1 + \Lambda_m) \mathcal{E}_m(f). \quad (5.6.16)$$

Thus, better is the approximation of  $f$  by polynomials of degree  $\leq m$ , smaller is the interpolation error. Unfortunately,  $\Lambda_m$  is not uniformly bounded: for any choice of the nodes  $x_i = x_i^{(m)}$ ,  $i = \overline{0, m}$ , it can show that  $\Lambda_m > O(\log m)$ , when  $m \rightarrow \infty$ . Nevertheless, it is

---

10



Henry Lebesgue (1875-1941) was a French mathematician known for his fundamental work on the theory of real function, notably the concepts of measure and integral that now bear his name.

not possible, based on Weierstrass' approximation theorem (that is, from  $\mathcal{E}_m \rightarrow 0, m \rightarrow \infty$ ), to draw the conclusion that Lagrange interpolation converges uniformly for any function  $f$ , not even for judiciously chosen nodes; in fact, it is known that the convergence does not hold.

If we wish to use Lagrange or Hermite interpolation polynomial to approximate a function  $f$  at a point  $x \in [a, b]$ ,  $x \neq x_k$ ,  $k = \overline{0, m}$ , we need to estimate the error  $(R_n f)(x) = f(x) - (H_n f)(x)$ . If we have not any information about  $f$  excepting the values at  $x_i$ , we can say nothing about  $(R_n f)(x)$ ; we can change  $f$  everywhere excepting the points  $x_i$  without modify  $(H_n f)(x)$ . We need some supplementary assumptions (regularity conditions) on  $f$ . Let  $C^m[a, b]$  be the space of real functions  $m$  times continuous-differentiable on  $[a, b]$ . We have the following theorem about error in Hermite interpolation.

**Theorem 5.6.10.** Suppose  $f \in C^m[\alpha, \beta]$  and there exists  $f^{(n+1)}$  on  $(\alpha, \beta)$ , where  $\alpha = \min\{x, x_0, \dots, x_m\}$  and  $\beta = \max\{x, x_0, \dots, x_m\}$ ; then, for each  $x \in [\alpha, \beta]$ , there exists a  $\xi_x \in (\alpha, \beta)$  such that

$$(R_n f)(x) = \frac{1}{(n+1)!} u_n(x) f^{(n+1)}(\xi_x), \quad (5.6.17)$$

where

$$u_n(x) = \prod_{i=0}^m (x - x_i)^{r_i+1}.$$

*Proof.* If  $x = x_i$ ,  $(R_n f)(x) = 0$  and (5.6.17) holds trivially. Suppose  $x \neq x_i$ ,  $i = \overline{0, m}$  and for a fixed  $x$ , we introduce the auxiliary function

$$F(z) = \begin{vmatrix} u_n(z) & (R_n f)(z) \\ u_n(x) & (R_n f)(x) \end{vmatrix}.$$

Note that  $F \in C^n[\alpha, \beta]$ ,  $\exists F^{(n+1)}$  on  $(\alpha, \beta)$ ,  $F(x) = 0$  and  $F^{(j)}(x_k) = 0$  for  $k = \overline{0, m}$ ,  $j = \overline{0, r_k}$ . Thus,  $F$  has  $(n+2)$  zeros, considering their multiplicities. Applying successively generalized Rolle's Theorem, it results that there exists at least one  $\xi \in (\alpha, \beta)$  such that  $F^{(n+1)}(\xi) = 0$ , i.e.

$$F^{(m+1)}(\xi) = \begin{vmatrix} (n+1)! & f^{(n+1)}(\xi) \\ u_n(x) & (R_n f)(x) \end{vmatrix} = 0, \quad (5.6.18)$$

where we used the relation  $(R_n f)^{(n+1)} = f^{(n+1)} - (H_n f)^{(n+1)} = f^{(n+1)}$ . Expressing  $(R_n f)(x)$  from (5.6.18) one obtains (5.6.17).  $\square$

**Corollary 5.6.11.** We set  $M_{n+1} = \max_{x \in [a, b]} |f^{(n+1)}(x)|$ ; an upper bound of interpolation error  $(R_n f)(x) = f(x) - (H_n f)(x)$  is given by

$$|(R_n f)(x)| \leq \frac{M_{n+1}}{(n+1)!} |u_n(x)|.$$

Since  $H_n$  is a projector,  $R_n$  is also a projector and additionally  $\text{Ker } R_n = \mathbb{P}_n$ , because  $R_n f = f - H_n f = f - f = 0, \forall f \in \mathbb{P}_n$ . Thus, we can apply Peano's Theorem to  $R_n$ .

**Theorem 5.6.12.** If  $f \in C^{n+1}[a, b]$ , then

$$(R_n f)(x) = \int_a^b K_n(x; t) f^{(n+1)}(t) dt, \quad (5.6.19)$$

where

$$K_n(x; t) = \frac{1}{n!} \left\{ (x-t)_+^n - \sum_{k=0}^m \sum_{j=0}^{r_k} h_{kj}(x) [(x_k - t)_+^n]^{(j)} \right\}. \quad (5.6.20)$$

*Proof.* Applying Peano's Theorem, we have

$$(R_n f)(x) = \int_a^b K_n(x; t) f^{(n+1)}(t) dt$$

and taking into account that

$$K_n(x; t) = R_n \left[ \frac{(x-t)_+^n}{n!} \right] = \frac{(x-t)_+^n}{n!} - H_n \left[ \frac{(x-t)_+^n}{n!} \right],$$

the theorem follows immediately.  $\square$

Since Lagrange interpolation is a particular case of Hermite interpolation for  $r_i = 0$ ,  $i = 0, 1, \dots, m$  we have from Theorem 5.6.10:

**Corollary 5.6.13.** Suppose  $f \in C^m[\alpha, \beta]$  and there exists  $f^{(m+1)}$  on  $(\alpha, \beta)$ , where  $\alpha = \min\{x, x_0, \dots, x_m\}$  and  $\beta = \max\{x, x_0, \dots, x_m\}$ ; then, for each  $x \in [\alpha, \beta]$ , there exists a  $\xi_x \in (\alpha, \beta)$  such that

$$(R_m f)(x) = \frac{1}{(n+1)!} u_m(x) f^{(m+1)}(\xi_x), \quad (5.6.21)$$

where

$$u_m(x) = \prod_{i=0}^m (x - x_i).$$

Also, it follows from Peano's Theorem 5.6.12:

**Corollary 5.6.14.** If  $f \in C^{m+1}[a, b]$ , then

$$(R_m f)(x) = \int_a^b K_m(x; t) f^{(m+1)}(t) dt \quad (5.6.22)$$

where

$$K_m(x; t) = \frac{1}{m!} \left[ (x-t)_+^m - \sum_{k=0}^m l_k(x) (x_k - t)_+^m \right]. \quad (5.6.23)$$

**Example 5.6.15.** For interpolation polynomials in example 5.6.4 the corresponding remainders are

$$(R_1 f)(x) = \frac{(x - x_0)(x - x_1)}{2} f''(\xi),$$

and

$$(R_2 f)(x) = \frac{(x - x_0)(x - x_1)(x - x_2)}{6} f'''(\xi),$$

respectively.  $\diamond$

**Example 5.6.16.** The remainder for the Hermite interpolation formula with double nodes 0 and 1, for  $f \in C^4[\alpha, \beta]$ , is

$$(R_3 f)(x) = \frac{x^2(x - 1)^2}{6!} f^{(4)}(\xi). \quad \diamond$$

**Example 5.6.17.** Let  $f(x) = e^x$ . For  $x \in [a, b]$ , we have  $M_{n+1} = e^b$  and for every choice of the points  $x_i$ ,  $|u_n(x)| \leq (b - a)^{n+1}$ , which implies

$$\max_{x \in [a, b]} |(R_n f)(x)| \leq \frac{(b - a)^{n+1}}{(n + 1)!} e^b.$$

One gets

$$\lim_{n \rightarrow \infty} \left\{ \max_{x \in [a, b]} |(R_n f)(x)| \right\} = \lim_{n \rightarrow \infty} \|(R_n f)(x)\| = 0,$$

that is,  $H_n f$  converges uniformly to  $f$  on  $[a, b]$ , when  $n$  tends to  $\infty$ . In fact, we can prove an analogous result for any function which can be developed into a Taylor series in a disk centered in  $x = \frac{a+b}{2}$  and with the radius of convergence  $r > \frac{3}{2}(b - a)$ .  $\diamond$

## 5.7 Efficient Computation of Interpolation Polynomials

### 5.7.1 Aitken-type methods

Many times, the degree required to attain the desired accuracy in polynomial interpolation is not known. It can be obtained from the remainder expression, but this require  $\|f^{(m+1)}\|_\infty$  to be known.  $P_{m_1, m_2, \dots, m_k}$  will denote the Lagrange interpolation polynomial with nodes  $x_{m_1}, \dots, x_{m_k}$ .

**Proposition 5.7.1.** If  $f$  is defined at  $x_0, \dots, x_k$ ,  $x_j \neq x_i$ ,  $0 \leq i, j \leq k$ , then

$$\begin{aligned} P_{0,1,\dots,k} &= \frac{(x - x_j)P_{0,1,\dots,j-1,j+1,\dots,k}(x) - (x - x_i)P_{0,1,\dots,i-1,i+1,\dots,k}(x)}{x_i - x_j} = \\ &= \frac{1}{x_i - x_j} \left| \begin{array}{cc} x - x_j & P_{0,1,\dots,i-1,i+1,\dots,k}(x) \\ x - x_i & P_{0,1,\dots,j-1,j+1,\dots,k}(x) \end{array} \right|. \end{aligned} \quad (5.7.1)$$

*Proof.*  $Q = P_{0,1,\dots,i-1,i+1,\dots,k}$ ,  $\hat{Q} = P_{0,1,\dots,j-1,j+1,k}$

$$P(x) = \frac{(x - x_j)\hat{Q}(x) - (x - x_i)Q(x)}{x_i - x_j}$$

$$P(x_r) = \frac{(x_r - x_j)\hat{Q}(x_r) - (x_r - x_i)Q(x_r)}{x_i - x_j} = \frac{x_i - x_j}{x_i - x_j} f(x_r) = f(x_r)$$

for  $r \neq i \wedge r \neq j$ , since  $Q(x_r) = \hat{Q}(x_r) = f(x_r)$ . But,

$$P(x_i) = \frac{(x_i - x_j)\hat{Q}(x_i) - (x_i - x_j)Q(x_i)}{x_i - x_j} = f(x_i)$$

and

$$P(x_j) = \frac{(x_j - x_i)\hat{Q}(x_j) - (x_j - x_i)Q(x_j)}{x_i - x_j} = f(x_j),$$

hence  $P = P_{0,1,\dots,k}$ .  $\square$

Thus we established a recurrence relation between a Lagrange interpolation polynomial of degree  $k$  and two Lagrange interpolation polynomials of degree  $k - 1$ . The computation could be organized in a tabular fashion

$x_0$	$P_0$					
$x_1$	$P_1$	$P_{0,1}$				
$x_2$	$P_2$	$P_{1,2}$	$P_{0,1,2}$			
$x_3$	$P_3$	$P_{2,3}$	$P_{1,2,3}$	$P_{0,1,2,3}$		
$x_4$	$P_4$	$P_{3,4}$	$P_{2,3,4}$	$P_{1,2,3,4}$	$P_{0,1,2,3,4}$	

And now, suppose  $P_{0,1,2,3,4}$  does not provide the desired accuracy. One can select a new node and add a new line to the table

$$x_5 \quad P_5 \quad P_{4,5} \quad P_{3,4,5} \quad P_{2,3,4,5} \quad P_{1,2,3,4,5} \quad P_{0,1,2,3,4,5}$$

and neighbor elements on row, column and diagonal could be compared to check if the desired accuracy was achieved.

The method is called *Neville method*.

We can simplify the notations

$$Q_{i,j} := P_{i-j,i-j+1,\dots,i-1,i},$$

$$Q_{i,j-1} = P_{i-j+1,\dots,i-1,i},$$

$$Q_{i-1,j-1} := P_{i-j,i-j+1,\dots,i-1}.$$

Formula (5.7.1) implies

$$Q_{i,j} = \frac{(x - x_{i-j})Q_{i,j-1} - (x - x_i)Q_{i-1,j-1}}{x_i - x_{i-j}},$$

for  $j = 1, 2, 3, \dots, i = j + 1, j + 2, \dots$

Moreover,  $Q_{i,0} = f(x_i)$ . We obtain

$$\begin{array}{cccc} x_0 & Q_{0,0} & & \\ x_1 & Q_{1,0} & Q_{1,1} & \\ x_2 & Q_{2,0} & Q_{2,1} & Q_{2,2} \\ x_3 & Q_{3,0} & Q_{3,1} & Q_{3,2} & Q_{3,3} \end{array}$$

If the interpolation procedure converges, then the sequence  $Q_{i,i}$  also converges and a stopping criterion could be

$$|Q_{i,i} - Q_{i-1,i-1}| < \varepsilon.$$

The algorithm speeds-up by sorting the nodes on ascending order over  $|x_i - x|$ .

*Aitken methods* is similar to Neville method. It builds the table

$$\begin{array}{ccccc} x_0 & P_0 & & & \\ x_1 & P_1 & P_{0,1} & & \\ x_2 & P_2 & P_{0,2} & P_{0,1,2} & \\ x_3 & P_3 & P_{0,3} & P_{0,1,3} & P_{0,1,2,3} \\ x_4 & P_4 & P_{0,4} & P_{0,1,4} & P_{0,1,2,4} & P_{0,1,2,3,4} \end{array}$$

To compute a new value one takes the value in top of the preceding column and the value from the current line and the preceding column.

### 5.7.2 Divided difference method

Let  $L_k f$  denotes the Lagrange interpolation polynomial with nodes  $x_0, x_1, \dots, x_k$  for  $k = 0, 1, \dots, n$ . We shall construct  $L_m$  by recurrence. We have

$$(L_0 f)(x) = f(x_0)$$

for  $k \geq 1$ , the polynomial  $L_k - L_{k-1}$  is of degree  $k$ , vanish at  $x_0, x_1, \dots, x_{k-1}$ , so its form is

$$(L_k f)(x) - (L_{k-1} f)(x) = f[x_0, x_1, \dots, x_k](x - x_0)(x - x_1) \dots (x - x_{k-1}), \quad (5.7.2)$$

where  $f[x_0, x_1, \dots, x_k]$  denotes the coefficient of  $x^k$  in  $(L_k f)(x)$ . One derives the expression of the interpolation polynomial  $L_m f$  with nodes  $x_0, x_1, \dots, x_n$

$$(L_m f)(x) = f(x_0) + \sum_{k=1}^m f[x_0, x_1, \dots, x_k](x - x_0)(x - x_1) \dots (x - x_{k-1}), \quad (5.7.3)$$

called *Newton's <sup>11</sup> form of Lagrange interpolation polynomial*.

Formula (5.7.3) reduces the computation by recurrence of  $L_m f$  to that of the coefficients  $f[x_0, x_1, \dots, x_k]$ ,  $k = \overline{0, m}$ .

It holds

**Lemma 5.7.2.**

$$\forall k \geq 1 \quad f[x_0, x_1, \dots, x_k] = \frac{f[x_1, x_2, \dots, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{x_k - x_0} \quad (5.7.4)$$

and

$$f[x_i] = f(x_i), \quad i = 0, 1, \dots, k.$$

*Proof.* For  $k \geq 1$  let  $L_{k-1}^* f$  be the interpolation polynomial for  $f$ , having the degree  $k-1$  and the nodes  $x_1, x_2, \dots, x_k$ ; the coefficient of  $x^{k-1}$  is  $f[x_1, x_2, \dots, x_k]$ . The polynomial  $q_k$  of degree  $k$  defined by

$$q_k(x) = \frac{(x - x_0)(L_{k-1}^* f)(x) - (x - x_k)(L_{k-1}^* f)(x)}{x_k - x_0}$$

equates  $f$  at points  $x_0, x_1, \dots, x_k$ , hence  $q_k(x) \equiv (L_k f)(x)$ . Formula (5.7.4) is obtained by identification of  $x^k$  coefficients in both sides.  $\square$

**Definition 5.7.3.** The quantity  $f[x_0, x_1, \dots, x_k]$  is called *kth divided difference of f relative to the nodes  $x_0, x_1, \dots, x_k$* .

An alternative notation is  $[x_0, \dots, x_k; f]$ .

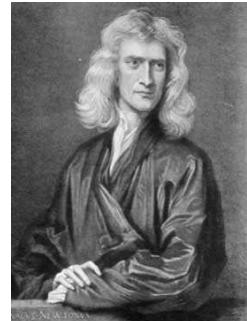
The definition implies that  $f[x_0, x_1, \dots, x_k]$  is independent of  $x$ 's order and it could be computed as a function of  $f(x_0), \dots, f(x_m)$ . Indeed, the Lagrange interpolation polynomial of degree  $\leq m$  relative to the nodes  $x_0, \dots, x_m$  can be written as

$$(L_m f)(x) = \sum_{i=0}^m l_i f(x_i)$$

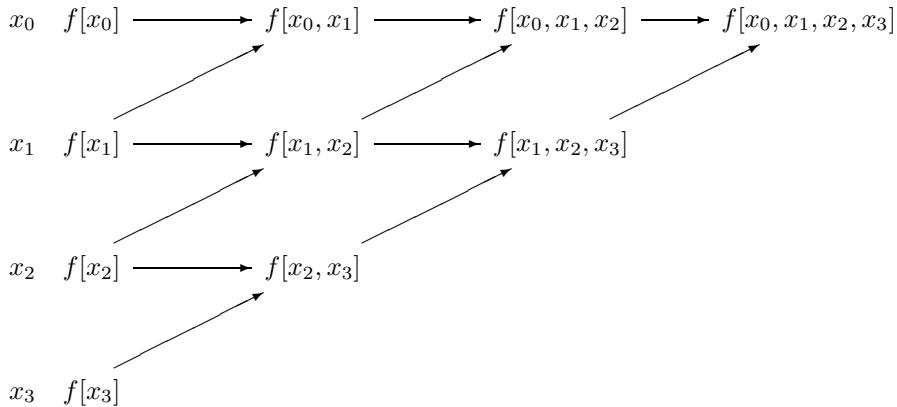
and the coefficient of  $x^m$  is

$$f[x_0, \dots, x_m] = \sum_{i=0}^m \frac{f(x_i)}{\prod_{\substack{j=0 \\ j \neq i}}^m (x_i - x_j)}. \quad (5.7.5)$$

Sir Isaac Newton (1643 - 1727) was an eminent figure of the 17th century mathematics and physics. Not only did he lay the foundations of modern physics, but he was also one of the co-inventors of differential calculus. Another was Leibniz, with whom he became entangled in a bitter and life-long priority dispute. His most influential work was the *Principia*, which not only contains his ideas on interpolation, but also his suggestion to use the interpolating polynomial for purposes of integration.



The formula (5.7.4) can be used to generate *the table of divided differences*



The first column contains the values of function  $f$ , the second contains the 1st order divided difference and so on; we pass from a column to the next using formula (5.7.4): each entry is the difference of the entry to the left and below it and the one immediately to the left, divided by the difference of the  $x$ -value found by going diagonally down and the  $x$ -value horizontally to the left. The divided differences that occur in the Newton formula (5.7.3) are precisely the  $m + 1$  entries in the first line of the table of divided differences. Their computation requires  $n(n + 1)$  additions and  $\frac{1}{2}n(n + 1)$  divisions. Adding another data point  $(x_{m+1}, f[x_{m+1}])$  requires the generation of the next diagonal.  $L_{m+1}f$  can be obtained from  $L_m f$  by adding to it the term  $f[x_0, \dots, x_{m+1}](x - x_0) \dots (x - x_{m+1})$ .

MATLAB Source 5.13 generates the divided difference table, and MATLAB Source 5.14 computes Newton's form of Lagrange interpolation polynomial.

---

#### **MATLAB Source 5.13** Generate divided difference table

---

```

function td=divdiff(x,f)
%DIVDIFF - compute divided difference table
%call td=divdiff(x,f)
%x - nodes
%f- function value
%td - divided difference table

lx=length(x);
td=zeros(lx, lx);
td(:,1)=f';
for j=2:lx
    td(1:lx-j+1, j)=diff(td(1:lx-j+2, j-1)) ./ ...
        (x(j:lx)-x(1:lx-j+1))';
end

```

---

**Remark 5.7.4.** The interpolation error is given by

$$f(x) - (L_m f)(x) = u_m(x)f[x_0, x_1, \dots, x_m, x]. \quad (5.7.6)$$

**MATLAB Source 5.14** Compute Newton's form of Lagrange interpolation polynomial

---

```

function z=Newtonpol(td,x,t)
%NEWTONPOL - computes Newton interpolation polynomial
%call z=Newtonpol(td,x,t)
%td - divided difference table
%x - interpolation nodes
%t - evaluation points
%z - values of interpolation polynomial

lt=length(t); lx=length(x);
for j=1:lt
    d=t(j)-x;
    z(j)=[1,cumprod(d(1:lx-1))]*td(1,:)';
end

```

---

Indeed, it is sufficient to note that

$$(L_m f)(t) + u_m(t)f[x_0, \dots, x_m; x]$$

is, according to (5.7.3) the interpolation polynomial (in  $t$ ) of  $f$  relative to the points  $x_0, x_1, \dots, x_m, x$ . The theorem on the remainder of Lagrange interpolation formula (5.6.17) implies the existence of a  $\xi \in (a, b)$  such that

$$f[x_0, x_1, \dots, x_m] = \frac{1}{m!} f^{(m)}(\xi) \quad (5.7.7)$$

(mean formula for divided differences).  $\diamond$

A divided difference could be written as the quotient of two determinants.

**Theorem 5.7.5.** *It holds*

$$f[x_0, \dots, x_m] = \frac{(Wf)(x_0, \dots, x_m)}{V(x_0, \dots, x_m)} \quad (5.7.8)$$

where

$$(Wf)(x_0, \dots, x_n) = \begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{m-1} & f(x_0) \\ 1 & x_1 & x_1^2 & \dots & x_1^{m-1} & f(x_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{m-1} & f(x_m) \end{vmatrix}, \quad (5.7.9)$$

and  $V(x_0, \dots, x_m)$  is the Vandermonde determinant.

*Proof.* One expands  $(Wf)(x_0, \dots, x_m)$  over the last columns; taking into account that every algebraic complement is a Vandermonde determinant, one gets

$$f[x_0, \dots, x_m] = \frac{1}{V(x_0, \dots, x_m)} \sum_{i=0}^m V(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_m) f(x_i) =$$

$$= \sum_{i=0}^m (-1)^{m-i} \frac{f(x_i)}{(x_i - x_0) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_n - x_i)},$$

that after the sign changing of the last  $m - i$  terms implies (5.7.5).  $\square$

### 5.7.3 Barycentric Lagrange Interpolation

We rewrite (5.6.2), (5.6.3) such that the Lagrange interpolation polynomial can be evaluated and updated in  $O(m)$  operations, in a numerically stable way. We have

$$\ell_j(x) = \frac{u_m(x)}{\prod_{k \neq j} (x_j - x_k)} \cdot \frac{1}{x - x_j}, \quad (5.7.10)$$

where

$$u_m(x) = (x - x_0)(x - x_1) \dots (x - x_m). \quad (5.7.11)$$

If one defines the barycentric weights by

$$w_j = \frac{1}{\prod_{k \neq j} (x_j - x_k)}, \quad j = 0, \dots, m, \quad (5.7.12)$$

that is,  $w_j = 1/u'_m(x_j)$ , we can thus write the basic Lagrange polynomials  $\ell_j$  as

$$\ell_j(x) = u_m(x) \frac{w_j}{x - x_j}. \quad (5.7.13)$$

Now, (setting  $f_j := f(x_j)$ ) we can write the Lagrange interpolation polynomial as

$$(L_m f)(x) = u_m(x) \sum_{j=0}^m \frac{w_j}{x - x_j} f_j. \quad (5.7.14)$$

The formula (5.7.14) is called *the first barycentric formula*.

By interpolating the constant function 1, whose interpolant is itself, we obtain

$$1 = \sum_{j=0}^m \ell_j(x) = u_m(x) \sum_{j=0}^m \frac{w_j}{x - x_j}. \quad (5.7.15)$$

Dividing (5.7.14) by the above expression and simplifying with the factor  $u_m(x)$ , we obtain the formula

$$p(x) = \frac{\sum_{j=0}^m \frac{w_j}{x - x_j} f_j}{\sum_{j=0}^m \frac{w_j}{x - x_j}}, \quad (5.7.16)$$

called *the second barycentric formula* [77].

### Remarkable point distributions

For certain particular distributions of nodes there exist explicit formulas for the barycentric weights  $w_j$ , starting from the formula

$$w_j = \frac{1}{u'_m(x_j)}, \quad (5.7.17)$$

- For equispaced nodes, we have

$$w_j = (-1)^j \binom{m}{j}. \quad (5.7.18)$$

- The *Chebyshev nodes family* could be obtained by projecting equally spaced points laid on the unit circle on the interval  $[-1, 1]$ .

- The *Chebyshev points of the first kind* of the first kind on  $[-1, 1]$  are given by

$$x_j = \cos \frac{(2j+1)\pi}{2m+2}, \quad j = 0, \dots, m.$$

Simplifying factors independents of  $j$  we find

$$w_j = (-1)^j \sin \frac{(2j+1)\pi}{2m+2}. \quad (5.7.19)$$

- The *Chebyshev points of the second kind* or *Gauss-Lobatto points* on  $[-1, 1]$  are given by

$$x_j = \cos \frac{\pi j}{n}, \quad j = 0, \dots, n \quad (5.7.20)$$

and the corresponding barycentric weights are

$$w_j = \frac{2^{n-1}}{n} \begin{cases} (-1)^j / 2 & \text{dacă } j = 0 \text{ sau } j = n, \\ (-1)^j & \text{altfel.} \end{cases} \quad (5.7.21)$$

The common factor  $2^{n-1}/n$  could be eliminated, since it appears both in numerator and denominator of the barycentric formula (5.7.16):

$$w_j = (-1)^j \delta_j, \quad \delta_j = \begin{cases} 1/2, & j = 0 \text{ sau } j = n, \\ 1, & \text{altfel.} \end{cases} \quad (5.7.22)$$

- For an arbitrary interval  $[a, b]$  we can use the change of variable

$$t = \frac{2x - b - a}{b - a}.$$

### Interpolation at Chebyshev nodes

We can avoid the difficulties generated by higher degree polynomial interpolation by using points set which are clustered at the ends of the interval, for example Chebyshev points of second kind.

These nodes are utilized in `chebfun` MATLAB package, implemented at University of Oxford by a group led by professor L. N. Trefethen [26].

If  $(x_j)_{j=0}^n$  are Chebyshev points, the polynomial of nodes satisfies

$$\left| \prod_{j=0}^n (x - x_j) \right| \leq 2^{-n+1}$$

Theorem 5.7.6 gives several properties of interpolation at Chebyshev points of second kind.:

**Theorem 5.7.6.** Let  $f \in C[-1, 1]$ ,  $p_n$  its interpolation polynomial at Chebyshev points (5.7.20) and  $p_n^*$  its best approximation polynomial with respect to the uniform norm  $\|\cdot\|_\infty$ . Then

1.  $\|f - p_n\|_\infty \leq (2 + \frac{2}{\pi} \log n) \|f - p_n^*\|_\infty$
2. If  $\exists k \in \mathbb{N}^*$  such that  $f^{(k)}$  is a bounded variation function on  $[-1, 1]$ , then  $\|f - p_n\|_\infty = O(n^{-k})$ , as  $n \rightarrow \infty$ .
3. If  $f$  is analytic in a complex neighbourhood of  $[-1, 1]$ , there exists  $C < 1$  such that  $\|f - p_n\|_\infty = O(C^n)$ ; in particular, if  $f$  is analytic in a closed ellipse with foci  $\pm 1$  and semi-axis  $M \geq 1$  and  $m \geq 0$ , we can choose  $C = 1/(M + m)$ .

We can implement the barycentric formula as follows:

- If the evaluation point  $x$  is among the nodes, say  $x = x_i$ , one returns  $f_i = f(x_i)$ .
- Otherwise, we apply formula (5.7.16).

We give a MATLAB implementation of barycentric method (function `barycentricInterpolation`), see MATLAB source 5.15, inspired from [5].

The function `barycentricWeights`, MATLAB source 5.16, computes the barycentric weights. For barycentric interpolation at Chebyshev nodes of first kind and second kind, we give the functions `ChebLagrangeK1` (MATLAB source 5.17) and `ChebLagrange` (MATLAB source 5.18), respectively.

As an example we will interpolate the function  $f : [-1, 1] \rightarrow \mathbb{R}$ ,  $f(x) = |x| + 1/2x - x^2$  for  $m = 20$  and  $m = 100$  Chebyshev nodes of second kind.

```
m = 20;
fun = @(x) abs(x) + .5*x - x.^2;
x = sort(cos(pi*(0:m)/m));
f = fun(x);
xx = linspace(-1, 1, 5000);
```

**MATLAB Source 5.15** Barycentric Lagrange interpolation

---

```

function ff=barycentricInterpolation(x,y,xx,c)
%BARYCENTRICINTERPOLATION - barycentric Lagrange interpolation
%call ff=barycentricInterpolation(x,y,xx,c)
%x - nodes
%y - function values
%xx - interpolation points
%c - barycentric weights
%ff - values of interpolation polynomial

numer = zeros(size(xx));
denom = zeros(size(xx));
exact = zeros(size(xx)); %test if xx=nodes
for j=1:n+1
    xdiff = xx-x(j);
    temp = c(j)./xdiff;
    numer = numer+temp*y(j);
    denom = denom+temp;
    exact(xdiff==0) = j;
end
ff = numer ./ denom;
jj = find(exact);
ff(jj) = y(exact(jj));

```

---

**MATLAB Source 5.16** Bayicentric weights

---

```

function c = barycentricweigths( x )
%BARYCENTRICWEIGHTS - compute barycentric weights(coefficient)
%call c = barycentricweigths( x )
%x - nodes
%c - weights

n=length(x)-1;
c=ones(1,n+1);
for j=1:n+1
    c(j)=prod(x(j)-x([1:j-1, j+1:n+1]));
end
c=1./c;
end

```

---

**MATLAB Source 5.17** Barycentric Lagrange interpolation with Chebyshev nodes of first kind

---

```

function ff=ChebLagrange1(y,xx,a,b)
%CHEBLAGRANGE1 - Lagrange interpolation for Chebyshev #1 points
%- barycentric method
%call ff=ChebLagrange1(y,xx,a,b)
%y - function values;
%xx - evaluation points
%a,b - interval
%ff - values of Lagrange interpolation polynomial

n = length(y)-1;
if nargin==2
    a=-1; b=1;
end
c = sin((2*(0:n)' + 1)*pi/(2*n+2)).*(-1).^(0:n)';
x = sort(cos((2*(0:n)' + 1)*pi/(2*n+2))*(b-a)/2 + (a+b)/2);
ff=barycentricInterpolation(x,y,xx,c);
end

```

---

**MATLAB Source 5.18** Barycentric Lagrange interpolation with Chebyshev nodes of second kind

---

```

function ff=ChebLagrange(y,xx,a,b)
%CHEBLAGRANGE - Lagrange interpolation for Chebyshev #2 points
%- barycentric
%call ff=ChebLagrange(y,xx,a,b)
%y - function values;
%xx - evaluation points
%a,b - interval
%ff - values of Lagrange interpolation polynomial

n = length(y)-1;
if nargin==2
    a=-1; b=1;
end
c = [1/2; ones(n-1,1); 1/2].*(-1).^(0:n)';
x = sort(cos((0:n)' * pi/n)) * (b-a)/2 + (a+b)/2;
ff=barycentricInterpolation(x,y,xx,c);
end

```

---

```

ff=ChebLagrange(f,xx);
subplot(2,1,1)
plot(x,f,'.',xx,ff,'-')
m = 100;
x = sort(cos(pi*(0:m)/m));
f = fun(x);
xx = linspace(-1,1,5000);
ff=ChebLagrange(f,xx);
subplot(2,1,2)
plot(x,f,'.',xx,ff,'-')

```

The graph is given in Figure 5.11.

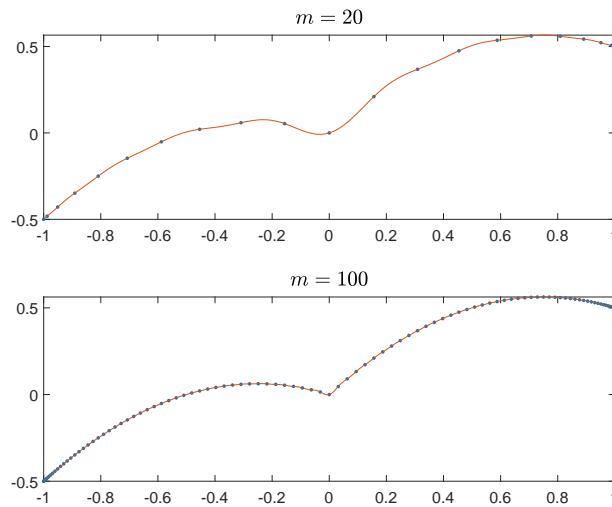


Figure 5.11: Barycentric Lagrange interpolation

### 5.7.4 Multiple nodes divided differences

Formula (5.7.8) allows us to introduce the notion of a multiple nodes divided difference: if  $f \in C^m[a, b]$  and  $\alpha \in [a, b]$ , then

$$\lim_{x_0, \dots, x_n \rightarrow \alpha} [x_0, \dots, x_n; f] = \lim_{\xi \rightarrow \alpha} \frac{f^m(\xi)}{m!} = \frac{f^{(m)}(\alpha)}{m!}$$

This suggests the relation

$$[\underbrace{\alpha, \dots, \alpha}_{m+1}; f] = \frac{1}{m!} f^{(m)}(\alpha).$$

Expressing this as a quotient of two determinants one obtains

$$(Wf) \left( \underbrace{\alpha, \dots, \alpha}_{m+1} \right) = \begin{vmatrix} 1 & \alpha & \alpha^2 & \dots & \alpha^{m-1} & f(\alpha) \\ 0 & 1 & 2\alpha & \dots & (m-1)\alpha^{m-2} & f'(\alpha) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & (m-1)! & f^{(m-1)}(\alpha) \end{vmatrix}$$

and

$$V \left( \underbrace{\alpha, \dots, \alpha}_{m+1} \right) = \begin{vmatrix} 1 & \alpha & \alpha^2 & \dots & \alpha^m \\ 0 & 1 & 2\alpha & \dots & m\alpha^{m-1} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & m! \end{vmatrix},$$

that is, the two determinants are built from the line of the node  $\alpha$  and its successive derivatives with respect to  $\alpha$  up to the  $m$ th order.

The generalization to several nodes is:

**Definition 5.7.7.** Let  $r_k \in \mathbb{N}$ ,  $k = \overline{0, m}$ ,  $n = r_0 + \dots + r_m$ . Suppose that  $f^{(j)}(x_k)$ ,  $k = \overline{0, m}$ ,  $j = \overline{0, r_k - 1}$  exist. The quantity

$$[\underbrace{x_0, \dots, x_0}_{r_0}, \underbrace{x_1, \dots, x_1}_{r_1}, \dots, \underbrace{x_m, \dots, x_m}_{r_m}; f] = \frac{(Wf)(x_0, \dots, x_0, \dots, x_m, \dots, x_m)}{V(x_0, \dots, x_0, \dots, x_m, \dots, x_m)}$$

where

$$(Wf)(x_0, \dots, x_0, \dots, x_m, \dots, x_m) = \begin{vmatrix} 1 & x_0 & \dots & x_0^{r_0-1} & \dots & x_0^{n-1} & f(x_0) \\ 0 & 1 & \dots & (r_0-1)x_0^{r_0-2} & \dots & f'(x_0) & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & (r_0-1)! & \dots & \prod_{p=1}^{r_0-1} (n-p)x_0^{n-r_0} & f^{(r_0-1)}(x_0) \\ 1 & x_m & \dots & x_m^{r_m-1} & \dots & x_m^{n-1} & f(x_m) \\ 0 & 1 & \dots & (r_m-1)x_m^{r_m-2} & \dots & (n-1)x_m^{n-2} & f'(x_m) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & (r_m-1)! & \dots & \prod_{p=1}^{r_m-1} (n-p)x_m^{n-r_m} & f^{(r_m-1)}(x_n) \end{vmatrix}$$

and  $V(x_0, \dots, x_0, \dots, x_m, \dots, x_m)$  is as above, excepting the last column which is

$$(x_0^n, nx_0^{n-1}, \dots, \prod_{p=0}^{r_0-2} (n-p)x_0^{n-r_0+1}, \dots, x_m^n, nx_m^{n-1}, \dots, \prod_{p=0}^{r_m-2} x_m^{n-r_m+1})^T$$

is called divided difference with multiple nodes  $x_k$ ,  $k = \overline{0, m}$  and orders of multiplicity  $r_k$ ,  $k = \overline{0, m}$ .

By generalization of Newton's form for Lagrange interpolation polynomial one obtains a method for computing Hermite interpolation polynomial based on multiple nodes divided difference.

Suppose nodes  $x_i$ ,  $i = \overline{0, m}$  and values  $f(x_i)$ ,  $f'(x_i)$  are given. We define the sequence of nodes  $z_0, z_1, \dots, z_{2n+1}$  by  $z_{2i} = z_{2i+1} = x_i$ ,  $i = \overline{0, m}$ . We build the divided difference table relative to the nodes  $z_i$ ,  $i = \overline{0, 2m+1}$ . Since  $z_{2i} = z_{2i+1} = x_i$  for every  $i$ ,  $f[z_{2i}, z_{2i+1}]$  is a divided difference with a double node and it equates  $f'(x_i)$ ; therefore we shall use  $f'(x_0), f'(x_1), \dots, f'(x_m)$  instead of first order divided differences

$$f[z_0, z_1], f[z_2, z_3], \dots, f[z_{2m}, z_{2m+1}].$$

$z_0 = x_0$	$f[z_0]$	$f[z_0, z_1] = f'(x_0)$	
$z_1 = x_0$	$f[z_1]$	$f[z_1, z_2] = \frac{f(z_2) - f(z_1)}{z_2 - z_1}$	$f[z_0, z_1, z_2] = \frac{f[z_1, z_2] - f[z_0, z_1]}{z_2 - z_0}$
$z_2 = x_1$	$f[z_2]$	$f[z_2, z_3] = f'(x_1)$	$f[x_1, z_2, z_3] = \frac{f[z_3, z_2] - f[z_2, z_1]}{z_3 - z_1}$
$z_3 = x_1$	$f[z_3]$	$f[z_3, z_4] = \frac{f(z_4) - f(z_3)}{z_4 - z_3}$	$f[z_2, z_3, z_4] = \frac{f[z_4, z_3] - f[z_3, z_2]}{z_4 - z_2}$
$z_4 = x_2$	$f[z_4]$	$f[z_4, z_5] = f'(x_2)$	$f[z_3, z_4, z_5] = \frac{f[z_5, z_4] - f[z_4, z_3]}{z_5 - z_3}$
$z_5 = x_2$	$f[z_5]$		

Table 5.4: A divided difference table with double nodes

The other divided differences are obtained as usual, as the Table 5.4 shows. This idea could be extended to another Hermite interpolation problems. The method is due to Powell.

MATLAB Source 5.19 contains a function for the computation of a divided difference table with double nodes, `divdiffdn`. It returns the nodes, taking into account their multiplicities, and the divided difference table. The Hermite interpolation polynomial can be computed using the function `Newtonpol` (see MATLAB Source 5.14, page 204) with nodes and table returned by `divdiffdn`.

## 5.8 Convergence of polynomial interpolation

Let's explain first what we mean by "convergence". We assume that we are given a triangular array of interpolation nodes  $x_i = x_i^{(m)}$ , exactly  $m + 1$  distinct nodes for each  $m = 0, 1, 2, \dots$ .

$$\begin{array}{ccccccc} & & x_0^{(0)} & & & & \\ & & x_0^{(1)} & x_1^{(1)} & & & \\ & & x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & & \\ & & \vdots & \vdots & \vdots & \ddots & \\ & & x_0^{(m)} & x_1^{(m)} & x_2^{(m)} & \dots & x_m^{(m)} \\ & & \vdots & \vdots & \vdots & & \vdots \end{array} \quad (5.8.1)$$

We assume further that all nodes are contained in some finite interval  $[a, b]$ . Then, for each  $m$  we define

$$P_m(x) = L_m(f; x_0^{(m)}, x_1^{(m)}, \dots, x_m^{(m)}; x), \quad x \in [a, b]. \quad (5.8.2)$$

We say that Lagrange interpolation based on the triangular array of nodes (5.8.1) converges if

$$p_m(x) \rightrightarrows f(x), \text{ when } n \rightarrow \infty \text{ pe } [a, b]. \quad (5.8.3)$$

**Example 5.8.1 (Runge's example).**

$$\begin{aligned} f(x) &= \frac{1}{1+x^2}, \quad x \in [-5, 5], \\ x_k^{(m)} &= -5 + 10 \frac{k}{m}, \quad k = \overline{0, m}. \end{aligned} \quad (5.8.4)$$

**MATLAB Source 5.19** Generates a divided difference table with double nodes

```

function [z,td]=divddffdn(x,f,fd)
%DIVDIFFDN - compute divide difference table for double nodes
%call [z,td]=divddffdn(x,f,fd)
%x -nodes
%f - function values at nodes
%fd - derivative values in nodes
%z - doubled nodes
%td - divided difference table

z=zeros(1,2*length(x));
lz=length(z);
z(1:2:1z-1)=x;
z(2:2:1z)=x;
td=zeros(lz,lz);
td(1:2:1z-1,1)=f';
td(2:2:1z,1)=f';
td(1:2:1z-1,2)=fd';
td(2:2:1z-2,2)=(diff(f)./diff(x))';
for j=3:1z
    td(1:1z-j+1,j)=diff(td(1:1z-j+2,j-1))./...
        (z(j:1z)-z(1:1z-j+1))';
end

```

Here the nodes are equally spaced in  $[-5, 5]$ . Note that  $f$  has two poles at  $z = \pm i$ . It has been shown, indeed, that

$$\lim_{m \rightarrow \infty} |f(x) - p_m(f; x)| = \begin{cases} 0 & \text{if } |x| < 3.633\dots \\ \infty & \text{if } |x| > 3.633\dots \end{cases} \quad (5.8.5)$$

◇

The graph for  $m = 10, 13, 16$  is given in Figure 5.12. It was generated with MATLAB Source 5.20, by command

```
>>runge3([10,13,17],[-5,5,-2,2])
```

using MATLAB graph's annotation facilities.

**Example 5.8.2 (Bernstein's example).** Let us consider the function

$$f(x) = |x|, \quad x \in [-1, 1],$$

and the nodes

$$x_k^{(m)} = -1 + \frac{2k}{m}, \quad k = 0, 1, 2, \dots, m. \quad (5.8.6)$$

Here analyticity of  $f$  is completely gone,  $f$  being not differentiable at  $x = 0$ . One finds that

$$\lim_{m \rightarrow \infty} |f(x) - L_m(f; x)| = \infty \quad \forall x \in [-1, 1]$$

excepting the points  $x = -1$ ,  $x = 0$  and  $x = 1$ . See figure 5.13(a), for  $m = 20$ . The convergence in  $x = \pm 1$  is trivial, since they are interpolation nodes, where the error is zero. The same is true for  $x = 0$  when  $m$  is even, but not if  $m$  is odd. The failure of the convergence in the last two examples can only in part be blamed on insufficient regularity of  $f$ . Another culprit is the equidistribution of nodes. There are better distributions such as Chebyshev nodes. Figure 5.13(b) gives the graph for  $m = 17$ . ◇

---

**MATLAB Source 5.20 Runge's Counterexample**

---

```
function runge3(n,w)
%n- a vector of degrees, w- window
clf
xg=-5:0.1:5; yg=1./(1+xg.^2);
plot(xg,yg,'k-','Linewidth',2);
hold on
nl=length(n);
ta=5*[-1:0.001:-0.36,-0.35:0.01:0.35, 0.36:0.001:1]';
ya=zeros(length(ta),nl);
leg=cell(1,nl+1); leg{1}='f';
for l=1:nl
    xn=5*[-1:2/n(l):1]; yn=1./(1+xn.^2);
    ya(:,l)=lagr(xn,yn,ta);
    leg{l+1}=strcat('L_{',int2str(n(l)),'}');
end
plot(ta,ya); axis(w)
legend(leg,-1)
```

---

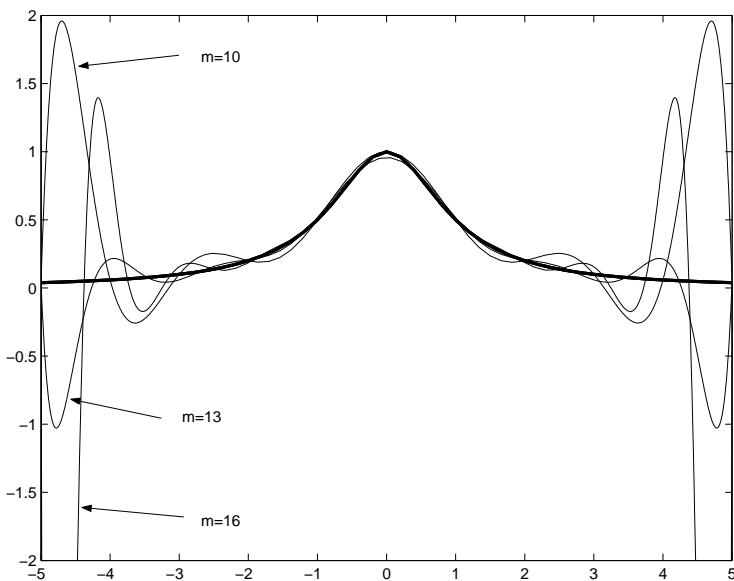


Figure 5.12: A graphical illustration of Runge's counterexample

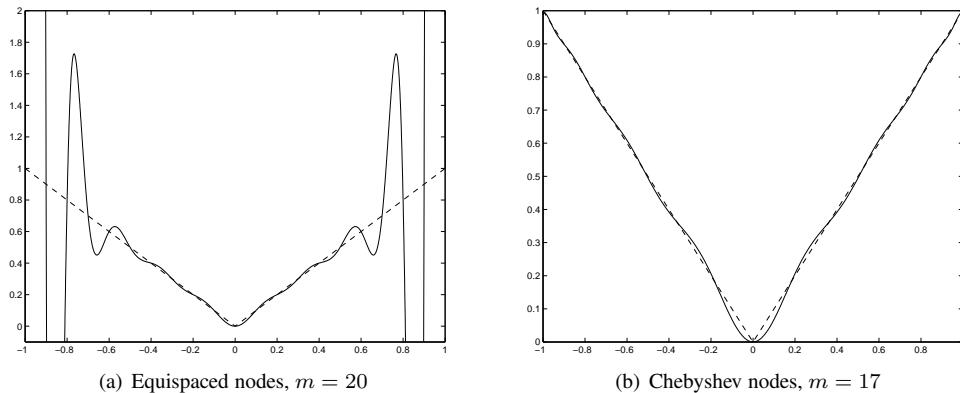


Figure 5.13: Behavior of Lagrange interpolation for  $f : [-1, 1] \rightarrow \mathbb{R}$ ,  $f(x) = |x|$ .

The problem of convergence was solved for the general case by Faber and Bernstein during 1914 and 1916. Faber has proved that for each triangular array of nodes of type 5.8.1 in  $[a, b]$  there exists a function  $f \in C[a, b]$  such that the sequence of Lagrange interpolation polynomials  $L_m f$  for the nodes  $x_i^{(m)}$  (row wise) does not converge uniformly to  $f$  on  $[a, b]$ .

Bernstein<sup>12</sup> has proved that for any array of nodes as before there exists a function  $f \in C[a, b]$  such that the corresponding sequence  $(L_m f)$  is divergent.

Remedies:

- Local approach – the interval  $[a, b]$  is taken very small – the approach used to numerical solution of differential equations;
- Spline interpolation – the interpolant is piecewise polynomial.

## 5.9 Spline Interpolation

Let  $\Delta$  be a subdivision upon the interval  $[a, b]$

$$\Delta : a = x_1 < x_2 < \dots < x_{n-1} < x_n = b \quad (5.9.1)$$

We shall use low-degree polynomials on each subinterval  $[x_i, x_{i+1}]$ ,  $i = \overline{1, n-1}$ . The rationale behind this is the recognition that on a sufficiently small interval, functions can be approximated arbitrarily by polynomials of low degree, even degree 1, or 0 for that matter.

Sergi Natanovitch Bernstein (1880-1968) made major contribution to polynomial approximation, continuing the tradition of Chebyshev. In 1911 he introduced what are now called the Bernstein polynomials<sup>12</sup> to give a constructive proof of Weierstrass's theorem (1885), namely that a continuous function on a finite subinterval of the real line can be uniformly approximated as closely as we wish by a polynomial. He is also known for his works on differential equations and probability theory.



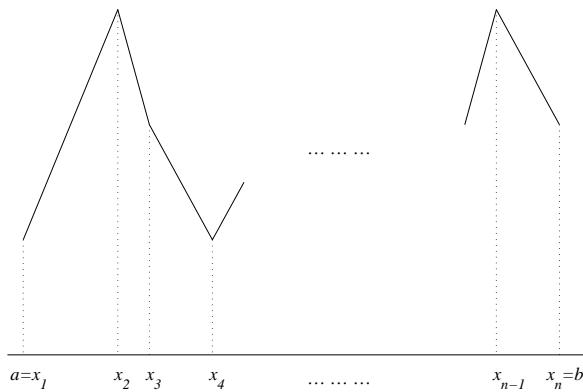


Figure 5.14: Piecewise linear interpolation

We have already introduced the space

$$\mathbb{S}_m^k(\Delta) = \{s : s \in C^k[a, b], s|_{[x_i, x_{i+1}]} \in \mathbb{P}_m, i = 1, 2, \dots, n - 1\} \quad (5.9.2)$$

$m \geq 0$ ,  $k \in \mathbb{N} \cup \{-1\}$ , of spline functions of degree  $m$  and smoothness class  $k$  relative to the subdivision  $\Delta$ . If  $k = m$ , then functions  $s \in \mathbb{S}_m^m(\Delta)$  are polynomials.

For  $m = 1$  and  $k = 0$  one obtains *linear splines*.

We wish to find  $s \in \mathbb{S}_1^0(\Delta)$  such that

$$s(x_i) = f_i, \text{ where } f_i = f(x_i), \quad i = 1, 2, \dots, n.$$

The solution is trivial, see Figure 5.14. On the interval  $[x_i, x_{i+1}]$

$$s(f; x) = f_i + (x - x_i)f[x_i, x_{i+1}], \quad (5.9.3)$$

and

$$|f(x) - s(f(x))| \leq \frac{(\Delta x_i)^2}{8} \max_{x \in [x_i, x_{i+1}]} |f''(x)|. \quad (5.9.4)$$

It follows that

$$\|f(\cdot) - s(f, \cdot)\|_\infty \leq \frac{1}{8} |\Delta|^2 \|f''\|_\infty. \quad (5.9.5)$$

The dimension of  $\mathbb{S}_1^0(\Delta)$  can be computed in the following way: since we have  $n - 1$  subintervals, each linear piece has 2 coefficients (2 degrees of freedom) and each continuity condition reduces the degree of freedom by 1, we have finally

$$\dim \mathbb{S}_1^0(\Delta) = 2n - 2 - (n - 2) = n.$$

A basis of this space is given by the so-called *B-spline* functions:

We let  $x_0 = x_1, x_{n+1} = x_n$ , for  $i = \overline{1, n}$

$$B_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{for } x_{i-1} \leq x \leq x_i \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{for } x_i \leq x \leq x_{i+1} \\ 0, & \text{otherwise} \end{cases} \quad (5.9.6)$$

Note that the first equation, when  $i = 1$ , and the second when  $i = n$  are to be ignored. The functions  $B_i$  may be referred to as “hat functions” (Chinese hats), but note that the first and the last hat is cut in half. The functions  $B_i$  are depicted in Figure 5.15.

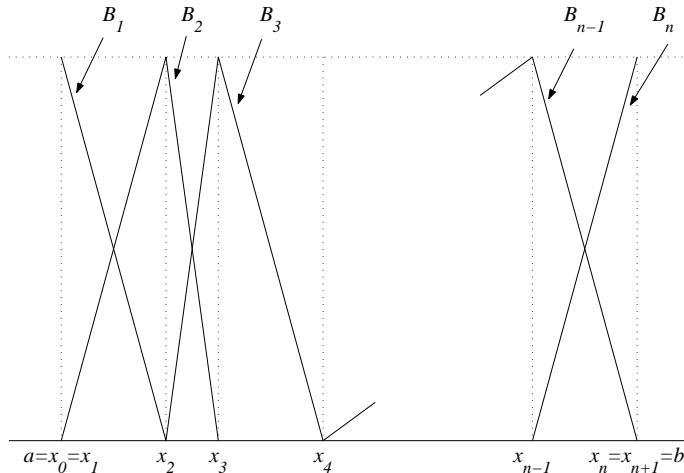


Figure 5.15: First degree B-spline functions

They have the property

$$B_i(x_j) = \delta_{ij},$$

are linear independent, since

$$s(x) = \sum_{i=1}^n c_i B_i(x) = 0 \wedge x \neq x_j \Rightarrow c_j = 0.$$

and

$$\langle B_i \rangle_{i=\overline{1,n}} = S_1^0(\Delta),$$

$B_i$  plays the same role as elementary Lagrange polynomials  $\ell_i$ .

### 5.9.1 Interpolation by cubic splines

Cubic spline are the most widely used. We first discuss the interpolation problem for  $s \in \mathbb{S}_3^1(\Delta)$ . Continuity of the first derivative of any cubic spline interpolant  $s_3(f; \cdot)$  can be enforced by prescribing

the values of the first derivative at each point  $x_i, i = 1, 2, \dots, n$ . Thus, let  $m_1, m_2, \dots, m_n$  be arbitrary given numbers, and denote

$$s_3(f; \cdot)|_{[x_i, x_{i+1}]} = p_i(x), \quad i = 1, 2, \dots, n-1 \quad (5.9.7)$$

Then we enforce  $s_3(f; x_i) = m_i, i = \overline{1, n}$ , by selecting each piece  $p_i$  of  $s_3(f, \cdot)$  to be the (unique) solution of a Hermite interpolation problem, namely,

$$\begin{aligned} p_i(x_i) &= f_i, & p_i(x_{i+1}) &= f_{i+1}, & i &= \overline{1, n-1}, \\ p'_i(x_i) &= m_i, & p'_i(x_{i+1}) &= m_{i+1} \end{aligned} \quad (5.9.8)$$

We solve the problem by Newton's interpolation formula. The required divided differences are

$$\begin{array}{cccccc} x_i & f_i & m_i & \frac{f[x_i, x_{i+1}] - m_i}{\Delta x_i} & \frac{m_{i+1} + m_i - 2f[x_i, x_{i+1}]}{(\Delta x_i)^2} \\ x_i & f_i & f[x_i, x_{i+1}] & \frac{m_{i+1} - f[x_i, x_{i+1}]}{\Delta x_i} \\ x_{i+1} & f_{i+1} & m_{i+1} \\ x_{i+1} & f_{i+1} & \end{array}$$

and the Hermite interpolation polynomial (in Newton form) is

$$\begin{aligned} p_i(x) &= f_i + (x - x_i)m_i + (x - x_i)^2 \frac{f[x_i, x_{i+1}] - m_i}{\Delta x_i} + \\ &\quad + (x - x_i)^2(x - x_{i+1}) \frac{m_{i+1} + m_i - 2f[x_i, x_{i+1}]}{(\Delta x_i)^2}. \end{aligned}$$

Alternatively, in Taylor's form, we can write for  $x_i \leq x \leq x_{i+1}$

$$p_i(x) = c_{i,0} + c_{i,1}(x - x_i) + c_{i,2}(x - x_i)^2 + c_{i,3}(x - x_i)^3 \quad (5.9.9)$$

and since  $x - x_{i+1} = x - x_i - \Delta x_i$ , by identification we have

$$\begin{aligned} c_{i,0} &= f_i \\ c_{i,1} &= m_i \\ c_{i,2} &= \frac{f[x_i, x_{i+1}] - m_i}{\Delta x_i} - c_{i,3}\Delta x_i \\ c_{i,3} &= \frac{m_{i+1} + m_i - 2f[x_i, x_{i+1}]}{(\Delta x_i)^2} \end{aligned} \quad (5.9.10)$$

Thus to compute  $s_3(f; x)$  for any given  $x \in [a, b]$  that is not an interpolation node, one first locates the interval  $[x_i, x_{i+1}] \ni x$  and then computes the corresponding piece (5.9.7) by (5.9.9) and (5.9.10).

We now discuss some possible choices of the parameters  $m_1, m_2, \dots, m_n$ .

### Piecewise cubic Hermite interpolation

Here one selects  $m_i = f'(x_i)$  (assuming that these derivative values are known). This gives rise to a strictly local scheme, in that each piece  $p_i$  can be determined independently from the others. Furthermore, the interpolation error is, for

$$|f(x) - p_i(x)| \leq \left( \frac{1}{2} \Delta x_i \right)^4 \max_{x \in [x_i, x_{i+1}]} \frac{|f^{(4)}(x)|}{4!}, \quad x_i \leq x \leq x_{i+1}. \quad (5.9.11)$$

Hence,

$$\|f(\cdot) - s_3(f; \cdot)\|_\infty \leq \frac{1}{384} |\Delta|^4 \|f^{(4)}\|_\infty. \quad (5.9.12)$$

For equally spaced points  $x_i$ , one has  $|\Delta| = (b - a)/(n - 1)$  and, therefore

$$\|f(\cdot) - s_3(f; \cdot)\|_\infty = O(n^{-4}), \quad n \rightarrow \infty. \quad (5.9.13)$$

### Cubic spline interpolation

Here we require  $s_3(f; \cdot) \in \mathbb{S}_3^2(\Delta)$ , that is, continuity of the second derivatives. In terms of pieces (5.9.7) of  $s_3(f; \cdot)$ , this means that

$$p''_{i-1}(x_i) = p''_i(x_i), \quad i = \overline{2, n-1}, \quad (5.9.14)$$

and translates into a condition for the Taylor coefficients in (5.9.9), namely

$$2c_{i-1,2} + 6c_{i-1,3}\Delta x_{i-1} = 2c_{i,2}, \quad i = \overline{2, n-1}.$$

Plugging in explicit values (5.9.10) for these coefficients, we arrived at the linear system

$$\Delta x_i m_{i-1} + 2(\Delta x_{i-1} + \Delta x_i)m_i + (\Delta x_{i-1})m_{i+1} = b_i, \quad i = \overline{2, n-1} \quad (5.9.15)$$

where

$$b_i = 3\{\Delta x_i f[x_{i-1}, x_i] + \Delta x_{i-1} f[x_i, x_{i+1}]\} \quad (5.9.16)$$

These are  $n - 2$  linear equations in the  $n$  unknowns  $m_1, m_2, \dots, m_n$ . Once  $m_1$  and  $m_n$  have been chosen in some way, the system becomes again tridiagonal in the remaining unknowns and hence is readily solved by Gaussian elimination, by factorization or by an iterative method.

Here are some possible choices of  $m_1$  and  $m_n$ .

**Complete (clamped) spline.** We take  $m_1 = f'(a)$ ,  $m_n = f'(b)$ . It is known that for this spline, if  $f \in C^4[a, b]$ ,

$$\|f^{(r)}(\cdot) - s^{(r)}(f; \cdot)\|_\infty \leq c_r |\Delta|^{4-r} \|f^{(n)}\|_\infty, \quad r = 0, 1, 2, 3 \quad (5.9.17)$$

where  $c_0 = \frac{5}{384}$ ,  $c_1 = \frac{1}{24}$ ,  $c_2 = \frac{3}{8}$ , and  $c_3$  is a constant depending on the ratio  $\frac{|\Delta|}{\min_i \Delta x_i}$ .

**Matching of the second derivatives at the endpoints.** For this type of spline, we enforce the conditions  $s_3''(f; a) = f''(a)$ ;  $s_3''(f; b) = f''(b)$ . Each of these conditions gives rise to an additional equation, namely,

$$\begin{aligned} 2m_1 + m_2 &= 3f[x_1, x_2] - \frac{1}{2}f''(a)\Delta x_1 \\ m_{n-1} + 2m_n &= 3f[x_{n-1}, x_n] + \frac{1}{2}f''(b)\Delta x_{n-1} \end{aligned} \quad (5.9.18)$$

One conveniently adjoins the first equation to the top of the system (5.9.15), and the second to the bottom, thereby preserving the tridiagonal structure of the system.

**Natural cubic spline.** Enforcing  $s''(f; a) = s''(f; b) = 0$ , one obtains two additional equations, which can be obtained from (5.9.18) by putting there  $f''(a) = f''(b) = 0$ . The nice thing about this spline is that it requires only function values of  $f$  – no derivatives – but the price one pays is a degradation of the accuracy to  $O(|\Delta|^2)$  near the endpoints (unless indeed  $f''(a) = f''(b) = 0$ ).

**"Not-a-knot spline".** (C. deBoor). Here we impose the conditions  $p_1(x) \equiv p_2(x)$  and  $p_{n-2}(x) \equiv p_{n-1}(x)$ ; that is, the first two pieces and the last two should be the same polynomial. In effect, this means that the first interior knot  $x_2$ , and the last one  $x_{n-1}$  both are inactive. This again gives rise to two supplementary equations expressing the continuity of  $s_3''(f; x)$  in  $x = x_2$  and  $x = x_{n-1}$ . The continuity condition of  $s_3(f, \cdot)$  at  $x_2$  and  $x_{n-1}$  implies the equality of the leading coefficients  $c_{1,3} = c_{2,3}$  and  $c_{n-2,3} = c_{n-1,3}$ . This gives rise to the equations

$$\begin{aligned} (\Delta x_2)^2 m_1 + [(\Delta x_2)^2 - (\Delta x_1)^2]m_2 - (\Delta x_1)^2 m_3 &= \beta_1 \\ (\Delta x_2)^2 m_{n-2} + [(\Delta x_2)^2 - (\Delta x_1)^2]m_{n-1} - (\Delta x_1)^2 m_n &= \beta_2, \end{aligned}$$

where

$$\begin{aligned}\beta_1 &= 2\{(\Delta x_2)^2 f[x_1, x_2] - (\Delta x_1)^2 f[x_2, x_3]\} \\ \beta_2 &= 2\{(\Delta x_{n-1})^2 f[x_{n-2}, x_{n-1}] - (\Delta x_{n-2})^2 f[x_{n-1}, x_n]\}.\end{aligned}$$

The first equation adjoins to the top of the system  $n - 2$  equations in  $n$  unknowns given by (5.9.15) and (5.9.16) and the second to the bottom. The system is no more tridiagonal, but it can be turned into a tridiagonal one, by combining equations 1 and 2, and  $n - 1$  and  $n$ , respectively. After this transformations, the first and the last equations become

$$\Delta x_2 m_1 + (\Delta x_2 + \Delta x_1) m_2 = \gamma_1 \quad (5.9.19)$$

$$(\Delta x_{n-1} + \Delta x_{n-2}) m_{n-1} + \Delta x_{n-2} m_n = \gamma_2, \quad (5.9.20)$$

where

$$\begin{aligned}\gamma_1 &= \frac{1}{\Delta x_2 + \Delta x_1} \{f[x_1, x_2] \Delta x_2 [\Delta x_1 + 2(\Delta x_1 + \Delta x_2)] + (\Delta x_1)^2 f[x_2, x_3]\} \\ \gamma_2 &= \frac{1}{\Delta x_{n-1} + \Delta x_{n-2}} \{\Delta x_{n-1}^2 f[x_{n-2}, x_{n-1}] + [2(\Delta x_{n-1} + \Delta x_{n-2}) + \Delta x_{n-1}] \Delta x_{n-2} f[x_{n-1}, x_n]\}.\end{aligned}$$

The function `CubicSpline` computes the coefficients of all four spline types presented. It builds the tridiagonal system with sparse matrices and solve it using the `\operatorname{operator}`. The differences between types occur in first and last equations, implemented via a `switch` instruction.

```
function [a,b,c,d]=CubicSpline(x,f,type,der)
%CUBICSPLINE - find coefficients of a cubic spline
%call [a,b,c,d]=Splinecubic(x,f,type,der)
%x - abscissas
%f - ordinates
%type - 0 complete (clamped)
%      1 match second derivatives
%      2 natural
%      3 not a knot (deBoor)
%der - values of derivatives
%      [f'(a),f'(b)] for type 0
%      [f''(a), f''(b)] for type 1

if ( nargin<4 ) | (type==2), der=[0,0]; end

n=length(x);
%sort nodes
if any(diff(x)<0), [x,ind]=sort(x); else, ind=1:n; end
y=f(ind); x=x(:); y=y(:);
%find equations 2 ... n-1
dx=diff(x); ddiv=diff(y)./dx;
ds=dx(1:end-1); dd=dx(2:end);
dp=2*(ds+dd);
md=3*(dd.*ddiv(1:end-1)+ds.*ddiv(2:end));
```

```
%treat separately over type - equations 1,n
switch type
case 0 %complete
    dp1=1; dpn=1; vd1=0; vdn=0;
    md1=der(1); mdn=der(2);
case 1,2 %d2 si natural
    dp1=2; dpn=2; vd1=1; vdn=1;
    md1=3*ddiv(1)-0.5*dx(1)*der(1);
    mdn=3*ddiv(end)+0.5*dx(end)*der(2);
case 3 %deBoor
    x31=x(3)-x(1); xn=x(n)-x(n-2);
    dp1=dx(2); dpn=dx(end-1);
    vd1=x31; vdn=xn;
    md1=((dx(1)+2*x31)*dx(2)*ddiv(1)+dx(1)^2*ddiv(2))/x31;
    mdn=(dx(end)^2*ddiv(end-1)+(2*xn+dx(end))*dx(end-1)...
        ddiv(end))/xn;
end
%build sparse system
dp=[dp1;dp;dpn]; dp1=[0;vd1;dd];
dm1=[ds;vdn;0]; md=[md1;md;mdn];
A=spdiags([dm1,dp,dp1],-1:1,n,n);
m=A \ md;
d=y(1:end-1);
c=m(1:end-1);
a=[(m(2:end)+m(1:end-1)-2*ddiv)./(dx.^2)];
b=[(ddiv-m(1:end-1))./dx-dx.*a];
```

The values of spline functions are computed using a single function for all types:

```
function z=evalspline(x,a,b,c,d,t)
%EVALSPLINE - compute cubic spline value
%call z=evalspline(x,a,b,c,d,t)
%z - values
%t - evaluation points
%x - nodes (knots)
%a,b,c,d - coefficients
n=length(x);
x=x(:); t=t(:);
k = ones(size(t));
for j = 2:n-1
    k(x(j) <= t) = j;
end
% interpolant evaluation
s = t - x(k);
z = d(k) + s.* (c(k) + s.* (b(k) + s.* a(k)));
```

### 5.9.2 Minimality properties of cubic spline interpolants

The complete and natural splines have interesting optimality properties. To formulate them, it is convenient to consider not only the subdivision  $\Delta$  in (5.9.1), but also the subdivision

$$\Delta' : a = x_0 = x_1 < x_2 < x_3 < \cdots < x_{n-1} < x_n = x_{n+1} = b, \quad (5.9.21)$$

in which the endpoints are double knots. This means that whenever we interpolate on  $\Delta'$ , we interpolate to function values at all interior points, but to the functions as well as first derivative values at the endpoints. The first of the two theorems relates to the complete cubic spline interpolant,  $s_{compl}(f; \cdot)$ .

**Theorem 5.9.1.** *For any function  $g \in C^2[a, b]$  that interpolates  $f$  on  $\Delta'$ , there holds*

$$\int_a^b [g''(x)]^2 dx \geq \int_a^b [s''_{compl}(f; x)]^2 dx, \quad (5.9.22)$$

with equality if and only if  $g(\cdot) = s_{compl}(f; \cdot)$ .

**Remark 5.9.2.**  $s_{compl}(f; \cdot)$  in Theorem 5.9.1 also interpolates  $f$  on  $\Delta'$  and among all such interpolants its second derivative has the smallest  $L^2$  norm.  $\diamond$

*Proof.* We write (for short)  $s_{compl} = s$ . The theorem follows once we have shown that

$$\int_a^b [g''(x)]^2 dx = \int_a^b [g''(x) - s''(x)]^2 dx + \int_a^b [s''(x)]^2 dx. \quad (5.9.23)$$

Indeed, this immediately implies (5.9.22), and equality in (5.9.22) holds if and only if  $g''(x) - s''(x) \equiv 0$ , which, integrating twice from  $a$  to  $x$  and using the interpolation properties of  $s$  and  $g$  at  $x = a$  gives  $g(x) \equiv s(x)$ .

To complete the proof, note that the relation (5.9.23) is equivalent to

$$\int_a^b s''(x)[g''(x) - s''(x)] dx = 0. \quad (5.9.24)$$

Integrating by parts and taking into account that  $s'(b) = g'(b) = f'(b)$  and  $s'(a) = g'(a) = f'(a)$ , we get

$$\begin{aligned} & \int_a^b s''(x)[g''(x) - s''(x)] dx = \\ &= s''(x)[g'(x) - s'(x)] \Big|_a^b - \int_a^b s'''(x)[g'(x) - s'(x)] dx \\ &= - \int_a^b s'''(x)[g'(x) - s'(x)] dx. \end{aligned} \quad (5.9.25)$$

But  $s'''$  is piecewise constant, so

$$\begin{aligned} & \int_a^b s'''(x)[g'(x) - s'(x)] dx = \sum_{\nu=1}^{n-1} s'''(x_\nu + 0) \int_{x_\nu}^{x_{\nu+1}} [g'(x) - s'(x)] dx = \\ &= \sum_{\nu=1}^{n-1} s'''(x_\nu + 0) [g(x_{\nu+1}) - s(x_{\nu+1}) - (g(x_\nu) - s(x_\nu))] = 0 \end{aligned}$$

since both  $s$  and  $g$  interpolate  $f$  on  $\Delta$ . This proves (5.9.24) and hence the theorem.  $\square$

For interpolation on  $\Delta$ , the distinction of being optimal goes to the natural cubic spline interpolant  $s_{nat}(f; \cdot)$ .

**Theorem 5.9.3.** *For any function  $g \in C^2[a, b]$  that interpolates  $f$  on  $\Delta$  (not  $\Delta'$ ), there holds*

$$\int_a^b [g''(x)]^2 dx \geq \int_a^b [s''_{nat}(f; x)]^2 dx \quad (5.9.26)$$

with equality if and only if  $g(\cdot) = s_{nat}(f; \cdot)$ .

The proof of Theorem 5.9.3 is virtually the same as that of Theorem 5.9.1, since (5.9.23) holds again, this time because  $s''(b) = s''(a) = 0$ .

Putting  $g(\cdot) = s_{compl}(f; \cdot)$  in Theorem 5.9.3 immediately gives

$$\int_a^b [s''_{compl}(f; x)]^2 dx \geq \int_a^b [s''_{nat}(f; x)]^2 dx. \quad (5.9.27)$$

Therefore, in a sense, the natural cubic spline is the “smoothest” interpolant.

The property expressed in Theorem 5.9.3 is the origin of the name “spline”. A spline is a flexible strip of wood used in drawing curves. If its shape is given by the equation  $y = g(x)$ ,  $x \in [a, b]$  and if the spline is constrained to pass through the points  $(x_i, g_i)$ , then it assumes a form that minimizes the bending energy

$$\int_a^b \frac{[g''(x)]^2 dx}{(1 + [g'(x)]^2)^3},$$

over all functions  $g$  similarly constrained. For slowly varying  $g$  ( $\|g'\|_\infty \ll 1$ ) this is nearly the same as the minimum property of Theorem 5.9.3.

## 5.10 Interpolation in MATLAB

MATLAB has functions for interpolation in one, two and more dimensions. The `polyfit` function returns the coefficients of Lagrange interpolation polynomial if the degree  $n$  equates the number of observations minus 1.

Function `interp1` accepts  $x(i)$ ,  $y(i)$  data pairs and a vector  $xi$  as input parameters. It fits an interpolant to the data and then returns the values of the interpolant at the points in  $xi$ :

```
yi = interp1(x, y, xi, method)
```

The elements of  $x$  must be in increasing order. Four types of interpolant are supported, as specified by a fourth input parameter, `method` which is one of

- ‘nearest’ – nearest neighbor interpolation
- ‘linear’ – linear interpolation
- ‘spline’ – piecewise cubic spline interpolation (SPLINE)
- ‘pchip’ – shape-preserving piecewise cubic interpolation
- ‘cubic’ – same as ‘pchip’
- ‘v5cubic’ – the cubic interpolation from MATLAB 5, which does not extrapolate and uses ‘spline’ if  $X$  is not equally spaced.

The example below illustrates `interp1` (file `exinterp1.m`). It chooses six points on graph of  $f(x) = x + \sin \pi x^2$  and computes the interpolant using `nearest`, `linear` and `spline` methods. The `cubic` interpolant was omitted, since it is close to `spline` interpolant and it would complicate the picture. The graph is given in Figure 5.16.

```

x=[-1,-3/4, -1/3, 0, 1/2, 1]; y=x+sin(pi*x.^2);
xi=linspace(-1,1,60); yi=xi+sin(pi*xi.^2);
yn=interp1(x,y,xi,'nearest');
yl=interp1(x,y,xi,'linear');
ys=interp1(x,y,xi,'spline');
%yc=interp1(x,y,xi,'pchip');
plot(xi,yi,:',x,y,'o','MarkerSize',12); hold on
plot(xi,yl,'--',xi,ys,'-')
stairs(xi,yn,'-')
set(gca,'XTick',x);
set(gca,'XTickLabel','|-3/4|-1/3|0|1/2|1')
set(gca,'XGrid','on')
axis([-1.1, 1.1, -1.1, 2.1])
legend('f','data','linear', 'spline', 'nearest',4)
hold off

```

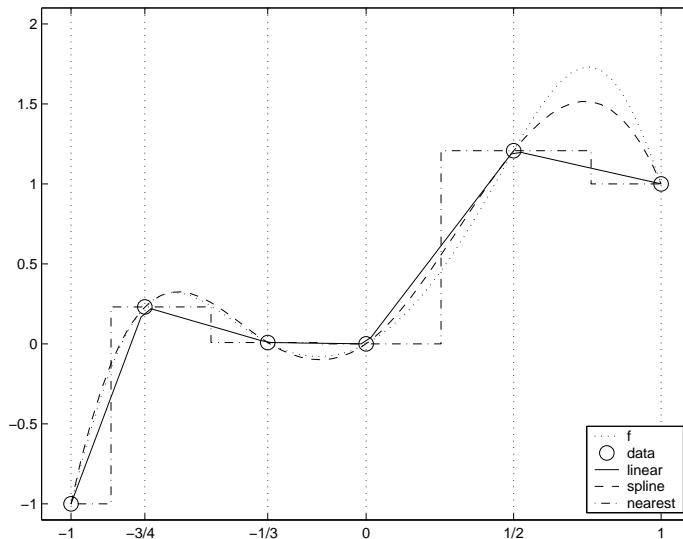


Figure 5.16: Interpolating a curve using `interp1`

The smoothest interpolation is `spline`, but piecewise Hermite interpolation preserves the shape. The next example illustrates the differences between `spline` and piecewise Hermite interpolation (file `expсли_cub.m`).

```

x = [-0.99, -0.76, -0.48, -0.18, 0.07, 0.2, ...
       0.46, 0.7, 0.84, 1.09, 1.45];
y = [0.39, 1.1, 0.61, -0.02, -0.33, 0.65, ...
       1.13, 1.46, 1.07, 1.2, 0.3];
plot(x,y,'o'); hold on
xi=linspace(min(x),max(x),100);
ys=interp1(x,y,xi,'spline');

```

```

yc=interp1(x,y,xi,'cubic');
h=plot(xi,ys,'-',xi,yc,'-.');
legend(h,'spline','cubic',4)
axis([-1.1,1.6,-0.8,1.6])

```

Figure 5.17 gives the corresponding graph.

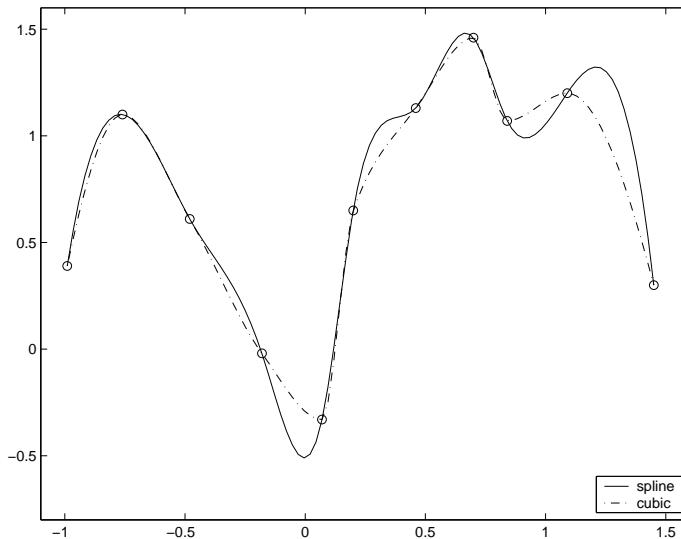


Figure 5.17: Cubic spline and piecewise Hermite interpolation

We can perform cubic spline and piecewise Hermite interpolation directly, by calling `spline` and `pchip`. function, respectively.

Given the vectors `x` and `y`, the command `yy = spline(x, y, xx)` returns in `yy` vector the values of cubic spline interpolant at points in `xx`. If `y` is a matrix, we have vector values interpolation and interpolation is done after column of `y`; the size of `yy` is `length(xx)` over `size(y, 2)`. The `spline` function computes a deBoor-type spline interpolant. If `y` is a vector that contains two more values than `x` has entries, the first and last value in `y` are used as the endslopes for the cubic spline. (For terminology on spline type see Section 5.9.1).

The next example takes six points on the graph of  $y = \sin(x)$ , determines and plots the graph of deBoor spline and complete spline (see Figure 5.18).

```

x = 0:2:10;
y = sin(x); yc=[cos(0),y,cos(10)];
xx = 0.:0.01:10;
yy = spline(x,y,xx);
yc = spline(x,yc,xx);
plot(x,y,'o',xx,sin(xx),'-',xx,yy,'--',xx,yc,'-.')
axis([-0.5,10.5,-1.3,1.3])
legend('nodes','sine','deBoor','complete',4)

```

There are situations which require the handling of spline coefficients (for example, if nodes are preserved and `xx` changes). The command `pp=spline(x,y)` stores the coefficients in a `pp` (piece-

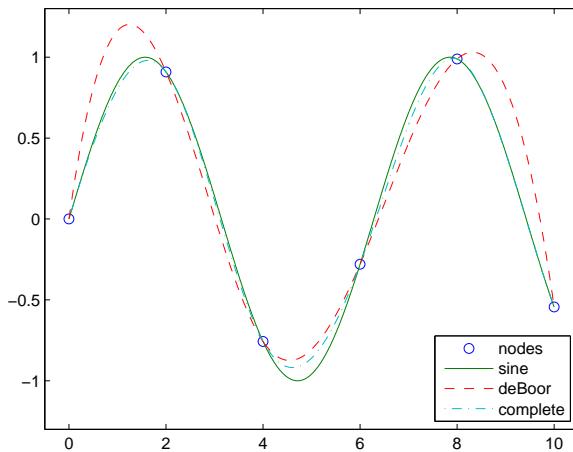


Figure 5.18: deBoor and complete spline

wise polynomial) structure, that contains the shape, coefficient matrix (a four column matrix for cubic splines), number of subintervals, order (degree plus one) and the size. The `ppval` function evaluates the spline using such a structure. Other low-level operations are possible via `mkpp` (make a pp-structure) and `unmkpp` (details about the components of a pp-structure). For example, the command `yy = spline(x,y,xx)` can be replaced by

```
pp = spline(x,y);
yy = ppval(pp,xx);
```

We shall give now an example of vector spline interpolation and use of `ppval`. We wish to introduce interactively in graphical mode a set of data points from the screen and to plot the parametric spline that passes through these points, with two different resolutions (say 20 and 150 intermediary points on the curve). Here is the source (M-file `splinevect.m`):

```
axis([0,1,0,1]);
hold on
[x,y]=ginput;
data=[x';y'];
t=linspace(0,1,length(x));
tt1=linspace(0,1,20);
tt2=linspace(0,1,150);
pp=spline(t,data);
yy1=ppval(pp,tt1);
yy2=ppval(pp,tt2);
plot(x,y,'o',yy1(1,:),yy1(2,:),yy2(1,:),yy2(2,:));
hold off
```

Interactive input is performed by `ginput`. The spline coefficients are computed only once. For the evaluation we use `ppval`. We recommend the user to try this example.

We have two forms of call syntax for `pchip` function:

```
yi = pchip(x,y,xx)
pp = pchip(x,y)
```

The first form returns the values of interpolant at `xx` in `yi`, while the second returns a `pp`-structure. The meaning of parameters is the same as for `spline` function. The next example computes the deBoor and piecewise Hermite cubic interpolant for the same data set (script `expchip.m`):

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend({'data','pchip','spline'},4)
```

The graph appears in Figure 5.19. The spline interpolant is smoother, but the piecewise Hermite interpolant preserves the shape.

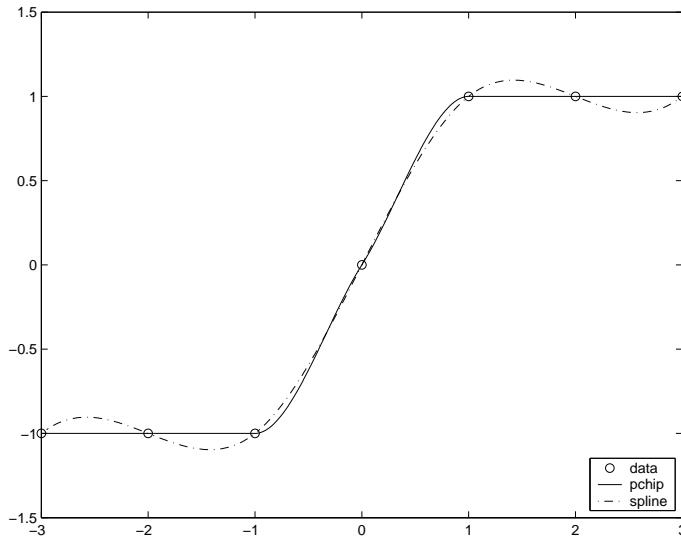


Figure 5.19: `pchip` and `spline` example

## 5.11 Applications

### 5.11.1 Spline and sewing machines

The basic idea of this application [83] is to use a parametric representation  $(x(s), y(s))$  for the curve and approximate independently the coordinate functions  $x(s)$  and  $y(s)$ . The parameter  $s$  can be anything, but a proper choice is the arc length. Having chosen the nodes  $s_i$ ,  $i = 1, \dots, n$ , we can interpolate

the data  $x_i = x(s_i)$  by a spline  $S_x(s)$  and likewise the data  $y_i = y(s_i)$  by a spline  $S_y(s)$ . The curve  $(x(s), y(s))$  is approximated by  $(S_x(s), S_y(s))$ . This yields a curve in the plane that passes through all the points  $(x(s_i), y(s_i))$  in order.

A problem is the computation of parameter  $s$ . We take  $s_1 = 0$  and

$$s_{i+1} = s_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}. \quad (5.11.1)$$

An alternative is

$$s_{i+1} = s_i + |x_{i+1} - x_i| + |y_{i+1} - y_i|. \quad (5.11.2)$$

An interesting example of the technique is furnished by a need to approximate curves for automatic control of sewing machines. Arc length is a natural parameter because a constant increment in arc length corresponds to a constant stitch length.

An example taken from [83] fits the data  $(2.5, -2.5), (3.5, -0.5), (5.0, 2.0), (7.5, 4.0), (9.5, 4.5), (11.8, 3.5), (13.0, 0.5), (11.5, -2.0), (9.0, -3.0), (6.0, -3.3), (2.5, -2.5), (0.0, 0.0), (-1.5, 2.0), (-3.0, 5.0), (-3.5, 9.0), (-2.0, 11.0), (0.0, 11.5), (2.0, 11.0), (3.5, 9.0), (3.0, 5.0), (1.5, 2.0), (0.0, 0.0), (-2.5, -2.5), (-6.0, -3.3), (-9.0, -3.0), (-11.5, -2.0), (-13.0, 0.5), (-11.8, 3.5), (-9.5, 4.5), (-7.5, 4.0), (-5.0, 2.0), (-3.5, -0.5), (-2.5, -2.5)$ . The resulting spline curve is shown in Figure 5.20

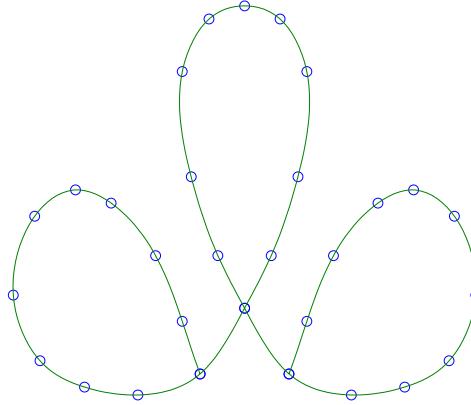


Figure 5.20: Curve fit for a sewing machine pattern

The MATLAB code to generate Figure 5.20 is

```
load sm
s=zeros(length(sm),1);
s(2:end)=sqrt(diff(sm(:,1)).^2+diff(sm(:,2)).^2);
s=cumsum(s);
t=linspace(s(1),s(end),300);
xg=spline(s,sm(:,1),t);
yg=spline(s,sm(:,2),t);
plot(sm(:,1),sm(:,2),'o',xg,yg)
```

```
axis off
```

The file sm.mat contains the points to be fitted. If we want an arc increment given by (5.11.2), line 3 can be replaced by

```
s(2:end)=abs(diff(sm(:,1)))+abs(diff(sm(:,2)));
```

The result with this modification is very close to the result obtained using (5.11.1).

## 5.11.2 A membrane deflection problem

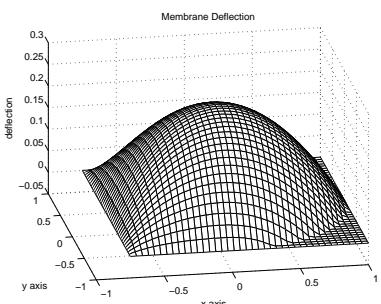
Least squares approximation has numerous applications when we are dealing with a large number of equations involving a smaller number of parameters used to closely fit some constraints. Let us illustrate how least squares approximation can be used to compute the transverse deflection of a membrane subjected to uniform pressure [105]. The transverse deflection  $u$  for a membrane which has zero deflection on a boundary  $L$  satisfies the differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\gamma,$$

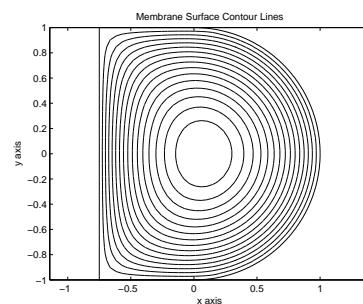
where  $(x, y)$  is inside  $L$  and  $\gamma$  is a physical constant. The differential equation is satisfied by a series of the form

$$u = \gamma \left[ \frac{-|z|^2}{4} + \sum_{j=1}^n c_j \operatorname{real}(z^{j-1}) \right],$$

where  $z = x + iy$  and constants  $c_j$  are chosen to minimize the boundary deflection in the least squares sense. As example, we analyze a membrane consisting of a rectangular part on the left joined with a semicircular part on the right. The surface plot in Figure 5.21(a) and the contour plot in Figure 5.21(b) were produced by the function membran listed below. The function generates boundary data, solves the series coefficients, and constructs plots depicting the deflection pattern. The results obtained using twenty-term series satisfy the boundary condition quite well.



(a) Surface plot of membrane



(b) Contour plot of membrane

Figure 5.21: Membrane plot

```
function [df1,cof]=membran(h,np,ns,nx,ny)
%MEMBRAN Computes transverse deflection of a
% uniformly tensioned membrane
```

```
% [DFL,COF]=membran(H,NP,NS,NX,NY)
% Example use: membran(0.75,100,50,40,40);
% h - width of the rectangular part
% np - number of least square points (about 3.5*np)
% ns - number of terms
% nx,ny - the number of x points and y points
% dfl - computed array of deflection values
% cof - coefficients in the series

if nargin==0
    h=.75; np=100; ns=50; nx=40; ny=40;
end

% Generate boundary points for least square
% approximation
z=[exp(1i*linspace(0,pi/2,round(1.5*np))),...
    linspace(1i,-h+1i,np),...
    linspace(-h+1i,-h,round(np/2))];
z=z(:); xb=real(z); xb=[xb;xb(end:-1:1)];
yb=imag(z); yb=[yb;-yb(end:-1:1)];

% Form the least square equations and solve
% for series coefficients
a=ones(length(z),ns);
for j=2:ns, a(:,j)=a(:,j-1).*z; end
cof=real(a)\(z.*conj(z))/4;

% Generate a rectangular grid for evaluation
% of deflections
xv=linspace(-h,1,nx); yv=linspace(-1,1,ny);
[X,Y]=meshgrid(xv,yv); Z=X+1i*Y;

% Evaluate the deflection series on the grid
dfl=-Z.*conj(Z)/4+ ...
    real(polyval(cof(ns:-1:1),Z));

% Set values outside the physical region of
% interest to zero
dfl=real(dfl).* (1-(abs(Z)>=1)&(real(Z)>=0));

% Make surface and contour plots
hold off; close; surf(X,Y,dfl);
xlabel('x axis'); ylabel('y axis');
zlabel('deflection'); view(-10,30);
title('Membrane Deflection'); colormap([1 1 1]);
shg
figure(2)
contour(X,Y,dfl,15,'k'); hold on
plot(xb,yb,'k-'); axis('equal'), hold off
```

```
xlabel('x axis'); ylabel('y axis');
title('Membrane Surface Contour Lines'), shg
```

## Problems

**Problem 5.3.** (a) Given a function  $f \in C[a, b]$ , determine  $\hat{s}_1(f; \cdot) \in \mathbb{S}_1^0(\Delta)$  such that

$$\int_a^b [f(x) - \hat{s}_1(f; x)]^2 dx$$

be minimal.

(b) Write a MATLAB function that builds and solves the system of normal equation in (a).

(c) Test the above MATLAB function for a function and a division at your choice.

**Problem 5.4.** Compute discrete least squares approximations for the function  $f(t) = \sin\left(\frac{\pi}{2}t\right)$  on  $0 \leq t \leq 1$  of the form

$$\varphi_n(t) = t + t(1-t) \sum_{j=1}^n c_j t^{j-1}, \quad n = 1(1)5,$$

using  $N$  abscissas  $t_k = k/(N+1)$ ,  $k = \overline{1, N}$ . Note that  $\varphi_n(0) = 0$  and  $\varphi_n(1) = 1$  are the exact values of  $f$  at  $t = 0$ , and  $t = 1$ , respectively.

(Hint: Approximate  $f(t) - t$  by a linear combination of polynomials  $\pi_j(t) = t(1-t)t^{j-1}$ ,  $j = \overline{1, n}$ .) The system of normal equations has the form  $Ac = b$ ,  $A = [(\pi_i, \pi_j)]$ ,  $b = [(\pi_i, f - t)]$ ,  $c = [c_j]$ .

Output (for  $n = 1, 2, \dots, 5$ ) :

- The condition number of the system.
- The values of the coefficients.
- Maximum and minimum of the error:

$$e_{\min} = \min_{1 \leq k \leq N} |\varphi_n(t_k) - f(t_k)|, \quad e_{\max} = \max_{1 \leq k \leq N} |\varphi_n(t_k) - f(t_k)|.$$

Run twice for

- (a)  $N = 5, 10, 20$ ,
- (b)  $N = 4$ .

Comment the results. Plot the function and the approximations on the same graph.

**Problem 5.5.** Plot on the same graph for  $[a, b] = [0, 1]$ ,  $n = 11$ , the function, Lagrange interpolation polynomial, and Hermite interpolation polynomial with double nodes when:

- (a)  $x_i = \frac{i-1}{n-1}$ ,  $i = \overline{1, n}$ ,  $f(x) = e^{-x}$  and  $f(x) = x^{5/2}$ ;
- (b)  $x_i = \left(\frac{i-1}{n-1}\right)^2$ ,  $i = \overline{1, n}$ ,  $f(x) = x^{5/2}$ .

**Problem 5.6.** The same problem as above, but for the four types of cubic interpolation splines.

**Problem 5.7.** Choosing several values of  $n$ , for each chosen  $n$  plot the Lebesgue function for  $n$  equally spaced nodes and  $n$  Chebyshev nodes in interval  $[0, 1]$ .

**Problem 5.8.** Consider the points  $P_i \in \mathbb{R}^2$ ,  $i = \overline{0, n}$ . Write:

- A MATLAB function to find a  $n$ -th degree parametric polynomial that passes through the given points.
- A MATLAB function to find a cubic parametric spline curve that passes through the given points, using the function for natural cubic spline or that for deBoor spline, given in this chapter.

Test these two functions, by introducing the points interactively with `ginput` and then plotting the points and the curves determined in this way.

**Problem 5.9.** Find a parametric cubic curve, that passes through two given points and has given tangent vectors at those given points.

**Problem 5.10.** Write a MATLAB function that computes the coefficients and the values of Hermite cubic spline, that is cubic spline of class  $C^1[a, b]$  which verifies

$$s_3(f, x_i) = f(x_i), \quad s'_3(f, x_i) = f'(x_i), \quad i = \overline{1, n}.$$

Plot on the same graph the function  $f(x) = e^{-x^2}$  and the interpolant of  $f$  for 5 equally-spaced nodes and 5 Chebyshev nodes on  $[0, 1]$ .

**Problem 5.11.** Implement a MATLAB function for the inversion of Vandermonde matrix, using the results from pages 188–191.

**Problem 5.12.** [66] Write a MATLAB function for the computing of a periodic cubic spline of class  $C^2[a, b]$ . This means that our input data must check  $f_n = f_1$ , and the interpolating spline must be periodic, with period  $x_n - x_1$ . The endpoint periodicity condition can be forced easier by considering two additional points  $x_0 = x_1 - \Delta x_{n-1}$  and  $x_{n+1} = x_n + \Delta x_1$ , where the function would have the values  $f_0 = f_{n-1}$  and  $f_{n+1} = f_2$ , respectively.

**Problem 5.13.** Consider the input data

```
x = -5:5;
y = [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0];
```

Find the coefficients of 7th degree least squares polynomial approximation for this data and plot on the same graph the corresponding least squares approximation and the Lagrange interpolation polynomial.

**Problem 5.14.** The density of sodium (in  $\text{kg/m}^3$ ) for three temperatures ( ${}^\circ\text{C}$ ) is given in the table

Temperature	$T_i$	94	205	371
Density	$\rho_i$	929	902	860

- Find the Lagrange interpolation polynomial for this data, using the Symbolic toolbox.
- Find the density for  $T = 251^\circ$  using Lagrange interpolation.

**Problem 5.15.** Approximate

$$y = \frac{1+x}{1+2x+3x^2}$$

for  $x \in [0, 5]$  using Lagrange, Hermite and spline interpolation. Choose five nodes and plot on the same graph the function and the interpolants. Then plot the approximation errors.

$T$	605	645	685	725	765	795	825
$C(T)$	0.622	0.639	0.655	0.668	0.679	0.694	0.730
$T$	845	855	865	875	885	895	905
$C(T)$	0.812	0.907	1.044	1.336	1.181	2.169	2.075
$T$	915	925	935	955	975	1015	1065
$C(T)$	1.598	1.211	0.916	0.672	0.615	0.603	0.601

Table 5.5: A property of titanium as a function of temperature

**Problem 5.16.** Table 5.5 gives the values for a property of titanium as a function of temperature,  $T$ . Find and plot a cubic interpolating spline for this data using 15 nodes. How well does the spline fit the data at the other 6 points?

**Problem 5.17.** Determine a discrete least squares approximation of the form

$$y = \alpha \exp(\beta x)$$

for data

$x$	$y$
0.0129	9.5600
0.0247	8.1845
0.0530	5.2616
0.1550	2.7917
0.3010	2.2611
0.4710	1.7340
0.8020	1.2370
1.2700	1.0674
1.4300	1.1171
2.4600	0.7620

Plot the points and the approximation.

*Hint:* apply logarithm.

**Problem 5.18.** Implement a MATLAB function for discrete least squares approximation based on Chebyshev #1 polynomials and the inner product (5.3.13).

**Problem 5.19.** Determine a discrete least squares approximation of the form

$$y = c_1 + c_2 x + c_3 \sin(\pi x) + c_4 \sin(2\pi x)$$

for data

$i$	$x_i$	$y_i$
1	0.1	0.0000
2	0.2	2.1220
3	0.3	3.0244
4	0.4	3.2568
5	0.5	3.1399
6	0.6	2.8579
7	0.7	2.5140
8	0.8	2.1639
9	0.9	1.8358

Plot data and the approximation.

# CHAPTER 6

---

## Linear Functional Approximation

---

### 6.1 Introduction

Let  $X$  be a linear space,  $L_1, \dots, L_m$  real linear functionals, that are linear independent, defined on  $X$  and  $L : X \rightarrow \mathbb{R}$  be a real linear functional such that  $L, L_1, \dots, L_m$  are linear independent.

**Definition 6.1.1.** A formula for approximation of a linear functional  $L$  with respect to linear functionals  $L_1, \dots, L_m$  is a formula having the form

$$L(f) = \sum_{i=1}^m A_i L_i(f) + R(f), \quad f \in X. \quad (6.1.1)$$

Real parameters  $A_i$  are called coefficients (weights) of the formula, and  $R(f)$  is the remainder term.

For a formula of form (6.1.1), given  $L_i$ , we wish to determine the weights  $A_i$  and to study the remainder term corresponding to these coefficients.

**Remark 6.1.2.** The form of  $L_i$  depends on information on  $f$  available (they really express this information), but also on the nature of the approximation problem, that is, the form of  $L$ . ◇

**Example 6.1.3.** If  $X = \{f \mid f : [a, b] \rightarrow \mathbb{R}\}$ ,  $L_i(f) = f(x_i)$ ,  $i = \overline{0, m}$ ,  $x_i \in [a, b]$ , and  $L(f) = f(\alpha)$ ,  $\alpha \in [a, b]$ , the Lagrange interpolation formula

$$f(\alpha) = \sum_{i=0}^m l_i(\alpha) f(x_i) + (Rf)\alpha$$

provides us an example of type (6.1.1), having the coefficients  $A_i = l_i(\alpha)$ , and a possible representation of the remainder is

$$(Rf)(\alpha) = \frac{u(\alpha)}{(m+1)!} f^{(m+1)}(\xi), \quad \xi \in [a, b],$$

if  $f^{(m+1)}$  exists  $[a, b]$ . ◇

**Example 6.1.4.** If  $X$  and  $L_i$  are like in Example 6.1.3 and  $f^{(k)}(\alpha)$  exists,  $\alpha \in [a, b]$ ,  $k \in \mathbb{N}^*$ , and  $L(f) = f^{(k)}(\alpha)$  one obtains a formula for the approximation of the  $k$ th derivative of  $f$  at  $\alpha$

$$f^{(k)}(\alpha) = \sum_{i=0}^m A_i f(x_i) + R(f),$$

called *numerical differentiation formula*.  $\diamond$

**Example 6.1.5.** If  $X$  is a space of functions which are defined on  $[a, b]$ , integrable on  $[a, b]$  and there exists  $f^{(j)}(x_k)$ ,  $k = \overline{0, m}$ ,  $j \in I_k$ , with  $x_k \in [a, b]$  and  $I_k$  are given sets of indices

$$L_{kj}(f) = f^{(j)}(x_k), \quad k = \overline{0, m}, \quad j \in I_k,$$

and

$$L(f) = \int_a^b f(x) dx,$$

one obtains a formula

$$\int_a^b f(x) dx = \sum_{k=0}^m \sum_{j \in I_k} A_{kj} f^{(j)}(x_k) + R(f),$$

called *numerical integration formula*.  $\diamond$

**Definition 6.1.6.** If  $\mathbb{P}_r \subset X$ , then the number  $r \in \mathbb{N}$  such that  $\text{Ker}(R) = \mathbb{P}_r$  is called degree of exactness of the approximation formula (6.1.1).

**Remark 6.1.7.** Since  $R$  is a linear functional, the property  $\text{Ker}(R) = \mathbb{P}_r$  is equivalent to  $R(e_k) = 0$ ,  $k = \overline{0, r}$  and  $R(e_{r+1}) \neq 0$ , where  $e_k(x) = x^k$ .  $\diamond$

We are now ready to formulate the *general approximation problem*: given a linear functional  $L$  on  $X$ ,  $m$  linear functional  $L_1, L_2, \dots, L_m$  on  $X$  and their values (the “data”)  $l_i = L_i f$ ,  $i = \overline{1, m}$  applied to some function  $f$  and given a linear subspace  $\Phi \subset X$  with  $\dim \Phi = m$ , we want to find an approximation formula of the type

$$L f \approx \sum_{i=1}^m a_i L_i f \tag{6.1.2}$$

that is exact (i.e., holds with equality), whenever  $f \in \Phi$ .

It is natural (since we want to interpolate) to make the following

**Assumption:** the “interpolation problem”

find  $\varphi \in \Phi$  such that

$$L_i \varphi = s_i, \quad i = \overline{1, m} \tag{6.1.3}$$

has a unique solution  $\varphi(\cdot) = \varphi(s, \cdot)$ , for arbitrary  $s = [s_1, \dots, s_m]^T$ .

We can express our assumption more explicitly in terms of a given basis  $\varphi_1, \varphi_2, \dots, \varphi_m$  of  $\Phi$  and

the associated Gram<sup>-1</sup> matrix

$$G = [L_i \varphi_j] = \begin{vmatrix} L_1 \varphi_1 & L_1 \varphi_2 & \dots & L_1 \varphi_m \\ L_2 \varphi_1 & L_2 \varphi_2 & \dots & L_2 \varphi_m \\ \dots & \dots & \dots & \dots \\ L_m \varphi_1 & L_m \varphi_2 & \dots & L_m \varphi_m \end{vmatrix} \in \mathbb{R}^{m \times m}. \quad (6.1.4)$$

What we require is that

$$\det G \neq 0. \quad (6.1.5)$$

It is easily seen that this condition is independent of the particular choice of basis.

To show that unique solvability of (6.1.3) and (6.1.5) are equivalent, we express  $\varphi$  in (6.1.3) as a linear combination of the basis functions

$$\varphi = \sum_{j=1}^{nm} c_j \varphi_j \quad (6.1.6)$$

and note that the interpolation conditions

$$L_i \left( \sum_{j=1}^m c_j \varphi_j \right) = s_i, \quad i = \overline{1, m}$$

by the linearity of the functionals  $L_i$ , can be written in the form

$$\sum_{j=1}^m c_j L_i \varphi_j = s_i, \quad i = \overline{1, m},$$

that is,

$$Gc = s, \quad c = [c_1, c_2, \dots, c_m]^T, \quad s = [s_1, s_2, \dots, s_m]^T. \quad (6.1.7)$$

This has a unique solution for arbitrary  $s$  if and only if (6.1.5) holds.

We have two approaches for the solution of this problem.

### 6.1.1 Method of interpolation

We solve the general approximation problem by interpolation

$$Lf \approx L\varphi(\ell; \cdot), \quad \ell = [\ell_1, \ell_2, \dots, \ell_m]^T, \quad \ell_i = L_i f \quad (6.1.8)$$

In other words, we apply  $L$  not to  $f$ , but to  $\varphi(\ell; \cdot)$  — the solution of the interpolation problem (6.1.3) in which  $s = \ell$ , the given “data”. Our assumption guarantees that  $\varphi(\ell; \cdot)$  is uniquely determined.

1



Jørgen Pedersen Gram (1850-1916), Danish mathematician who studied at the University of Copenhagen. After graduation, he entered an insurance company as computer assistant and, moving up the ranks, eventually became its director. He was interested in series expansions of special functions and also contributed to Chebyshev and least squares approximation. The “Gram determinant” was introduced by him in connection with his study of linear independence.

In particular, if  $f \in \Phi$ , then (6.1.8) holds with equality, since trivially  $\varphi(l; \cdot) = f(\cdot)$ . Thus, our approximation (6.1.8) already satisfies the exactness condition required for (6.1.2). It remains only to show that (6.1.8) produces indeed an approximation of the form (6.1.2).

To do so, observe that the interpolant in (6.1.8) is

$$\varphi(\ell; \cdot) = \sum_{j=1}^m c_j \varphi_j(\cdot)$$

where the vector  $c = [c_1, c_2, \dots, c_m]^T$  satisfies (6.1.7) with  $s = \ell$

$$Gc = \ell, \quad \ell = [L_1 f, L_2 f, \dots, L_m f]^T.$$

Writing

$$\lambda_j = L\varphi_j, \quad j = \overline{1, m}, \quad \lambda = [\lambda_1, \lambda_2, \dots, \lambda_m]^T, \quad (6.1.9)$$

we have by the linearity of  $L$

$$L\varphi(\ell; \cdot) = \sum_{j=1}^m c_j L\varphi_j = \lambda^T c = \lambda^T G^{-1} \ell = [(G^T)^{-1} \lambda]^T \ell,$$

that is,

$$L\varphi(\ell; \cdot) = \sum_{i=1}^m a_i L_i f, \quad a = [a_1, a_2, \dots, a_m]^T = (G^T)^{-1} \lambda. \quad (6.1.10)$$

### 6.1.2 Method of undetermined coefficients

Here we determined the coefficients  $a_i$  in (6.1.3) such that the equality holds  $\forall f \in \Phi$ , which, by the linearity of  $L$  and  $L_i$  is equivalent to equality for  $f = \varphi_1, f = \varphi_2, \dots, f = \varphi_m$ ; that is

$$\left( \sum_{j=1}^m a_j L_j \right) \varphi_i = L\varphi_i, \quad i = \overline{1, m},$$

or by (6.1.8)

$$\sum_{j=1}^m a_j L_j \varphi_i = \lambda_i, \quad i = \overline{1, m}.$$

Evidently, the matrix of this system is  $G^T$ , so that

$$a = [a_1, a_2, \dots, a_m]^T = (G^T)^{-1} \lambda,$$

in agreement with (6.1.10). Thus, the method of interpolation and the method of undetermined coefficients are mathematically equivalent — they produce exactly the same approximation.

It seems that, at least in the case of polynomial (i.e.  $\Phi = \mathbb{P}_d$ ), that the method of interpolation is more powerful than the method of undetermined coefficients, because it also yields an expression for the error term (if we carry along the remainder term of interpolation). But, the method of undetermined coefficients is allowed, using the condition of exactness to find the remainder term by the Peano Theorem.

## 6.2 Numerical Differentiation

For simplicity we consider only the first derivative; analogous techniques apply to higher order derivatives. We solve the problem by means of interpolation: instead to differentiate  $f \in C^{m+1}[a, b]$ , we shall differentiate its interpolation polynomial:

$$f(x) = (L_m f)(x) + (R_m f)(x). \quad (6.2.1)$$

We write the interpolation polynomial in Newton form

$$\begin{aligned} (L_m f)(x) &= (N_m f)(x) = f_0 + (x - x_0)f[x_0, x_1] + \cdots + \\ &\quad + (x - x_0) \dots (x - x_{m-1})f[x_0, x_1, \dots, x_m] \end{aligned} \quad (6.2.2)$$

and the remainder in the form

$$(R_m f)(x) = (x - x_0) \dots (x - x_m) \frac{f^{(m+1)}(\xi(x))}{(m+1)!}. \quad (6.2.3)$$

Differentiating (6.2.2) with respect to  $x$  and then putting  $x = x_0$  gives

$$\begin{aligned} (L_m f)(x_0) &= f[x_0, x_1] + (x_0 - x_1)f[x_0, x_1, x_2] + \cdots + \\ &\quad + (x_0 - x_1)(x_0 - x_2) \dots (x_0 - x_{m-1})f[x_0, x_1, \dots, x_m]. \end{aligned} \quad (6.2.4)$$

Assuming that  $f$  is has  $n+2$  continuous derivatives in an appropriate interval we get

$$(R_m f)'(x_0) = (x_0 - x_1) \dots (x_0 - x_m) \frac{f^{(m+1)}(\xi(x_0))}{(m+1)!}. \quad (6.2.5)$$

Therefore, differentiating (6.2.4), we find

$$f'(x_0) = (L_m f)'(x_0) + \underbrace{(R_m f)'(x_0)}_{e_m}. \quad (6.2.6)$$

If  $H = \max_i |x_0 - x_i|$ , the error has the form  $e_m = O(H^m)$ , when  $H \rightarrow 0$ .

We can thus get approximation formulae of arbitrarily high order, but those with large order are of limited practical use.

**Remark 6.2.1.** Numerical differentiation is a critical operation; for this reason it must be avoided as much as possible, since even good approximation lead to poor approximation of the derivative (see Figure 6.1). This also follows from Example 6.2.2.  $\diamond$

**Example 6.2.2.** Let the function

$$f(x) = g(x) + \frac{1}{n} \sin n^2(x - a), \quad g \in C^1[a, b].$$

We see that  $d(f, g) \rightarrow 0$  ( $n \rightarrow \infty$ ), but  $d(f', g') = n \not\rightarrow 0$ .  $\diamond$

The most important uses of differentiation formulae are made in the discretization of differential equations — ordinary or partial. In these applications, the spacing of the points is usually uniform, but unequally distribution points arise when partial differential operators are to be discretized near the boundary of the domain of interest.

We can also use another interpolation procedures such as: Taylor, Hermite, spline, least squares.

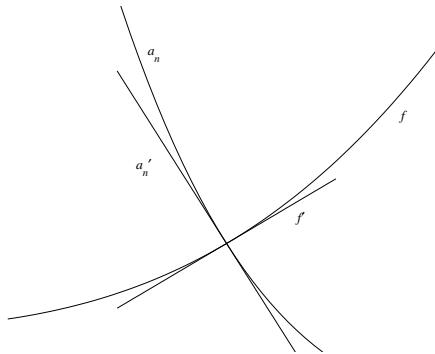


Figure 6.1: The drawbacks of numerical differentiation

### 6.3 Numerical Integration

The basic problem is to calculate the definite integral of a given function  $f$  over a finite interval  $[a, b]$ . If  $f$  is well behaved, this is a routine problem, for which the simplest integration rules, such as the composite trapezoidal or Simpson's rule will be quite adequate, the former having an edge over the latter if  $f$  is periodic with period  $b - a$ .

Complications arise if  $f$  has an integrable singularity, or the interval of integration extends to infinity (which is just other manifestation of the singular behavior). By breaking up the integral, if necessary, into several pieces, it can be assumed that the singularity, if its location is known, is at one (or both) ends of the interval  $[a, b]$ . Such improper integrals can usually be treated by weighted quadrature; that is one incorporates the singularity into a weight function, which then becomes one factor of the integrand, leaving the other factor well behaved. The most important example of this is Gaussian quadrature relative to such a weight function. Finally, it is possible to accelerate the convergence of quadrature schemes by suitable recombinations. The best-known example of this is Romberg integration.

Let  $f : [a, b] \rightarrow \mathbb{R}$  be a function integrable on  $[a, b]$ ,  $F_k(f)$ ,  $k = \overline{0, m}$  information on  $f$  (usually linear functionals) and  $w : [a, b] \rightarrow \mathbb{R}_+$  is a weight function, integrable over  $[a, b]$ .

**Definition 6.3.1.** *A formula of the form*

$$\int_a^b w(x)f(x)dx = Q(f) + R(f), \quad (6.3.1)$$

where

$$Q(f) = \sum_{j=0}^m A_j F_j(f),$$

is called a numerical integration formula for the function  $f$  or a quadrature formula. Parameters  $A_j$ ,  $j = \overline{0, m}$  are called weights or coefficients of the formula, and  $R(f)$  is its remainder term.  $Q$  is called quadrature functional.

**Definition 6.3.2.** *The natural number  $d = d(Q)$  having the property  $\forall f \in \mathbb{P}d$ ,  $R(f) = 0$  and  $\exists g \in \mathbb{P}d+1$  such that  $R(g) \neq 0$  is called degree of exactness of the quadrature formula..*

Since  $R$  is linear, a quadrature formula has the degree of exactness  $d$  if and only if  $R(e_j) = 0$ ,  $j = \overline{0, d}$  and  $R(e_{d+1}) \neq 0$ .

If the degree of exactness of a quadrature formula is known, the remainder could be determined using Peano theorem.

### 6.3.1 The composite trapezoidal and Simpson's rule

These formulae are called by Gautschi in [33] “the workhorses of numerical integration”. They will do the job when the interval is finite and the integrand is unproblematic. The trapezoidal rule is sometimes surprisingly effective on infinite intervals.

Both rules are obtained by applying the simplest kind of interpolation on subintervals of the decomposition

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b, \quad x_k = a + kh, \quad h = \frac{b-a}{n}. \quad (6.3.2)$$

of the interval  $[a, b]$ . In the trapezoidal rule, one interpolates linearly on each subinterval  $[x_k, x_{k+1}]$ , and obtains

$$\int_{x_k}^{x_{k+1}} f(x)dx = \int_{x_k}^{x_{k+1}} (L_1 f)(x)dx + \int_{x_k}^{x_{k+1}} (R_1 f)(x)dx, \quad f \in C^1[a, b], \quad (6.3.3)$$

where

$$(L_1 f)(x) = f_k + (x - x_k)f[x_k, x_{k+1}].$$

Integrating, we have

$$\int_{x_k}^{x_{k+1}} f(x)dx = \frac{h}{2}(f_k + f_{k+1}) + R_1(f),$$

where (using Peano Theorem)

$$R_1(f) = \int_{x_k}^{x_{k+1}} K_1(t)f''(t)dt,$$

and

$$\begin{aligned} K_1(t) &= \frac{(x_{k+1} - t)^2}{2} - \frac{h}{2}[(x_k - t)_+ + (x_{k+1} - t)_+] \\ &= \frac{(x_k - t)^2}{2} - \frac{h(x_{k+1} - t)}{2} \\ &= \frac{1}{2}(x_{k+1} - t)(x_k - t) \leq 0. \end{aligned}$$

So

$$R_1(f) = -\frac{h^3}{12}f''(\xi_k), \quad \xi_k \in (x_k, x_{k+1})$$

and

$$\int_{x_k}^{x_{k+1}} f(x)dx = \frac{h}{2}(f_k + f_{k+1}) - \frac{1}{12}h^3 f''(\xi_k). \quad (6.3.4)$$

This is the *elementary trapezoidal rule*. Summing over all subinterval gives the *trapezes rule* or the *composite trapezoidal rule*.

$$\int_a^b f(x)dx = h \left( \frac{1}{2}f_0 + f_1 + \cdots + f_{n-1} + \frac{1}{2}f_n \right) - \frac{1}{12}h^3 \sum_{k=0}^{n-1} f''(\xi_k).$$

Since  $f''$  is continuous on  $[a, b]$ , the remainder term could be written as

$$R_{1,n}(f) = -\frac{(b-a)h^2}{12}f''(\xi) = -\frac{(b-a)^3}{12n^2}f''(\xi). \quad (6.3.5)$$

Since  $f''$  is bounded in absolute value on  $[a, b]$  we have

$$R_{1,n}(f) = O(h^2),$$

when  $h \rightarrow 0$  and so the composite trapezoidal rule converges when  $h \rightarrow 0$  (or equivalently,  $n \rightarrow \infty$ ), provided that  $f \in C^2[a, b]$ .

MATLAB Source 6.1 gives an implementation of trapezes rule.

---

#### MATLAB Source 6.1 Composite trapezoidal rule

---

```
function I=trapezes(f,a,b,n);
%TRAPEZES trapezes formula
%call I=trapezes(f,a,b,n);

h=(b-a)/n;
I=(f(a)+f(b)+2*sum(f([1:n-1]*h+a)))*h/2;
```

---

If instead of linear interpolation one uses quadratic interpolation over two consecutive intervals, one gives rise to the *composite Simpson's formula*. Its "elementary" version, called *Simpson's rule* or *Simpson formula* is

$$\int_{x_k}^{x_{k+1}} f(x)dx = \frac{h}{3}(f_k + 4f_{k+1} + f_{k+2}) - \frac{1}{90}h^5 f^{(4)}(\xi_k), \quad x_k \leq \xi_k \leq x_{k+1}, \quad (6.3.6)$$

where it has been assumed that  $f \in C^4[a, b]$ .

Let us prove the formula for the remainder of Simpson rule. Since de degree of exactness is 3, Peano theorem yields to

$$R_2(f) = \int_{x_k}^{x_{k+2}} K_2(t)f^{(4)}(t) dt.$$

where

$$K_2(t) = \frac{1}{3!} \left\{ \frac{(x_{k+1}-t)^4}{4} - \frac{h}{3} [(x_k-t)_+^3 + 4(x_{k+1}-t)_+^3 + (x_{k+2}-t)_+^3] \right\},$$

that is,

$$K_2(t) = \frac{1}{6} \begin{cases} \frac{(x_{k+2}-t)^4}{4} - \frac{h}{3} [4(x_{k+1}-t)^3 + (x_{k+2}-t)^3], & t \in [x_k, x_{k+1}], \\ \frac{(x_{k+2}-t)^4}{4} - \frac{h}{3}(x_{k+2}-t)^3, & t \in [x_{k+1}, x_{k+2}]. \end{cases}$$

One easily checks that for  $t \in [a, b]$ ,  $K_2(t) \leq 0$ , so we can apply Peano's Theorem corollary.

$$R_2(f) = \frac{1}{4!} f^{(4)}(\xi_k) R_2(e_4),$$

$$\begin{aligned}
R_2(e_4) &= \frac{x_{k+2}^5 - x_k^5}{5} - \frac{h}{3} [x_k^4 + 4x_{k+1}^4 + x_{k+1}^4] \\
&= h \left[ 2 \frac{x_{k+2}^4 + x_{k+2}^3 x_k + x_{k+2}^2 x_k^2 + x_{k+2} x_k^3 + x_k^4}{5} \right. \\
&\quad \left. - \frac{5x_k^4 + 4x_k^3 x_{k+2} + 6x_k^2 x_{k+2}^2 + 4x_k x_{k+2}^3 + 5x_{k+2}^4}{12} \right] \\
&= \frac{h}{60} (-x_k^4 + 4x_k^3 x_{k+2} + 6x_k^2 x_{k+2}^2 + 4x_k x_{k+2}^3 - x_{k+2}^4) \\
&= -\frac{h}{60} (x_{k+2} - x_k)^4 = -4 \frac{h^5}{15}.
\end{aligned}$$

Thus,

$$R_2(f) = -\frac{h^5}{90} f^{(4)}(\xi_k).$$

For the composite Simpson <sup>2</sup>rule we get

$$\int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \dots + 4f_{n-1} + f_n) + R_{2,n}(f) \quad (6.3.7)$$

with

$$R_{2,n}(f) = -\frac{1}{180}(b-a)h^4 f^{(4)}(\xi) = -\frac{(b-a)^5}{2880n^4} f^{(4)}(\xi), \quad \xi \in (a, b). \quad (6.3.8)$$

One notes that  $R_{2,n}(f) = O(h^4)$ , which assures convergence when  $n \rightarrow \infty$ . We have also a gain with 1 in the order of accuracy. This is the reason why Simpson's rule has long been, and continues to be, one of the most popular general-purpose integration methods. For an implementation, see MATLAB Source 6.2.

---

### MATLAB Source 6.2 Composite Simpson formula

---

```

function I=Simpson(f,a,b,n);
%SIMPSON composite Simpson formula
%call I=Simpson(f,a,b,n);

h=(b-a)/n;
x2=[1:n-1]*h+a;
x4=[0:n-1]*h+a+h/2;
I=h/6*(f(a)+f(b)+2*sum(f(x2))+4*sum(f(x4)));

```

---

Thomas Simpson (1710-1761) was an English mathematician, self-educated, and author of many textbooks popular at the time. Simpson published his formula in 1743, but it was already known to Cavalieri (1639), Gregory (1668), and Cotes (1722), among others.



The composite trapezoidal rule works well for trigonometric polynomials. Suppose, without loss of generality that  $[a, b] = [0, 2\pi]$  and let

$$\begin{aligned}\mathbb{T}_m[0, 2\pi] = \{t(x) : t(x) = a_0 + a_1 \cos x + a_2 \cos 2x + \cdots + a_m \cos mx \\ + b_1 \sin x + b_2 \sin 2x + \cdots + b_m \sin mx\}.\end{aligned}$$

Then

$$R_{n,1}(f) = 0, \forall f \in \mathbb{T}_{n-1}[0, 2\pi]. \quad (6.3.9)$$

We can easily check this by taking  $f(x) = e_\nu(x) := e^{i\nu x} = \cos \nu x + i \sin \nu x, \nu = 0, 1, 2, \dots$ :

$$\begin{aligned}R_{n,1}(e_\nu) &= \int_0^{2\pi} e_\nu(x) dx - \frac{2\pi}{n} \left[ \frac{1}{2} e_\nu(0) + \sum_{k=1}^{n-1} e_\nu\left(\frac{2k\pi}{n}\right) + \frac{1}{2} e_\nu(2\pi) \right] \\ &= \int_0^{2\pi} e^{i\nu x} dx - \frac{2\pi}{n} \sum_{k=0}^{n-1} e^{\frac{2\pi ik\nu}{n}}.\end{aligned}$$

When  $\nu = 0$ , this is zero, and otherwise, since  $\int_0^{2\pi} e^{i\nu x} dx = (i\nu)^{-1} e^{i\nu x} \Big|_0^{2\pi} = 0$ ,

$$R_{n,1}(e_\nu) = \begin{cases} -2\pi & \text{if } \nu = 0 \pmod{n}, \nu > 0 \\ -\frac{2\pi}{n} \frac{1 - e^{i\nu n \cdot 2\pi/n}}{1 - e^{i\nu \cdot 2\pi/n}} = 0 & \text{if } \nu \neq 0 \pmod{n} \end{cases} \quad (6.3.10)$$

In particular,  $R_{n,1}(e_\nu) = 0$  for  $\nu = 0, 1, \dots, n-1$ , which proves (6.3.9). Taking the real and imaginary part in (6.3.10) one obtains

$$R_{n,1}(\cos \nu \cdot) = \begin{cases} -2\pi, & \nu = 0 \pmod{n}, \nu \neq 0 \\ 0, & \text{otherwise} \end{cases}, \quad R_{n,1}(\sin \nu \cdot) = 0.$$

Therefore, if  $f$  is  $2\pi$ -periodic and has a uniform convergent Fourier expansion

$$f(x) = \sum_{\nu=0}^{\infty} [a_\nu(f) \cos \nu x + b_\nu(f) \sin \nu x], \quad (6.3.11)$$

where  $a_\nu(f), b_\nu(f)$  are the Fourier coefficients of  $f$ , then

$$\begin{aligned}R_{n,1}(f) &= \sum_{\nu=0}^{\infty} [a_\nu(f) R_{n,1}(\cos \nu \cdot) + b_\nu(f) R_{n,1}(\sin \nu \cdot)] \\ &= -2\pi \sum_{l=1}^{\infty} a_{ln}(f)\end{aligned} \quad (6.3.12)$$

From the theory of Fourier series it is known that Fourier coefficients of  $f$  go to zero faster the smoother  $f$  is. More precisely, if  $f \in C^r[\mathbb{R}]$ , then  $a_\nu(f) = O(\nu^{-r})$  as  $\nu \rightarrow \infty$  (and similarly for  $b_\nu(f)$ ). Since by (6.3.12)

$$R_{n,1}(f) \simeq -2\pi a_n(f),$$

it follows that

$$R_{n,1}(f) = O(n^{-r}) \text{ as } n \rightarrow \infty \quad f \in C^r[\mathbb{R}], 2\pi\text{-periodic}, \quad (6.3.13)$$

which for  $r > 2$  is better than  $R_{n,1}(f) = O(n^{-2})$ , valid for nonperiodic functions  $f$ . In particular, if  $r = \infty$ , the trapezes rule converges faster than any power of  $n^{-1}$ . It should be noted, however, that  $f$

must be smooth on the whole real line  $\mathbb{R}$ . Starting with a function  $f \in C^r[0, 2\pi]$  and extending it by periodicity to  $\mathbb{R}$ , will *not* in general produce a function  $f \in C^r[\mathbb{R}]$ .

Another instance in which the trapezoid rule works very well is for functions  $f$  defined on  $\mathbb{R}$  which have the following property: for some  $r \geq 1$

$$\begin{aligned} f &\in C^{2r+1}[\mathbb{R}], \quad \int_{\mathbb{R}} |f^{(2r+1)}(x)| dx < \infty, \\ \lim_{x \rightarrow -\infty} f^{(2\rho-1)}(x) &= \lim_{x \rightarrow +\infty} f^{(2\rho-1)}(x) = 0, \quad \rho = 1, 2, \dots, r. \end{aligned} \quad (6.3.14)$$

In this case, one can show that

$$\int_{\mathbb{R}} f(x) dx = h \sum_{k=-\infty}^{\infty} f(kh) + R(f; h) \quad (6.3.15)$$

has an error  $R(f, h)$  satisfying  $R(f, h) = O(h^{2r+1})$ ,  $h \rightarrow 0$ . Hence, if (6.3.14) holds for each  $r \in \mathbb{N}$ , then, the error tends to zero faster than any power of  $h$ .

### 6.3.2 Weighted Newton-Cotes and Gauss formulae

A *weighted quadrature formula* is a formula of the type

$$\int_a^b f(t) w(t) dt = \sum_{k=1}^n w_k f(t_k) + R_n(f) \quad (6.3.16)$$

where the weight function  $w$  is nonnegative and integrable on  $(a, b)$ . The interval  $(a, b)$  may be finite or infinite. If it is infinite, we must make sure that the integral in (6.3.16) is well defined, at least when  $f$  is a polynomial. We achieve this by requiring that all moments of the weight function,

$$\mu_s = \int_a^b t^s w(t) dt, \quad s = 0, 1, 2, \dots \quad (6.3.17)$$

exist and are finite.

We call (6.3.16) *interpolatory*, if it has the degree of exactness  $d = n - 1$ . Interpolatory formulae are precisely those “obtained by interpolation”, that is, for which

$$\sum_{k=1}^n w_k f(t_k) = \int_a^b L_{n-1}(f; t_1, \dots, t_n, t) w(t) dt \quad (6.3.18)$$

or equivalently,

$$w_k = \int_a^b l_k(t) w(t) dt, \quad k = 1, 2, \dots, n, \quad (6.3.19)$$

where

$$l_k(t) = \prod_{\substack{l=1 \\ l \neq k}}^n \frac{t - t_l}{t_k - t_l} \quad (6.3.20)$$

are the elementary Lagrange interpolation polynomials associated with the nodes  $t_1, t_2, \dots, t_n$ . The fact that (6.3.16) with  $w_k$  given by (6.3.19) has the degree of exactness  $d = n - 1$  is evident, since

for any  $f \in \mathbb{P}_{n-1}$   $L_{n-1}(f; \cdot) \equiv f(\cdot)$  in (6.3.18). Conversely, if (6.3.16) has the degree of exactness  $d = n - 1$ , then putting  $f(t) = l_r(t)$  in (6.3.17) gives

$$\int_a^b l_r(t)w(t)dt = \sum_{k=1}^n w_k l_r(t_k) = w_r, \quad r = 1, 2, \dots, n,$$

that is, (6.3.19).

We see that given any  $n$  distinct nodes  $t_1, \dots, t_n$  it is always possible to construct a formula of type (6.3.16) which is exact for all polynomials of degree  $\leq n - 1$ . In the case  $w(t) \equiv 1$  on  $[-1, 1]$  and  $t_k$  equally spaced on  $[-1, 1]$ , the feasibility of such a construction was already alluded to by Newton in 1687 and implemented in detail by Cotes<sup>3</sup> around 1712. By extension, we call the formula (6.3.16), with the  $t_k$  prescribed and the  $w_k$  given by (6.3.19) a *Newton-Cotes formula*.

The question naturally arises whether we can do better, that is, whether we can achieve the degree of exactness  $d > n - 1$  by a judicious choice of the nodes  $t_k$  (the weights  $w_k$  being necessarily given by (6.3.19)). The answer is surprisingly simple and direct. To formulate it we introduce the node polynomial

$$u_n(t) = \prod_{k=1}^n (t - t_k). \quad (6.3.21)$$

**Theorem 6.3.3.** *Given an integer  $k$ , with  $0 \leq k \leq n$ , the quadrature formula (6.3.16) has the degree of exactness  $d = n - 1 + k$  if and only if both of the following conditions are satisfied.*

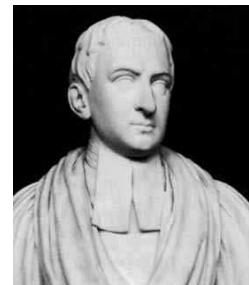
- (a) *The formula (6.3.16) is interpolatory;*
- (b) *The node polynomial  $u_n$  in (6.3.21) satisfies*

$$\int_a^b u_n(t)p(t)w(t)dt = 0, \quad \forall p \in \mathbb{P}_{k-1}.$$

The condition in (b) imposes  $k$  conditions on the nodes  $t_1, t_2, \dots, t_n$  of (6.3.16). (If  $k = 0$ , there is no restriction since, as we know, we can always get  $d = n - 1$ ). In effect,  $u_n$  must be orthogonal to  $\mathbb{P}_{k-1}$  relative to the weight function  $w$ . Since  $w(t) \geq 0$ , we have necessarily  $k \leq n$ ; otherwise,  $u_n$  would have to be orthogonal to  $\mathbb{P}_n$ , in particular, orthogonal to itself, which is impossible. Thus  $k = n$  is optimal, giving rise to a quadrature rule of maximum degree of exactness  $d_{\max} = 2n - 1$ . Condition (b) then amounts to orthogonality of  $u_n$  to all polynomials of lower degree; that is  $u_n(\cdot) = \pi_n(\cdot, w)$  is precisely the  $n$ th-degree orthogonal polynomial with respect to the weight function  $w$ . This optimal formula is called the *Gaussian quadrature formula* associated with the weight function  $w$ . Its nodes, therefore, are the roots of  $\pi_n(\cdot, w)$ , and the weights (coefficients)  $w_k$  are given as in (6.3.19); thus

$$\begin{aligned} \pi_n(t_k; w) &= 0 \\ w_k &= \int_a^b \frac{\pi_n(t, w)}{(t - t_k)\pi'_n(t_k, w)} w(t)dt, \quad k = 1, 2, \dots, n. \end{aligned} \quad (6.3.22)$$

Roger Cotes (1682-1716), precocious son of an English country pastor, was entrusted with the preparation of the second edition of Newton's *Principia*. He worked out in detail Newton's idea of numerical integration and published the coefficients — now known as Cotes numbers — of the  $n$ -point formula for all  $n < 11$ . Upon his death at the early age of 33, Newton said of him: "If he had lived, we might have known something."



The formula was developed in 1814 by Gauss for the special case  $w(t) \equiv 1$  on  $[-1, 1]$ , and extended to more general weight functions by Christoffel<sup>4</sup> in 1877. It is, therefore, also referred to as *Gauss-Christoffel quadrature formula*.

*Proof of theorem 6.3.3. Necessity.* Since the degree of exactness is  $d = n - 1 + k \geq n - 1$ , condition (a) is trivial. Condition (b) also follows immediately, since for any  $p \in \mathbb{P}_{k-1}$ ,  $u_n p \in \mathbb{P}_{n-1+k}$ . Hence,

$$\int_a^b u_n(t)p(t)w(t)dt = \sum_{k=1}^n w_k u_k(t_k)p(t_k),$$

which vanishes, since  $u_n(t_k) = 0$  for  $k = 1, 2, \dots, n$ .

*Sufficiency.* We must show that for any  $p \in \mathbb{P}_{n-1+k}$  we have  $R_n(p) = 0$  in (6.3.16). Given any such  $p$ , divide it by  $u_n$ , such that

$$p = qu_n + r, \quad q \in \mathbb{P}_{k-1}, \quad r \in \mathbb{P}_{n-1},$$

where  $q$  is the quotient and  $r$  the remainder. There follows

$$\int_a^b p(t)w(t)dt = \int_a^b q(t)u_n(t)w(t)dt + \int_a^b r(t)w(t)dt.$$

The first integral on the right, by (b), is zero, since  $q \in \mathbb{P}_{k-1}$ , whereas the second, by (a), since  $r \in \mathbb{P}_{n-1}$  equals

$$\sum_{k=1}^n w_k r(t_k) = \sum_{k=1}^n w_k [p(t_k) - q(t_k)u_n(t_k)] = \sum_{k=1}^n w_k p(t_k)$$

the last equality following again from  $u_n(t_k) = 0$ ,  $k = 1, 2, \dots, n$ , which completes the proof.  $\square$

The case  $k = n$  is discussed further in §6.3.3. Here we still mention two special cases with  $k < n$ , which are of some practical interest. The first is the *Gauss-Radau* quadrature formula in which one endpoint, say  $a$ , is finite and serves as a quadrature node, say  $t_1 = a$ . The maximum degree of exactness attainable then is  $d = 2n - 2$  and corresponds to  $k = n - 1$  in Theorem (6.3.3). Part (b) of that theorem tells us that the remaining nodes  $t_2, \dots, t_n$  must be the zeroes of  $\pi_{n-1}(\cdot, w_a)$ , where  $w_a(t) = (t - a)w(t)$ .

Similarly, in the *Gauss-Lobatto* formula, both endpoints are finite and serve as nodes, say,  $t_1 = a$ ,  $t_n = b$ , and the remaining nodes  $t_2, \dots, t_{n-1}$  are taken to be the zeros of  $\pi_{n-2}(\cdot; w_{a,b})$ ,  $w_{a,b}(t) = (t - a)(b - t)w(t)$ , thus achieving maximum degree of exactness  $d = 2n - 3$ .



4

Elvin Bruno Christoffel (1829-1900) was active for a short period of time in Berlin and Zurich and, for the rest of his life, in Strasbourg. He is best known for his work in geometry, in particular, tensor analysis, which became important in Einstein's theory of relativity.

### 6.3.3 Properties of Gaussian quadrature rules

The Gaussian quadrature rule (6.3.16) and (6.3.22), in addition to being optimal (i.e. has maximum degree of exactness), has some interesting and useful properties.

(i) All nodes  $t_k$  are real, distinct, and contained in the open interval  $(a, b)$ . This is a well-known property satisfied by the zeros of orthogonal polynomials.

(ii) All the weights (coefficients)  $w_k$  are positive. An ingenious observation of Stieltjes proves it almost immediately. Indeed,

$$0 < \int_a^b l_j^2(t) w(t) dt = \sum_{k=1}^n w_k l_j^2(t_k) = w_j, \quad j = 1, 2, \dots, n,$$

the first equality following since  $l_j^2 \in \mathbb{P}_{2n-2}$  and the degree of exactness is  $d = 2n - 1$ .

(iii) If  $[a, b]$  is a finite interval, then the Gauss formula converges for any continuous function; that is,  $R_n(f) \rightarrow 0$ , when  $n \rightarrow \infty$ , for any  $f \in C[a, b]$ . This is basically a consequence of the Weierstrass Approximation Theorem, which implies that, if  $\hat{p}_{2n-1}(f; \cdot)$  denotes the best polynomial approximation of degree  $2n - 1$  of  $f$  on  $[a, b]$  in the uniform norm, then

$$\lim_{n \rightarrow \infty} \|f(\cdot) - \hat{p}_{2n-1}(f; \cdot)\|_\infty = 0.$$

Since  $R_n(\hat{p}_{2n-1}) = 0$  (since  $d = 2n - 1$ ), it follows that

$$\begin{aligned} |R_n(f)| &= |R_n(f - \hat{p}_{2n-1})| \\ &= \left| \int_a^b [f(t) - \hat{p}_{2n-1}(f; t)] w(t) dt - \sum_{k=1}^n w_k [f(t_k) - \hat{p}_{2n-1}(f; t_k)] \right| \\ &\leq \int_a^b |f(t) - \hat{p}_{2n-1}(f; t)| w(t) dt + \sum_{k=1}^n w_k |f(t_k) - \hat{p}_{2n-1}(f; t_k)| \\ &\leq \|f(\cdot) - \hat{p}_{2n-1}(f; \cdot)\|_\infty \left[ \int_a^b w(t) dt + \sum_{k=1}^n w_k \right]. \end{aligned}$$

Here the positivity of weights  $w_k$  has been used crucially. Noting that

$$\sum_{k=1}^n w_k = \int_a^b w(t) dt = \mu_0,$$

we conclude

$$|R_n(f)| \leq 2\mu_0 \|f - \hat{p}_{2n-1}\|_\infty \rightarrow 0, \text{ c\^and } n \rightarrow \infty.$$

The next property is the background for an efficient algorithm for computing Gaussian quadrature formulae.

(iv) Let  $\alpha_k = \alpha_k(w)$  and  $\beta_k = \beta_k(w)$  be the recursion coefficients for the orthogonal polynomials  $\pi_k(\cdot) = \pi_k(\cdot; w)$ , that is

$$\begin{aligned} \pi_{k+1}(t) &= (t - \alpha_k) \pi_k(t) - \beta_k \pi_{k-1}(t), \quad k = 0, 1, 2, \dots \\ \pi_0(t) &= 1, \quad \pi_{-1}(t) = 0, \end{aligned} \tag{6.3.23}$$

where

$$\begin{aligned} \alpha_k &= \frac{(t\pi_k, \pi_k)}{(\pi_k, \pi_k)} \\ \beta_k &= \frac{(\pi_k, \pi_k)}{(\pi_{k-1}, \pi_{k-1})}, \end{aligned} \tag{6.3.24}$$

with  $\beta_0$  defined (as is customary) by

$$\beta_0 = \int_a^b w(t) dt (= \mu_0).$$

The  $n$ th order *Jacobi matrix* for the weight function  $w$  is a tridiagonal symmetric matrix defined by

$$J_n(w) = \begin{bmatrix} \alpha_0 & \sqrt{\beta_1} & & 0 \\ \sqrt{\beta_1} & \alpha_1 & \sqrt{\beta_2} & \\ & \sqrt{\beta_2} & \ddots & \\ & & \ddots & \ddots & \sqrt{\beta_{n-1}} \\ 0 & & & \sqrt{\beta_{n-1}} & \alpha_{n-1} \end{bmatrix}.$$

**Theorem 6.3.4.** *The nodes  $t_k$  of a Gauss-type quadrature formula are the eigenvalues of  $J_n$*

$$J_n v_k = t_k v_k, \quad v_k^T v_k = 1, \quad k = 1, 2, \dots, n, \quad (6.3.25)$$

and the weights  $w_k$  are expressible in terms of the first component  $v_k$  of the corresponding (normalized) eigenvectors by

$$w_k = \beta_0 v_{k,1}^2, \quad k = 1, 2, \dots, n \quad (6.3.26)$$

Thus, to compute the Gauss formula, we must solve an eigenvalue/eigenvector problem for a symmetric tridiagonal matrix. This is a routine problem in linear algebra, and very efficient methods (e.g. the QR algorithm) are known for solving it. Thus, the eigenvalue-based approach is more efficient than the classical one. Moreover, the classical approach is based on two ill-conditioned problems: the solution of a polynomial equation and the solution of a Vandermonde system of linear equations.

*Proof of theorem 6.3.4.* Let  $\tilde{\pi}_k(\cdot) = \tilde{\pi}_k(\cdot, w)$  denote the normalized orthogonal polynomials, so that  $\pi_k = \sqrt{(\pi_k, \pi_k)}_{d\lambda} \tilde{\pi}_k$ . Inserting this into (6.3.23), dividing by  $\sqrt{(\pi_k, \pi_k)}_{d\lambda}$ , and using (6.3.24), we obtain

$$\tilde{\pi}_{k+1}(t) = (t - \alpha_k) \frac{\tilde{\pi}_k}{\sqrt{\beta_{k+1}}} - \beta_k \frac{\tilde{\pi}_{k-1}}{\sqrt{\beta_{k+1}\beta_k}},$$

or, multiplying through by  $\sqrt{\beta_{k+1}}$  and rearranging

$$t\tilde{\pi}_k(t) = \alpha_k \tilde{\pi}_k(t) + \sqrt{\beta_k} \tilde{\pi}_{k-1}(t) + \sqrt{\beta_{k+1}} \tilde{\pi}_{k+1}(t), \quad k = 0, 1, \dots, n-1. \quad (6.3.27)$$

In terms of the Jacobi matrix  $J_n$  we can write these relations in vector form as

$$t\tilde{\pi}(t) = J_n \tilde{\pi}(t) + \sqrt{\beta_n} \tilde{\pi}_n(t) e_n, \quad (6.3.28)$$

where  $\tilde{\pi}(t) = [\tilde{\pi}_0(t), \tilde{\pi}_1(t), \dots, \tilde{\pi}_{n-1}(t)]^T$  and  $e_n(t) = [0, 0, \dots, 0, 1]^T$  are vectors in  $\mathbb{R}^n$ . Since  $t_k$  is a zero of  $\tilde{\pi}_n$ , it follows from (6.3.28) that

$$t_k \tilde{\pi}(t_k) = J_n \tilde{\pi}(t_k), \quad k = 1, 2, \dots, n. \quad (6.3.29)$$

This proves the first relation in Theorem 6.3.4, since  $\tilde{\pi}$  is a nonzero vector, its first component being

$$\tilde{\pi}_0 = \beta_0^{-1/2}. \quad (6.3.30)$$

To prove the second relation, note from (6.3.29) that the normalized eigenvector  $v_k$  is

$$v_k = \frac{1}{[\tilde{\pi}(t_k)^T \tilde{\pi}(t_k)]} \tilde{\pi}(t_k) = \left( \sum_{\mu=1}^n \tilde{\pi}_{\mu-1}^2(t_k) \right)^{-1/2} \tilde{\pi}(t_k).$$

Comparing the first component on far left and right, and squaring, gives, by virtue of (6.3.30)

$$\frac{1}{\sum_{\mu=1}^n \tilde{p}_{\mu-1}^2(t_k)} = \beta_0 v_{k,1}^2, \quad k = 1, 2, \dots, n. \quad (6.3.31)$$

On the other hand, letting  $f(t) = \tilde{\pi}_{\mu-1}(t)$  in (6.3.16), one gets, by orthogonality, using (6.3.30) again that

$$\beta_0^{1/2} \delta_{\mu-1,0} = \sum_{k=0}^n w_k \tilde{\pi}_{\mu-1}(t_k)$$

or in matrix form

$$Pw = \beta_0^{1/2} e_1, \quad (6.3.32)$$

where  $\delta_{\mu-1,0}$  is Kronecker's delta,  $P \in \mathbb{R}^{n \times n}$  is the matrix of eigenvectors,  $w \in \mathbb{R}^n$  is the vector of Gaussian weights, and  $e_1 = [1, 0, \dots, 0]^T \in \mathbb{R}^n$ . Since the columns of  $P$  are orthogonal, we have

$$P^T P = D, \quad D = \text{diag}(d_1, d_2, \dots, d_n), \quad d_k = \sum_{\mu=1}^n \tilde{\pi}_{\mu-1}^2(t_k).$$

Now multiply (6.3.32) from the left by  $P^T$  to obtain

$$Dw = \beta_0^{1/2} P^T e_1 = \beta_0^{1/2} * \beta_0^{-1/2} e = e, \quad e = [1, 1, \dots, 1]^T.$$

Therefore,  $w = D^{-1}e$ , that is,

$$w_k = \frac{1}{\sum_{\mu=1}^n \tilde{\pi}_{\mu-1}^2(t_k)}, \quad k = 1, 2, \dots, n.$$

Comparing this with (6.3.31) establishes the desired result.  $\square$

For details on algorithmic aspects concerning orthogonal polynomials and Gaussian quadratures see [34].

MATLAB Source 6.3 computes the nodes and the coefficients of a Gaussian quadrature formula by mean of eigenvalues and eigenvectors of Jacobi matrix. The input parameters are the coefficients  $\alpha$  and  $\beta$  in the recurrence relation. It uses the MATLAB function `eig`.

---

### MATLAB Source 6.3 Compute nodes and coefficients of a Gaussian quadrature rule

---

```
function [g_nodes,g_coeff]=Gaussquad(alpha,beta)
%GAUSSQUAD - generate Gaussian quadrature formula
%computes nodes and coefficients for
%Gauss rules given alpha and beta
%method - Jacobi matrix
n=length(alpha); rb=sqrt(beta(2:n));
J=diag(alpha)+diag(rb,-1)+diag(rb,1);
[v,d]=eig(J);
g_nodes=diag(d);
g_coeff=beta(1)*v(1,:).^2;
```

---

**MATLAB Source 6.4** Approximation of an integral using a Gaussian formula

---

```
function I=vquad(g_nodes,g_coeff,f)
I=g_coeff*f(g_nodes);
```

---

For the computation of the approximative value of an integral via a Gaussian rule, with nodes and coefficients computed by Gaussquad we need a single line of code – see MATLAB Source 6.4. We give in the sequel MATLAB functions which compute the coefficients and the nodes of various type Gaussian rules. They call the function Gaussquad. MATLAB Source 6.5 computes Gauss-Legendre nodes and coefficients.

**MATLAB Source 6.5** Generate a Gauss-Legendre formula

---

```
function [g_nodes,g_coeff]=Gauss_Legendre(n)
%GAUSS-LEGENDRE - Gauss-Legendre nodes and coefficients

beta=[2,(4-([1:n-1]).^(-2)).^(-1)];
alpha=zeros(n,1);
[g_nodes,g_coeff]=Gaussquad(alpha,beta);
```

---

Since a first kind Gauss-Chebyshev rule has equal coefficients, and the nodes are the roots of first kind Chebyshev polynomial, MATLAB Source 6.6 does not use the Jacobi matrix. We also give sources for second kind Gauss-Chebyshev rule, (MATLAB Source 6.7), Gauss-Hermite rule (MATLAB Source 6.8), Gauss-Laguerre rule (MATLAB Source 6.9) and Gauss-Jacobi rule (MATLAB Source 6.10).

**MATLAB Source 6.6** Generate a first kind Gauss-Chebyshev formula

---

```
function [g_nodes,g_coeff]=Gauss_Cheb1(n)
%GAUSS_CHEB1 - Gauss-Cebisev #1 nodes and coefficients

g_coeff=pi/n*ones(1,n);
g_nodes=cos(pi*([1:n]'-0.5)/n);
```

---

**MATLAB Source 6.7** Generate a second kind Gauss-Chebyshev formula

---

```
function [g_nodes,g_coeff]=Gauss_Cheb2(n)
%GAUSS_CHEB2 - Gauss-Chebyshev #2 nodes and coefficients

beta=[pi/2,1/4*ones(1,n-1)]; alpha=zeros(n,1);
[g_nodes,g_coeff]=Gaussquad(alpha,beta);
```

---

**MATLAB Source 6.8 Generate a Gauss-Hermite formula**

---

```
function [g_nodes,g_coeff]=Gauss_Hermite(n)
%GAUSS_HERMITE - Gauss-Hermite nodes and coefficients

beta=[sqrt(pi),[1:n-1]/2]; alpha=zeros(n,1);
[g_nodes,g_coeff]=Gaussquad(alpha,beta);
```

---

**MATLAB Source 6.9 Generate a Gauss-Laguerre formula**

---

```
function [g_nodes,g_coeff]=Gauss_Laguerre(n,a)
%GAUSS_HERMITE - Gauss-Laguerre nodes and coefficients

k=1:n-1;
alpha=[a+1, 2*k+a+1];
beta=[gamma(1+a),k.*(k+a)];
[g_nodes,g_coeff]=Gaussquad(alpha,beta);
```

---

**MATLAB Source 6.10 Generate a Gauss-Jacobi formula**

---

```
function [g_nodes,g_coeff]=Gauss_Jacobi(n,a,b)
%Gauss-Jacobi - Gauss-Jacobi nodes and coefficients

k=0:n-1;
k2=2:n-1;
%rec. relation coeffs
bet1=4*(1+a)*(1+b)/((2+a+b)^2)/(3+a+b);
bet=[2^(a+b+1)*beta(a+1,b+1), bet1, 4*k2.* (k2+a+b).* (k2+a).* ...
(k2+b)./(2*k2+a+b-1)./(2*k2+a+b).^2./ (2*k2+a+b+1)];
if a==b
    alpha=zeros(1,n);
else
    alpha=(b^2-a^2)./(2*k+a+b)./(2*k+a+b+2);
end
[g_nodes,g_coeff]=Gaussquad(alpha,bet);
```

---

(v) Markov<sup>5</sup> observed in 1885 that the Gauss quadrature formula can also be obtained by Hermite interpolation on the nodes  $t_k$ , double.

$$f(x) = (H_{2n-1}f)(x) + u_n^2(x)f[x, x_1, x_1, \dots, x_n, x_n],$$

$$\begin{aligned} \int_a^b w(x)f(x)dx &= \int_a^b w(x)(H_{2n-1}f)(x)dx + \\ &\quad + \int_a^b w(x)u_n^2(x)f[x, x_1, x_1, \dots, x_n, x_n]dx. \end{aligned}$$

But the degree of exactness  $2n - 1$  implies

$$\begin{aligned} \int_a^b w(x)(H_{2n-1}f)(x)dx &= \sum_{i=1}^n w_i(H_{2n-1}f)(x_i) = \sum_{i=1}^n w_i f(x_i), \\ \int_a^b w(x)f(x)dx &= \sum_{i=1}^n w_i f(x_i) + \int_a^b w(x)u_n^2(x)f[x, x_1, x_1, \dots, x_n, x_n]dx, \end{aligned}$$

so

$$R_n(f) = \int_a^b w(x)u_n^2(x)f[x, x_1, x_1, \dots, x_n, x_n]dx.$$

Since  $w(x)u_n^2(x) \geq 0$ , applying the second Mean Value Theorem for integrals and the Mean Value Theorem for divided differences, we get

$$\begin{aligned} R_n(f) &= f[\eta, x_1, x_1, \dots, x_n, x_n] \int_a^b w(x)u_n^2(x)dx \\ &= \frac{f^{(2n)}(\xi)}{(2n)!} \int_a^b w(x)[\pi_n(x, w)]^2 dx, \quad \xi \in [a, b]. \end{aligned}$$

For orthogonal polynomials and their recursion coefficients  $\alpha_k, \beta_k$  see Table 5.2, page 175.

## 6.4 Adaptive Quadratures

In a numerical integration method errors depend not only on the size of the interval, but also on values of certain higher order derivatives of the function to be integrated. This implies that the methods do not work well for functions having large values of higher order derivatives — especially for functions

---

Andrei Andrejevich Markov (1856-1922) was a Russian mathematician active in St. Petersburg who made important contributions to probability theory, number theory, and constructive approximation theory.



having large oscillations on the whole interval or on some subintervals. It is reasonable to use small subintervals where the derivatives have large values and large subintervals where the derivatives have small values. A method which does this systematically is called adaptive quadrature.

The general approach in an adaptive quadrature is to use two different methods for each interval, to compare the results, and to divide the interval when the differences are large. There are situations when we have two bad methods, the results are bad, but their differences are small. We can avoid this situation taking a method which overestimates the result and another which underestimates it. We shall give an example of general structure for a recursive adaptive quadrature (MATLAB Source 6.11). The

---

### MATLAB Source 6.11 Adaptive quadrature

---

```
function I=adaptquad(f,a,b,eps,g)
%ADAPTQUAD adaptive quadrature
%call I=ADAPTQUAD(F,A,B,EPS,G)
%F - integrand
%A,B - endpoints
%EPS -tolerance
%G - composed rule used on subintervals

m=4;
I1=g(f,a,b,m);
I2=g(f,a,b,2*m);
if abs(I1-I2) < eps %success
    I=I2;
    return
else %recursive subdivision
    I=adaptquad(f,a,(a+b)/2,eps,g)+adaptquad(f,(a+b)/2,b,eps,g);
end
```

---

parameter  $g$  is a function that implements a composite quadrature rule, such as trapezes or composite Simpson rule. Algorithm structure: DIVIDE AND CONQUER.

In contrast to other methods, which decide what amount of work is needed to achieve the desired accuracy, an adaptive quadrature compute only as much as it is necessary. This means that the absolute error  $\varepsilon$  must be chosen so that to avoid an infinite loop when one tries to achieve a precision which could not be achieved. The number of steps depends on the behavior of the function to be integrated.

Possible improvement: the accuracy can be scaled by the ratio of current interval length and the whole interval length.

**Example 6.4.1.** Approximate the length of a sinusoid on an interval of length equal to a period.

We have to approximate

$$I = \int_0^{2\pi} \sqrt{1 + \cos^2(x)} dx.$$

The integrand in MATLAB is

```
function y=lsin(x)
y=sqrt(1+cos(x).^2);
```

Here are two examples of using adaptquad

```

>> format long
>> I=adaptquad(@lsin,0,2*pi,1e-8,@Simpson)
I =
    7.64039557805542
>> I=adaptquad(@lsin,0,2*pi,1e-8,@trapez)
I =
    7.64039557011458

```

We recommend the reader to compare the running times.  $\diamond$

For supplementary details on design and implementation of adaptive quadrature formulas see [31].

## 6.5 Iterated Quadratures. Romberg Method

A drawback of previous adaptive quadrature is that it computes repeatedly the function values at nodes; also a time penalty appears at running time due to recursion or to stack management in an iterative implementation. Iterative quadratures remove these drawbacks. They apply at the first step a composite quadrature rule and then they divide the interval into equal parts using at each step the previously computed approximations. We shall illustrate this technique using a method that starts from composite trapezoidal rule and then improves the convergence using Richardson extrapolation.

The first step involves applying the composite trapezoidal rule for  $n_1 = 1, n_2 = 2, \dots, n_p = 2^{p-1}$ , where  $p \in \mathbb{N}^*$ . The step size  $h_k$  corresponding to  $n_k$  would be

$$h_k = \frac{b-a}{n_k} = \frac{b-a}{2^{k-1}}.$$

Using these notations the trapezes rule becomes

$$\int_a^b f(x)dx = \frac{h_k}{2} \left[ f(a) + f(b) + 2 \sum_{i=1}^{2^{n-1}-1} f(a + ih_k) \right] - \frac{b-a}{12} h_k^2 f''(\mu_k) \quad (6.5.1)$$

$\mu_k \in (a, b)$ .

Let  $R_{k,1}$  denote the result of approximation in accordance to (6.5.1).

$$R_{1,1} = \frac{h_1}{2} [f(a) + f(b)] = \frac{b-a}{2} [f(a) + f(b)] \quad (6.5.2)$$

$$\begin{aligned} R_{2,1} &= \frac{h_2}{2} [f(a) + f(b) + 2f(a + h_2)] = \\ &= \frac{b-a}{4} \left[ f(a) + f(b) + 2f\left(a + \frac{b-a}{2}\right) \right] \\ &= \frac{1}{2} \left[ R_{1,1} + h_1 f\left(a + \frac{1}{2}h_1\right) \right]. \end{aligned}$$

and generally

$$R_{k,1} = \frac{1}{2} \left[ R_{k-1,1} + h_{k-1} \sum_{i=1}^{2^{k-2}} f\left(a + \left(i - \frac{1}{2}\right) h_{k-1}\right) \right], \quad k = \overline{2, n} \quad (6.5.3)$$

Now, it follows the improvement by Richardson <sup>6</sup> extrapolation (in fact it is due to Archimedes <sup>7</sup> ).

$$I = \int_a^b f(x)dx = R_{k-1,1} - \frac{(b-a)}{12} h_k^2 f''(a) + O(h_k^4).$$

We shall eliminate the  $h_k^2$  term by combining two equations

$$\begin{aligned} I &= R_{k-1,1} - \frac{(b-a)}{12} h_k^2 f''(a) + O(h_k^4), \\ I &= R_{k,1} - \frac{b-a}{48} h_k^2 f''(a) + O(h_k^4). \end{aligned}$$

We get

$$I = \frac{4R_{k,1} - R_{k-1,1}}{3} + O(h^4).$$

We define

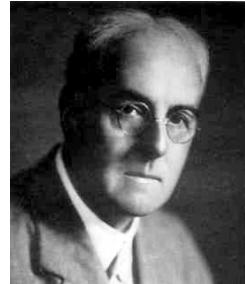
$$R_{k,2} = \frac{4R_{k,1} - R_{k-1,1}}{3}. \quad (6.5.4)$$

One applies the Richardson extrapolation to these values. If  $f \in C^{2n+2}[a, b]$ , then for  $k = \overline{1, n}$  we may write

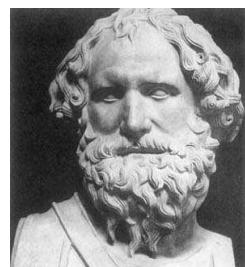
$$\begin{aligned} \int_a^b f(x)dx &= \frac{h_k}{2} \left[ f(a) + f(b) + 2 \sum_{i=1}^{2^{k-1}-1} f(a + ih_k) \right] \\ &\quad + \sum_{i=1}^k K_i h_k^{2i} + O(h_k^{2k+2}), \end{aligned} \quad (6.5.5)$$

where  $K_i$  does not depend on  $h_k$ .

Lewis Fry Richardson (1881-1953), born, educated, and active in England, did pioneering work in numerical weather prediction, proposing to solve the hydrodynamical and thermodynamical equations of meteorology by finite difference methods. He also did a <sup>6</sup>penetrating study of atmospheric turbulence, where a nondimensional quantity introduced by him is now called "Richardson's number". At the age of 50 he earned a degree in psychology and began to develop a scientific theory of international relations. He was elected fellow of the Royal Society in 1926.



Archimedes (287 B.C. - 212 B.C.) Greek mathematician from Syracuse. The achievements of Archimedes are quite outstanding. He is considered by most historians of mathematics as one of the greatest mathematicians of all time. He perfected a method of integration which allowed him to find areas, volumes and surface areas of many bodies. Archimedes was able to apply the method of exhaustion, <sup>7</sup>which is the early form of integration. He also gave an accurate approximation to  $\pi$  and showed that he could approximate square roots accurately. He invented a system for expressing large numbers. In mechanics Archimedes discovered fundamental theorems concerning the centre of gravity of plane figures and solids. His most famous theorem gives the weight of a body immersed in a liquid, called Archimedes' principle. He defended his town during the Romans' siege.



The formula (6.5.5) can be argued as follows. Let  $a_0 = \int_a^b f(x)dx$  and

$$A(h) = \frac{h}{2} \left[ f(a) + 2 \sum_{k=1}^{n-1} f(a + kh) + f(b) \right], \quad h = \frac{b-a}{k}.$$

(the approximation obtained using trapezes rule).

If  $f \in C^{2k+1}[a, b]$ ,  $k \in \mathbb{N}^*$ , then the following formula holds

$$A(h) = a_0 + a_1 h^2 + a_2 h^4 + \cdots + a_k h^{2k} + O(h^{2k+1}), \quad h \rightarrow 0 \quad (6.5.6)$$

where

$$a_k = \frac{B_{2k}}{(2k)!} [f^{(2k-1)}(b) - f^{(2k-1)}(a)], \quad k = 1, 2, \dots, K.$$

The quantities  $B_k$  are the coefficients in the expansion

$$\frac{z}{e^z - 1} = \sum_{k=0}^{\infty} \frac{B_k}{k!} z^k, \quad |z| < 2\pi;$$

they are called *Bernoulli*<sup>8</sup> numbers.

By eliminating successively the powers of  $h$  in (6.5.5) one obtains

$$R_{k,j} = \frac{4^{j-1} R_{k,j-1} - R_{k-1,j-1}}{4^{j-1} - 1}, \quad k = \overline{2, n}, \quad j = \overline{2, i}.$$

The computation could be performed in a tabular fashion:

$$\begin{array}{ccccccc} R_{1,1} & & & & & & \\ R_{2,1} & R_{2,2} & & & & & \\ R_{3,1} & R_{3,2} & R_{3,3} & & & & \\ \vdots & \vdots & \vdots & \ddots & & & \\ R_{n,1} & R_{n,2} & R_{n,3} & \dots & R_{n,n} & & \end{array}$$

Since  $(R_{n,1})$  is convergent,  $(R_{n,n})$  is also convergent, but faster than  $(R_{n,1})$ . One may choose as stopping criterion  $|R_{n-1,n-1} - R_{n,n}| \leq \varepsilon$ .

We give an implementation of Romberg's method (see MATLAB Source 6.12, the M-file Romberg.m).

**Example 6.5.1.** One can solve the problem in Example 6.4.1 as follows:

---

Jacob Bernoulli (1654-1705), the elder brother of Johann Bernoulli, was active in Basel. He was one of the first to appreciate Leibniz's and Newton's differential and integral calculus and enriched it by many original contributions of his own, often in (not always amicable) competition with his younger brother. He is also known in probability theory for his "law of large numbers".



---

**MATLAB Source 6.12 Romberg method**

---

```
function I=Romberg(f,a,b,epsi,nmax)
%ROMBERG - approximate an integral using Romberg method
%call I=romberg(f,a,b,epsi,nmax)
%f -integrand
%a,b - integration limits
%epsi - tolerance
%nmax - maximum number of iterations
if nargin < 5
    nmax=10;
end
if nargin < 4
    epsi=1e-3;
end
R=zeros(nmax, nmax);
h=b-a;
% first iteration
R(1,1)=h/2*(sum(f([a,b])));
for k=2:nmax
    %trapezes formula
    x=a+([1:2^(k-2)]-0.5)*h;
    R(k,1)=0.5*(R(k-1,1)+h*sum(f(x)));
    %extrapolation
    plj=4;
    for j=2:k
        R(k, j)=(plj*R(k, j-1)-R(k-1, j-1))/(plj-1);
        plj=plj*4;
    end
    if (abs(R(k,k)-R(k-1,k-1))<epsi) & (k>3)
        I=R(k,k);
        return
    end
    %halving step
    h=h/2;
end
error('iteration number exceeded')
```

---

```
>> I=Romberg(@lsin,0,2*pi,1e-8)
```

```
I =
7.64039557805609
```

Formula (6.5.6) is called *Euler*<sup>9</sup> -*MacLaurin*<sup>10</sup> formula.

## 6.6 Adaptive Quadratures II

The second column of tabular Romberg's method corresponds to a Simpson's rule approximation. We introduce the notation

$$S_{k,1} = R_{k,2}.$$

The third column is a combination of two Simpson approximation:

$$S_{k,2} = S_{k,1} + \frac{S_{k,1} - S_{k-1,1}}{15} = R_{k,2} + \frac{R_{k,2} - R_{k-1,2}}{15}.$$

We shall use the relation

$$S_{k,2} = S_{k,1} + \frac{S_{k,1} - S_{k-1,1}}{15}, \quad (6.6.1)$$

to devise and adaptive quadrature.

Let  $c = (a + b)/2$ . The elementary Simpson formula is

$$S = \frac{h}{6} (f(a) + 4f(c) + f(b)).$$

---

Leonhard Euler (1707-1783) was the son of a minister interested in mathematics who followed lectures of Jakob Bernoulli at the University of Basel. Euler himself was allowed to see Johann Bernoulli on Saturday afternoons for private tutoring. At the age of 20, after he was unsuccessful to obtain a professorship in physics at the University of Basel, because of a lottery system then in use (Euler lost), he emigrated to St. Petersburg; later, he moved on to Berlin, and then back to St. Petersburg. Euler unquestionable was the most prolific mathematician of the 18th century, working in virtually all branches of the differential and integral calculus and, in particular, being one of the founders of the calculus of variations. He also did pioneering work in the applied sciences, notably hydrodynamics, mechanics of deformable materials and rigid bodies, optics, astronomy and the theory of the spinning top. Not even his blindness at the age of 59 managed to break his phenomenal productivity. Euler's collected works are still being edited, 71 volumes having already been published.



Colin Maclaurin (1698-1768) was a Scottish mathematician who applied the new infinitesimal calculus to the various problems in geometry. He is best known for his power series expansion, but also contributed to the theory of equations.



For two subintervals one obtains

$$S_2 = \frac{h}{12} (f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)),$$

where  $d = (a + c)/2$  and  $e = (c + b)/2$ . Applying (6.6.1) to  $S_1$  and  $S_2$  yields

$$Q = S_2 + (S_2 - S)/15.$$

Now, we are able to give a recursive algorithm for the approximation of our integral. The function `adquad` evaluates the integral by applying Simpson. It calls `quadstep` recursively and apply extrapolation. The implementation appears in MATLAB Source 6.13.

**Example 6.6.1.** Problem in Example 6.4.1 can be solved using `adquad`:

```
>> I=adquad(@lsin,0,2*pi,1e-8)
```

```
I =
7.64039557801944
```

## 6.7 Numerical Integration in MATLAB

MATLAB has two main functions for numerical integration, `quad` and `quadl`. Both require  $a$  and  $b$  to be finite and the integrand to have no singularities on  $[a, b]$ . For infinite integrals and integrals with singularities a variety of approaches can be used in order to produce an integral that can be handled by `quad` and `quadl`; these include change of variable, integration by parts, and analytic treatment of the integral over part of the range. See numerical analysis textbooks for details, for example [88, 33, 101, 22, 83].

The most frequent usage is `q = quad(fun, a, b, tol)` (and similarly for `quadl`), where `fun` specifies the function to be integrated. It can be given as a string, an inline object or a function handle. The argument `tol` is an absolute error tolerance, which defaults to `1e-6`. A recommended value is a small multiple of `eps` times an estimate of the integral.

The form `q = quad(fun, a, b, tol, trace)` with a nonzero `trace` shows the values of `[fcnt a b-a Q]` calculated during the recursion.

`[q, fcount] = quad(...)` returns the number of function evaluations.

Suppose we want to approximate  $\int_0^\pi x \sin x dx$ . We can store the integrand in an M-file, say `xsin.m`:

```
function y=xsin(x)
y=x.*sin(x);
```

The approximate value is computed by:

```
>> quad(@xsin,0,pi)
ans =
3.1416
```

`quad` function is an implementation of a Simpson-type quadrature, as described in Section 6.6 or in [66]. `quadl` is more precise and is based on a 4 points Gauss-Lobatto formula (with degree of exactness 5) and a 7 points Kronrod extension (with degree of exactness 9), described in [31]. Both routines use adaptive quadrature. They break the range of integration into subintervals and apply the basic

---

**MATLAB Source 6.13 Adaptive quadrature, variant**

```
function [Q,fcount] = adquad(F,a,b,tol,varargin)
%ADQUAD adaptive quadrature
%call [Q,fcount] = adquad(F,a,b,tol,varargin)
% F - integrand
% a,b - interval endpoints
% tol - tolerance, default 1.e-6.
% other arguments are passed to the integrand, F(x,p1,p2,...).

% make F callable by feval.
if ischar(F) & exist(F)^~2
    F = inline(F);
elseif isa(F,'sym')
    F = inline(char(F));
end
if nargin < 4 | isempty(tol), tol = 1.e-6; end

% Initialization
c = (a + b)/2;
fa = feval(F,a,varargin{:}); fc = feval(F,c,varargin{:});
fb = feval(F,b,varargin{:});

% Recursive call
[Q,k] = quadstep(F, a, b, tol, fa, fc, fb, varargin{:});
fcount = k + 3;

% -----
function [Q,fcount] = quadstep(F,a,b,tol,fa,fc,fb,varargin)
% Recursive subfunction called by adquad

h = b - a;
c = (a + b)/2;
fd = feval(F, (a+c)/2,varargin{:});
fe = feval(F, (c+b)/2,varargin{:});
Q1 = h/6 * (fa + 4*fc + fb);
Q2 = h/12 * (fa + 4*fd + 2*fc + 4*fe + fb);
if abs(Q2 - Q1) <= tol
    Q = Q2 + (Q2 - Q1)/15;
    fcount = 2;
else
    [Qa,ka] = quadstep(F, a, c, tol, fa, fd, fc, varargin{:});
    [Qb,kb] = quadstep(F, c, b, tol, fc, fe, fb, varargin{:});
    Q = Qa + Qb;
    fcount = ka + kb + 2;
end
```

---

integration rule over each subinterval. They choose the subintervals according to the local behavior of the integrand, placing the smallest ones where the integrand is changing most rapidly. Warning messages are produced if the subintervals become very small or if an excessive number of function evaluations is used, either of which could indicate that the integrand has a singularity.

To illustrate how `quad` and `quadl` we shall approximate

$$\int_0^1 \left( \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.09)^2 + 0.04} - 6 \right) dx.$$

The integrand is the MATLAB function `humps`, used to test numerical integration functions or in MATLAB demos. We shall apply `quad` to this function with `tol=1e-4`. Figure 6.2 plots the integrand and shows with tick marks on the x-axis where the integrand was evaluated; circles mark the corresponding values of the integrand. The figure shows that the subintervals are smallest where the integrand is most rapidly varying. We obtained it by modifying `quad` MATLAB function.

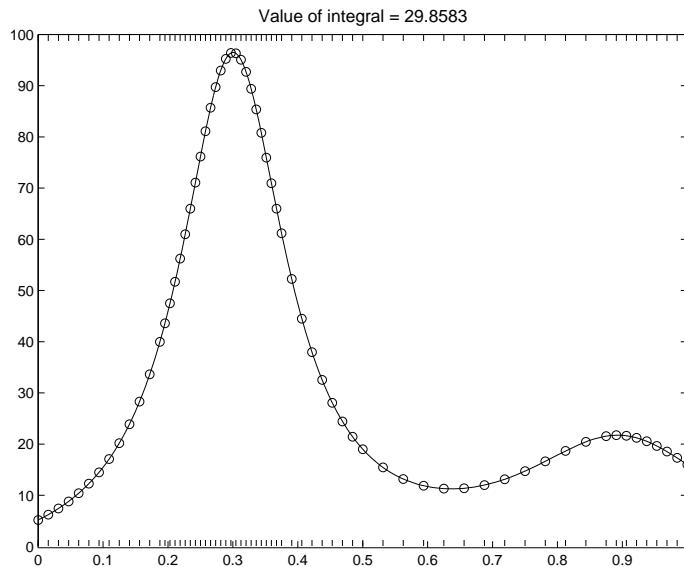


Figure 6.2: Numerical integration of `humps` function by `quad`

The next example approximates the Fresnel integrals

$$x(t) = \int_0^t \cos(u^2) du, \quad y(t) = \int_0^t \sin(u^2) du.$$

This are the parametric equations of a curve, called the Fresnel spiral. Figure 6.3 shows its graphical representation, produced by sampling 1000 equally spaced points on the interval  $[-4\pi, 4\pi]$ . For efficiency reasons, we exploit the symmetry and avoid repeatedly integrating on  $[0, t]$  by integrating over each subinterval and then evaluating the cumulative sums using `cumsum`. Here is the code:

```
n = 1000; x = zeros(1,n); y = x;
i1 = inline('cos(x.^2)'); i2 = inline('sin(x.^2)');
t=linspace(0,4*pi,n);
```

```

for i=1:n-1
    x(i) = quadl(i1,t(i),t(i+1),1e-3);
    y(i) = quadl(i2,t(i),t(i+1),1e-3);
end
x = cumsum(x); y = cumsum(y);
plot([-x(end:-1:1),0,x], [-y(end:-1:1),0,y])
axis equal

```

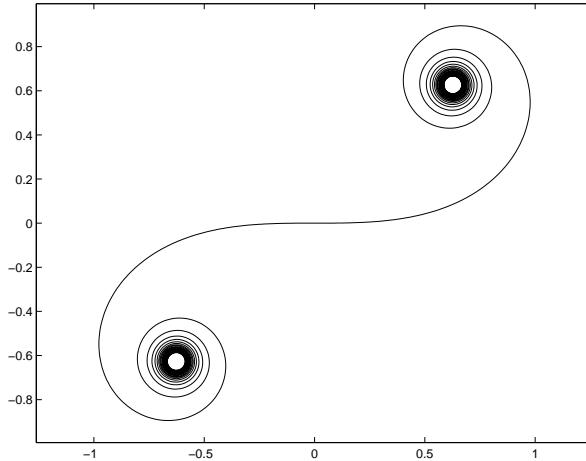


Figure 6.3: Fresnel spiral

To integrate functions given by values, not by their analytical expression, one uses `trapz` function. It implements the composite trapezoidal rule. The nodes need not to be equally spaced. As we have already seen in Section 6.3.1, it works well when integrates periodical function on intervals whose lengths is an integer multiple of the period. Example:

```

>> x=linspace(0,2*pi,10);
>> y=1./(2+sin(x));
>> trapz(x,y)
ans =
    3.62759872810065
>> 2*pi*sqrt(3)/3-ans
ans =
    3.677835813675756e-010

```

The exact value of the integral is  $\frac{2}{3}\sqrt{3}\pi$ , so the error is less than  $10^{-9}$ .

The function `quadv` accepts a vector argument and returns a vector.

A newer function is `quadgk`. It implements adaptive quadrature based on a Gauss-Kronrod pair (15th and 7th order formulas). Besides the approximate value of the integral, it can return an error bound and it accepts several options which control the integration process (for example we can specify the singularities). If the integrand is complex-valued or the limits are complex, it integrates on a straight line within the complex plane. We give two examples. The first computes

$$\int_{-1}^1 \frac{\sin x}{x} dx.$$

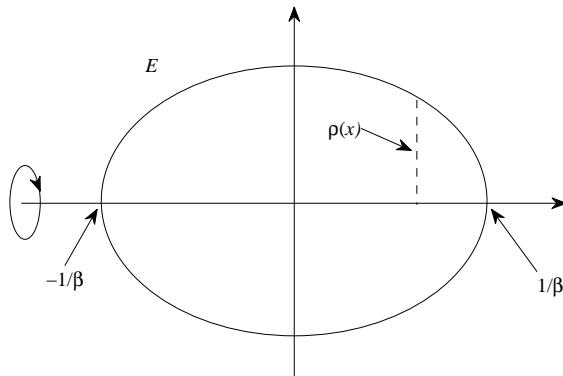


Figure 6.4: Section of the ellipsoid

```
>> format long
>> ff=@(x) sin(x)./x;
>> quadgk(ff,-1,1,'RelTol',1e-8,'AbsTol',1e-12)
ans =
    1.892166140734366
```

Notice that quad and quadl fail; they return NaN. Nevertheless, they succeed if we compute the integral as:

$$\int_{-1}^1 \frac{\sin x}{x} dx = \int_{-1}^0 \frac{\sin x}{x} dx + \int_0^1 \frac{\sin x}{x} dx.$$

The second example uses `Waypoints` to integrate around a pole using a piecewise linear contour:

```
>> Q = quadgk(@(z) 1./(2*z - 1), -1-i, -1+i, 'Waypoints', [1-i, 1+i, -1+i])
Q =
    0.0000 + 3.1416i
```

See doc `quadgk` and [82] for further information.

## 6.8 Applications

We present here two applications adapted after [72].

### 6.8.1 Computation of an ellipsoid surface

Consider an ellipsoid obtained by rotating the ellipse in Figure 6.4 around the  $x$  axis.

The radius  $\rho$  is described as a function of axial coordinate by the equation

$$\rho^2(x) = \alpha^2(1 - \beta^2 x^2), \quad -\frac{1}{\beta} \leq x \leq \frac{1}{\beta},$$

where  $\alpha$  and  $\beta$  are such that  $\alpha^2\beta^2 < 1$ .

For test we chose the following values for the parameters:  $\alpha = (\sqrt{2} - 1)/10$ ,  $\beta = 10$ . The surface is given by

$$I(f) = 4\pi\alpha \int_0^{1/\beta} \sqrt{1 - K^2x^2} dx,$$

where  $K^2 = \beta^2\sqrt{1 - \alpha^2\beta^2}$ . Since  $f'(1/\beta) = -100$ , an adaptive quadrature seems to be appropriate.

We can compute the exact value and its floating point approximation using Symbolic Math Toolbox:

```
clear
syms alpha beta K2 s2 x f vI
s2=sqrt(sym(2));
alpha=(s2-1)/10;
beta=sym(10);
K2=beta^2*sqrt(1-alpha^2*beta^2);
f=sqrt(1-K2*x^2);
vI=4*sym(pi)*alpha*int(f,0,1/beta)
vpa(vI,16)
```

The results are:

```
vI =
1/100 pi (-2 (-(-2 + 2 2 ) ) + 1) + 2 (-(-2 + 2 2 ) )
           1/2 1/2      1/2      1/2 1/2
           + 1) 2 + (-2 + 2 2 ) asin((-2 + 2 2 ) ))
ans =
0.04234752094082434
```

The next script approximates the surface with a tolerance of  $1e-8$  using the following functions: Romberg, adquad, and MATLAB quad and quadl.

```
err=1e-8;
beta=10;
alpha=(sqrt(2)-1)/10;
alpha2=alpha^2;
beta2=beta^2;
K2=beta2*sqrt(1-alpha2*beta2);
f=@(x) sqrt(1-K2*x.^2);
fpa=4*pi*alpha;
[vi(1),nfe(1)]=Romberg(f,0,1/beta,err,100);
[vi(2),nfe(2)]=adquad(f,0,1/beta,err);
[vi(3),nfe(3)]=quad(f,0,1/beta,err);
[vi(4),nfe(4)]=quadl(f,0,1/beta,err);
vi=fpa*vi;
meth={'Romberg','adquad','quad','quadl'};
for i=1:4
    fprintf('%8s %18.16f %3d\n',meth{i},vi(i),nfe(i))
end
```

Here is the output:

```

Romberg 0.0423475209214685 129
adquad 0.0423475209189811   65
quad   0.0423475203088494   37
quadl  0.0423475209279265   48

```

Romberg method is inferior to adaptive quadratures. Surprisingly, quad beats quadl.

### 6.8.2 Computation of the wind action on a sailboat mast

The sailboat schematically drawn in Figure 6.5(a) is subject to the action of wind force.

The straight line AB represent the mast, of length  $L$ , and BO is one of the two shrouds (strings for the side stiffening of the mast).

Any infinitesimal element of the sail transmits to the corresponding element of length  $dx$  of the mast a force of magnitude equal to  $f(x)dx$ . The change of  $f$  along with the height  $x$ , measured from the point A (basis of the mast), is expressed by the following law

$$f(x) = \frac{\alpha x}{x + \beta} e^{-\gamma x},$$

where  $\alpha$ ,  $\beta$  and  $\gamma$  are given constants.

The resultant  $R$  of the force  $f$  is defined as

$$R = I(f) \equiv \int_0^L f(x)dx$$

and is applied at distance equal to  $b$  (to be determined) from the basis of the mast. The formula for  $b$  is  $b = I(xf)/I(f)$ .

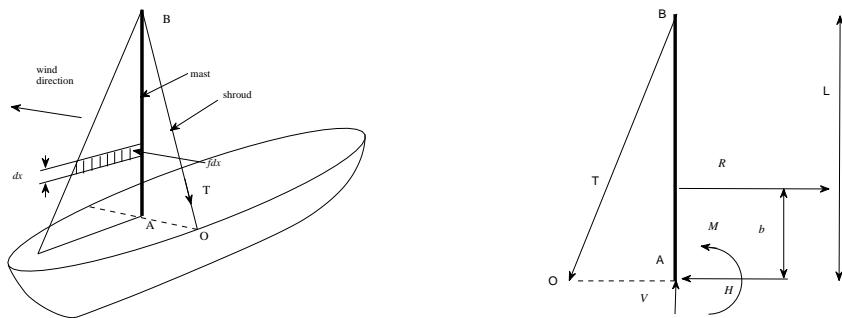


Figure 6.5: Schematic representation of a sailboat (left); forces acting on the mast (right)

Computing  $R$  and  $b$  is crucial for the structural design of the mast and shroud section. Once the values of  $R$  and  $b$  are known, it is possible to analyze the hyperstatic structure mast-shroud. This analysis has as results the reactions  $V$ ,  $H$ , and  $M$  at the basis of the mast and the traction  $T$  that is transmitted by the shroud (see Figure 6.5(b)). In a next step, the internal actions in the structure can be found, as well as the maximum stresses arising in the mast AB and in the shroud BO, from which, assuming that the safety verifications are satisfied, one can finally design the geometric parameters of the sections AB and BO.

We approximate  $R$  and  $b$  by adquad and MATLAB quad and quadl functions. The function sailboat in MATLAB Source 6.14 computes the approximations and plot number of function evaluations versus minus decimal logarithm of error. We test the function for  $\alpha = 50$ ,  $\beta = 5/3$ ,  $\gamma = 1/4$ ,

---

**MATLAB Source 6.14** Computation of the wind action on a sailboat mast

---

```
function sailboat(alpha, beta, gamma, L)
%SAILBOAT - computation of wind action on
%           a sailboat mast
% Alfio Quarteroni, Riccardo Sacco, Fausto Saleri
% Numerical Mathematics
% Springer 2000

f = @(x) alpha*x./(x+beta).*exp(-gamma*x);
xf = @(x) x.*f(x);
x=1:9;
err=10.^(-x);
for k=1:9
    [R1,ne1(k)]=adquad(f,0,L,err(k));
    [b1,neb1(k)]=adquad(xf,0,L,err(k)); b1=b1/R1;
    [R2,ne2(k)]=quad(f,0,L,err(k));
    [b2,neb2(k)]=quad(xf,0,L,err(k)); b2=b2/R2;
    [R3,ne3(k)]=quadl(f,0,L,err(k));
    [b3,neb3(k)]=quadl(xf,0,L,err(k)); b3=b3/R3;
end
subplot(1,2,1)
plot(x,ne1,'b-x',x,ne2,'r-+',x,ne3,'g--d')
xlabel('-log_{10}(err)', 'FontSize',14); ylabel('n', 'FontSize',14)
legend('adquad','quad','quadl',0)
title('Computation of \it{R}', 'FontSize',14)
subplot(1,2,2)
plot(x,neb1,'b-x',x,neb2,'r-+',x,neb3,'g--d')
xlabel('-log_{10}(err)', 'FontSize',14); ylabel('n', 'FontSize',14)
legend('adquad','quad','quadl',0)
title('Computation of \it{b}', 'FontSize',14)
R1,R2,R3
b1,b2,b3
```

---

and  $L = 10$ . Required tolerances are in the range  $10^{-1}$  to  $10^{-9}$ . The calling command and the results are given below:

```
>> sailboat(50, 5/3, 1/4, 10)
R1 =
          100.061368317961
R2 =
          100.061368317941
R3 =
          100.061368317962
b1 =
```

```

4.03145652950332
b2 =
4.03145652950425
b3 =
4.03145652950326

```

See Figure 6.6 for the number of functions evaluation required to attain the desired tolerances.

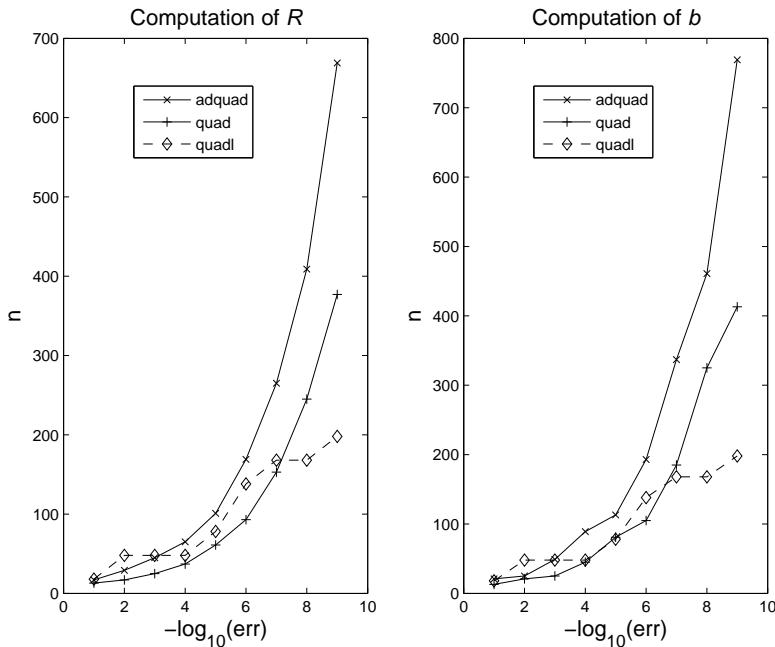


Figure 6.6: Number of function evaluations vs. tolerance for  $R$  (left) and  $b$

## Problems

**Problem 6.1.** Approximate

$$\int_0^1 \frac{\sin x}{x} dx,$$

using an adaptive quadrature and Romberg method. What kind of problems could occur? Compare your result to that provided by `quad` or `quadl`.

**Problem 6.2.** Starting from a convenient integral, approximate  $\pi$  with 8 exact decimal digits, using Romberg method and an adaptive quadrature.

**Problem 6.3.** Approximate

$$\int_{-1}^1 \frac{2}{1+x^2} dx$$

trapezoid formula and composed Simpson rule, for various values of  $n$ . How does the accuracy vary with  $n$ ? Give a graphical representation.

**Problem 6.4.** The error function, erf, is defined by

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Tabulate its values for  $x = 0.1, 0.2, \dots, 1$ , using `adquad` function. Compare the results with those provided by MATLAB functions `quad` and `erf`.

**Problem 6.5.** (a) Use `adquad` and MATLAB function `quad` to approximate

$$\int_{-1}^2 \frac{1}{\sin \sqrt{|t|}} dt.$$

(b) Why problems, like “divide by zero” do not occur at  $t = 0$ ?

**Problem 6.6.** For a number  $p \in \mathbb{N}$ , consider the integral

$$I_p = \int_0^1 (1-t)^p f(t) dt.$$

Compare the composite trapezoidal rule for  $n$  subintervals to Gauss-Jacobi formula with  $n$  nodes and the parameters  $\alpha = p$  and  $\beta = 0$ . Take, for example,  $f(t) = tgt$ ,  $p = 5(5)20$  and  $n = 10(10)50$  for the composite trapezoidal rule and  $n = 1(1)5$  Gauss-Jacobi formula.

**Problem 6.7.** Let

$$f(x) = \ln(1+x) \ln(1-x).$$

- (a) Use `ezplot` to plot  $f(x)$  for  $x \in [-1, 1]$ .
- (b) Use Maple or Symbolic Math toolbox to obtain the exact value of the integral

$$\int_{-1}^1 f(x) dx.$$

- (c) Find the numerical value of the expression in (b).
- (d) What problem occurs if we try to approximate the integral by

$$\text{adquad}'\log(1+x). * \log(1-x)', -1, 1)?$$

- (e) How can you avoid the difficulty? Argue your solution.
- (f) Use `adquad` with various accuracies (tolerances). Plot the error and the number of function evaluations as functions of tolerance.

**Problem 6.8.** A sphere of radius  $R$  floats half submerged in a liquid. If it is pushed down until the diametral plane is a distance  $p$  ( $0 < p \leq R$ ) below the surface and is released, the period of the resulting vibration is

$$T = 8R \sqrt{\frac{R}{g(6R^2 - p^2)}} \int_0^{2\pi} \frac{dt}{\sqrt{1 - k^2 \sin^2 t}},$$

where  $k^2 = p^2 / (6R^2 - p^2)$  and  $g = 10m/s^2$ . For  $R = 1$  and  $p = 0.5, 0.75, 1.0$  find  $T$ .



# CHAPTER 7

---

## Numerical Solution of Nonlinear Equations

---

### 7.1 Nonlinear Equations

The problems discussed in this chapter may be written generically in the form

$$f(x) = 0, \quad (7.1.1)$$

but allow different interpretations depending on the meaning of  $x$  and  $f$ . The simplest case is a *single equation* in a single unknown, in which case  $f$  is a given function of a real or complex variable, and we are trying to find values of this variables for which  $f$  vanishes. Such values are called *roots of the equation* (7.1.1) or *zeros of the function*  $f$ . If  $x$  in (7.1.1) is a vector, say  $x = [x_1, x_2, \dots, x_d]^T \in \mathbb{R}^d$ , and  $f$  is also a vector, each component of which is a function of  $d$  variables  $x_1, x_2, \dots, x_d$ , then (7.1.1) represents a *system of equations*. The system is *nonlinear* if at least one component of  $f$  depends nonlinearly of at least one of the variables  $x_1, x_2, \dots, x_d$ . If all components of  $f$  are linear functions of  $x_1, \dots, x_d$  we call (7.1.1) a *system of linear algebraic equations*. Still more generally, (7.1.1) could represent a *functional equation*, if  $x$  is an element of some function space and  $f$  a (linear or nonlinear) operator acting on this space. In each of these interpretations, the zero on the right of (7.1.1) has a different meaning: the number zero in the first case, the zero vector in the second, and the function identically equal to zero in the last case.

Much of this chapter is devoted to single nonlinear equations. Such equations are often encountered in the analysis of vibrating systems, where the roots correspond to critical frequencies (resonance). The special case of *algebraic equations*, where  $f$  in (7.1.1) is a polynomial, is also of considerable importance and deserves a special treatment.

### 7.2 Iterations, Convergence, and Efficiency

Even the simplest of nonlinear equations — for example, algebraic equations — are known to not admit solutions that are expressible rationally in terms of the data. It is therefore impossible, in general, to

compute roots of nonlinear equations in a finite numbers of arithmetic operations. What is required is an iterative method, that is, a procedure that generates an infinite sequence of approximations  $\{x_n\}_{n \in \mathbb{N}}$ , such that

$$\lim_{n \rightarrow \infty} x_n = \alpha, \quad (7.2.1)$$

for some root  $\alpha$  of the equation. In case of a system of equations, both  $x_k$  and  $\alpha$  are vectors of appropriate dimension, and convergence is to be understood in sense of a componentwise convergence.

Although convergence of an iterative process is certainly desirable, it takes more than just convergence to make it practical. What one wants is fast convergence. A basic concept to measure the speed of convergence is the *order of convergence*.

**Definition 7.2.1.** One says that  $x_n$  converge to  $\alpha$  (at least) linearly if

$$|x_n - \alpha| \leq e_n \quad (7.2.2)$$

where  $\{e_n\}$  is a positive sequence satisfying

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n} = c, \quad 0 < c < 1. \quad (7.2.3)$$

If (7.2.2) and (7.2.3) hold with equality in (7.2.2) then  $c$  is called asymptotic error constant.

The phrase “at least” in this definition relates to the fact that we have only inequality in (7.2.2), which in practice is all we can usually ascertain. So, strictly speaking, it is the bounds  $e_n$  that converge linearly, meaning that (e.g. for  $n$  large enough) each of these error bounds is approximately a constant fraction of the preceding one.

**Definition 7.2.2.** One says that  $x_n$  converges to  $\alpha$  with (at least) order  $p \geq 1$  if (7.2.2) holds with

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^p} = c, \quad c > 0 \quad (7.2.4)$$

Thus, convergence of order 1 is the same as linear convergence, whereas convergence of order  $p > 1$  is faster. Note that in this latter case there is no restriction on the constant  $c$ : once  $e_n$  is small enough, it will be the exponent  $p$  that takes care of the convergence. If we have equality in (7.2.2),  $c$  is again referred to as the asymptotic error constant.

The same definitions apply also to vector-valued sequences; one only needs to replace absolute values in (7.2.2) by (any) vector norm.

The classification of convergence with respect to order is still rather crude, as there are types of convergence that “fall between the cracks”. Thus, a sequence  $\{e_n\}$  may converge to zero slower than linearly, for example such that  $c = 1$  in (7.2.3). We call this type of convergence *sublinear*. Likewise,  $c = 0$  in (7.2.3) gives rise to *superlinear* convergence, if (7.2.4) does not hold for any  $p > 1$ .

It is instructive to examine the behavior of  $e_n$ , if instead of the limit relations (7.2.3) and (7.2.4) we had strict equality from some  $n$ , say,

$$\frac{e_{n+1}}{e_n^p} = c, \quad n = n_0, n_0 + 1, n_0 + 2, \dots \quad (7.2.5)$$

For  $n_0$  large enough, this is almost true. A simple induction argument then shows that

$$e_{n_0+k} = c^{\frac{p^k - 1}{p-1}} e_{n_0}^{p^k}, \quad (7.2.6)$$

which certainly holds for  $p > 1$ , but also for  $p = 1$  in the limits as  $p \downarrow 1$ :

$$e_{n_0+k} = c^k e_{n_0}, \quad k = 0, 1, 2, \dots, \quad (p = 1) \quad (7.2.7)$$

Assuming then  $e_{n_0}$  sufficiently small so that the approximation  $x_{n_0}$  has several correct decimal digits, we write  $e_{n_0+k} = 10^{-\delta_k} e_{n_0}$ . Then  $\delta_k$ , according to (7.2.2), approximately represents the number of additional correct digits in the approximation  $x_{n_0+k}$  (as opposed to  $x_{n_0}$ ). Taking logarithms in (7.2.6) and (7.2.7) gives

$$\delta_k = \begin{cases} k \log \frac{1}{c}, & \text{if } p = 1 \\ p^k \left[ \frac{1-p^{-k}}{p-1} \log \frac{1}{c} + (1-p^{-k}) \log \frac{1}{e_{n_0}} \right], & \text{if } p > 1 \end{cases}$$

hence as  $k \rightarrow \infty$

$$\delta_k \sim c_1 k \quad (p = 1), \quad \delta_k \sim c_p p^k \quad (p > 1), \quad (7.2.8)$$

where  $c_1 = \log \frac{1}{c} > 0$ , if  $p = 1$  and

$$c_p = \frac{1}{p-1} \log \frac{1}{c} + \log \frac{1}{e_{n_0}}.$$

(We assume here that  $n_0$  is large enough, and hence  $e_{n_0}$  small enough, to have  $c_p > 0$ ). This shows that the number of correct decimal digits increases linearly with  $k$  when  $p = 1$ , but exponentially when  $p > 1$ . In the latter case,  $\delta_{k+1}/\delta_k \sim p$  meaning that ultimately (for large  $k$ ) the number of correct decimal digits increases, per iteration step, by a factor of  $p$ .

If each iteration requires  $m$  units of work (a “unit of works” typically is the work involved in computing a function value or a value of one of its derivatives), then the *efficiency index* of the iteration may be defined by

$$\lim_{k \rightarrow \infty} [\delta_{k+1}/\delta_k]^{1/m} = p^{1/m}.$$

It provides a common basis on which to compare different iterative methods with one another. Methods that converge linearly have efficiency index 1.

Practical computation requires the employment of a *stopping rule* that terminates the iteration once the desired accuracy is (or is believed to be) obtained. Ideally, one stops as soon as  $\|x_n - \alpha\| < tol$ , where  $tol$  is a prescribed accuracy. Since  $\alpha$  is not known, one commonly replaces  $x_n - \alpha$  by  $x_n - x_{n-1}$  and requires

$$\|x_n - x_{n-1}\| \leq tol \quad (7.2.9)$$

where

$$tol = \|x_n\| \varepsilon_r + \varepsilon_a \quad (7.2.10)$$

with  $\varepsilon_r, \varepsilon_a$  prescribed tolerances. As a safety measure, one might require (7.2.9) not just for one, but a few consecutive values of  $n$ . Choosing  $\varepsilon_r = 0$  or  $\varepsilon_a = 0$  will make (7.2.10) a relative (resp., absolute) error tolerance. It is prudent, however, to use a “mixed error tolerance”, say  $\varepsilon_e = \varepsilon_a = \varepsilon$ . Then, if  $\|x_n\|$  is small or moderately large, one effectively controls the absolute error, whereas for  $\|x_n\|$  very large, it is in effect the relative error that is controlled. One can combine the above tests with  $\|f(x)\| \leq \varepsilon$ .

## 7.3 Sturm Sequences Method

There are situations in which it is desirable to be able to select one particular root among many others and have the iterative scheme converge to it. This is the case, for example, in orthogonal polynomials, where we know that all zeros are real and distinct. It may well be that we are interested in the second-largest or third-largest zero, and should be able to compute it without computing any of the others. This

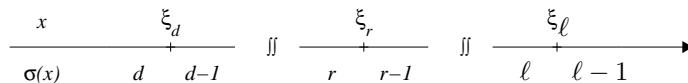


Figure 7.1: Sturm's theorem

is indeed possible if we combine the bisection method with the theorem of Sturm <sup>1</sup>.

Thus, consider

$$f(x) := \pi_d(x) = 0, \quad (7.3.1)$$

where  $\pi_d$  is a polynomial of degree  $d$ , orthogonal with respect to some positive measure. We know that  $\pi_d$  is the characteristic polynomial of a symmetric tridiagonal matrix and can be computed recursively by a three term recurrence relation

$$\begin{aligned} \pi_0(x) &= 1, & \pi_1(x) &= x - \alpha_0 \\ \pi_{k+1}(x) &= (x - \alpha_k)\pi_k(x) - \beta_k\pi_{k-1}(x), & k &= 1, 2, \dots, d-1 \end{aligned} \quad (7.3.2)$$

with all  $\beta_k$  positive. The recursion (7.3.2) is not only useful to compute  $\pi_d(x)$ , but has also the property due to Sturm.

**Theorem 7.3.1 (Sturm).** *Let  $\sigma(x)$  be the number of sign changes (zeros do not count) in the sequence of numbers*

$$\pi_d(x), \pi_{d-1}(x), \dots, \pi_1(x), \pi_0(x). \quad (7.3.3)$$

*Then, for any two numbers  $a, b$  with  $a < b$ , the number of real zeros of  $\pi_d$  in the interval  $a < x \leq b$  is equal to  $\sigma(a) - \sigma(b)$ .*

Since  $\pi_k(x) = x^k + \dots$ , it is clear that  $\sigma(-\infty) = d$ ,  $\sigma(+\infty) = 0$ , so that indeed the number of real zeros of  $\pi_d$  is  $\sigma(-\infty) - \sigma(\infty) = d$ . Moreover, if  $\xi_1 > \xi_2 > \dots > \xi_d$  denote the zeros of  $\pi_d$  in decreasing order, we have the behavior of  $\sigma(x)$  as shown in Figure 7.1.

It is now easy to see that

$$\sigma(x) \leq r - 1 \iff x \geq \xi_r. \quad (7.3.4)$$

Indeed, suppose that  $x \geq \xi_r$ . Then  $\{\# \text{zeros} \leq x\} \geq d + 1 - r$ ; hence, by Sturm's theorem,  $\sigma(-\infty) - \sigma(x) = d - \sigma(x) = \{\# \text{zeros} \leq x\} \leq d_1 - r$ , that is,  $\sigma(x) \leq r - 1$ . Conversely, if  $\sigma(x) \leq r - 1$ , then, again by Sturm's theorem,  $\{\# \text{zeros} \leq x\} = d - \sigma(x) \geq d + 1 - r$ , which implies  $x \geq \xi_r$  (see Figure 7.1).

The basic idea is to control the bisection process, not as in the bisection procedure, by checking the sign of  $\pi_d(x)$ , but rather, by checking the inequality (7.3.4) to see whether we are on the right or left

---

Jaques Charles François Sturm (1803-1855), a Swiss analyst and theoretical physicist, is best known for his theorem on Sturm sequences, <sup>1</sup> discovered in 1829, and his theory on Sturm-Liouville differential equation. He also contributed significantly to differential and projective geometry.



side of the zero  $\xi_r$ . In order to initialize the procedure, we need two values  $a_1 = a, b_1 = b$  such that  $a < \xi_d$  and  $b > \xi_1$ . These are trivially obtained as endpoints of the interval of orthogonality for  $\pi_d$ , if it is finite. More generally, one can apply Gershgorin's theorem to the Jacobi matrix  $J_d$  associated to the polynomial (7.3.2)

$$J_n = \begin{bmatrix} \alpha_0 & \sqrt{\beta_1} & & & 0 \\ \sqrt{\beta_1} & \alpha_1 & \sqrt{\beta_2} & & \\ & \sqrt{\beta_2} & \alpha_2 & \ddots & \\ & & \ddots & \ddots & \sqrt{\beta_{n-1}} \\ 0 & & & \sqrt{\beta_{n-1}} & \alpha_{n-1} \end{bmatrix}$$

and taking into account that the zeros of  $\pi_d$  are precisely the eigenvalues of  $J_d$ .

Gershgorin's theorem states that the eigenvalue of a matrix  $A = [a_{ij}]$  of order  $d$  are located in the union of the disks

$$\left\{ z \in \mathbb{C} : |z - a_{ii}| \leq r_i, r_i = \sum_{j \neq i} |a_{ij}| \right\}, \quad i = \overline{1, d}.$$

In this way,  $a$  can be chosen to be the smallest and  $b$  the largest of the  $d$  numbers  $\alpha_0 + \sqrt{\beta_1}, \alpha_1 + \sqrt{\beta_1} + \sqrt{\beta_2}, \dots, \alpha_{d-2} + \sqrt{\beta_{d-2}} + \sqrt{\beta_{d-1}}, \alpha_{d-1} + \sqrt{\beta_{d-1}}$ . The method of Sturm sequences then proceeds as follows, for any given  $r$  with  $1 \leq r \leq d$ :

```

for  $n := 1, 2, 3, \dots$  do
   $x_n := \frac{1}{2}(a_n + b_n);$ 
  if  $\sigma(x_n) > r - 1$  then
     $a_{n+1} := x_n; b_{n+1} := b_n;$ 
  else
     $a_{n+1} := a_n; b_{n+1} = x_n;$ 
  end if
end for
```

Since initially  $\sigma(a) = d > r - 1, \sigma(b) = 0 \leq r - 1$ , it follows by construction that

$$\sigma(a_n) > r - 1, \quad \sigma(b_n) \leq r - 1, \quad n = 1, 2, 3, \dots$$

meaning that  $\xi_r \in [a_n, b_n]$ , for all  $n = 1, 2, 3, \dots$ . Moreover, as in the bisection method,  $b_n - a_n = 2^{-(n-1)}(b - a)$ , so that  $|x_n - \xi_r| \leq \varepsilon_n$  with  $\varepsilon_n = 2^{-n}(b - a)$ . The method converges (at least) linearly to the root  $\xi_r$ . A computer implementation can be obtained by modifying the **if-else** statement appropriately.

## 7.4 Method of False Position

As in the method of bisection, we assume two numbers  $a < b$  such that

$$f \in C[a, b], \quad f(a)f(b) < 0 \tag{7.4.1}$$

and generate a sequence of nested intervals  $[a_n, b_n], n = 1, 2, 3, \dots$  with  $a_1 = a, b_1 = b$  such that  $f(a_n)f(b_n) < 0$ . Unlike the bisection method, however, we are not taking the midpoint of  $[a_n, b_n]$  to determine the next interval, but rather the solution  $x = x_n$  of the linear equation

$$(L_1 f)(x; a_n, b_n) = 0.$$

This would appear to be more flexible than bisection, as  $x_n$  will come to lie closer to the endpoint at which  $|f|$  is smaller.

More explicitly, the method proceeds as follows: define  $a_1 = a, b_1 = b$ . Then

```

for  $n := 1, 2, \dots$  do
   $x_n := a_n - \frac{a_n - b_n}{f(a_n) - f(b_n)} f(a_n);$ 
  if  $f(a_n)f(x_n) > 0$  then
     $a_{n+1} := x_n; b_{n+1} := b_n;$ 
  else
     $a_{n+1} := a_n; b_{n+1} := x_n;$ 
  end if
end for
```

One may terminate the iteration as soon as  $\min(x_n - a_n, b_n - x_n) \leq tol$ , where  $tol$  is a prescribed error tolerance, although this is not entirely fool-proof.

For an implementation see MATLAB Source 7.1.

---

### MATLAB Source 7.1 False position method for nonlinear equation in $\mathbb{R}$

---

```

function [x,nit]=falseposition(f,a,b,er,nmax)
%FALSEPOSITION - false position method
%call [X,NIT]=FALSEPOSITION(F,A,B,ER,NMAX)
%F - function
%A, B - endpoints
%ER - tolerance
%NMAX - maximum number of iterations

if nargin < 5, nmax=100; end
if nargin < 4, er=1e-3; end
nit=0; fa=f(a); fb=f(b);
for k=1:nmax
  x=a-(a-b)*fa/(fa-fb);
  if (x-a<er*(b-a)) || (b-x<er*(b-a))
    nit=k; return
  else
    fx=f(x);
    if sign(fx)==sign(fa)
      a=x; fa=fx;
    else
      b=x; fb=fx;
    end %if
  end %if
end %for
error('iteration number exceeded')
```

---

The convergence behavior is most easily analyzed if we assume that  $f$  is convex or concave on  $[a, b]$ . To fix ideas, suppose  $f$  is convex, say

$$f''(x) > 0, \quad x \in [a, b], \quad f(a) < 0, \quad f(b) > 0. \quad (7.4.2)$$

Then  $f$  has exactly one zero,  $\alpha$ , in  $[a, b]$ . Moreover, the secant connecting  $f(a)$  and  $f(b)$  lies entirely above the graph of  $y = f(x)$ , and hence intersects the real line to the left of  $\alpha$ . This will be the case of all subsequent secants, which means that the point  $x = b$  remains fixed while the other endpoint  $a$  gets continuously updated, producing a monotonically increasing sequence of approximation. The sequence defined by

$$x_{n+1} = x_n - \frac{x_n - b}{f(x_n) - f(b)} f(x_n), \quad n \in \mathbb{N}^*, \quad x_1 = a \quad (7.4.3)$$

is monotonically increasing sequence bounded from above by  $\alpha$ , therefore convergent to a limit  $x$ , and  $f(x) = 0$  (See Figure 7.2).

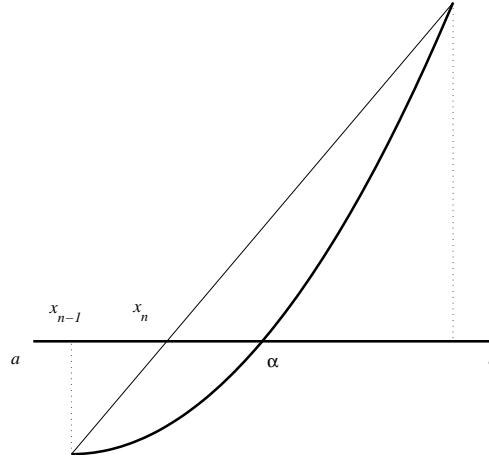


Figure 7.2: Method of false position

To determine the speed of convergence, we subtract  $\alpha$  from both side of (7.4.3) and use the fact that  $f(\alpha) = 0$ :

$$x_{n+1} - \alpha = x_n - \alpha - \frac{x_n - b}{f(x_n) - f(b)} [f(x_n) - f(\alpha)].$$

Now divide by  $x_n - \alpha$  to get

$$\frac{x_{n+1} - \alpha}{x_n - \alpha} = 1 - \frac{x_n - b}{f(x_n) - f(b)} \frac{f(x_n) - f(\alpha)}{x_n - \alpha}.$$

Letting here  $n \rightarrow \infty$  and using the fact that  $x_n \rightarrow \alpha$ , we obtain

$$\lim_{n \rightarrow \infty} \frac{x_{n+1} - \alpha}{x_n - \alpha} = 1 - (b - \alpha) \frac{f'(\alpha)}{f(b)}. \quad (7.4.4)$$

Thus, we have linear convergence with asymptotic error constant equal to

$$c = 1 - (b - a) \frac{f'(\alpha)}{f(b)}.$$

Due to the assumption of convexity,  $c \in (0, 1)$ . The proof when  $f$  is concave is analogous. If  $f$  is neither convex nor concave on  $[a, b]$ , but  $f \in C^2[a, b]$  and  $f''(\alpha) \neq 0$ ,  $f''$  has a constant sign in a

neighborhood of  $\alpha$  and for  $n$  large enough  $x_n$  will eventually come to lie in this neighborhood, and we can proceed as above.

As an example, we consider the equation  $\cos x \cosh x = 1$  on interval  $[\frac{3\pi}{2}, 2\pi]$ . Here is the code

```
ff=@(x) cos(x).*cosh(x)-1;
a=3/2*pi; b=2*pi;
[x,n]=falseposition(ff,a,b,eps)
```

The output is

```
x =
4.730040744862703
n =
75
```

Drawbacks. (i) Slow convergence; (ii) The fact that one of the endpoints remain fixed. If  $f$  is very flat near  $\alpha$ , the point  $a$  is nearby and  $b$  further away, the convergence is exceptionally slow.

## 7.5 Secant Method

The secant method is a simple variant of the method of false position in which it is no longer required that the function  $f$  have opposite signs at the endpoints of each interval generated, not even the initial interval. One starts with two arbitrary initial approximations  $x_0, x_1$  and continues with

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n), \quad n \in \mathbb{N}^* \quad (7.5.1)$$

This precludes the formation of a fixed false position, as in the method of false position, and hence suggest potentially faster convergence. Unfortunately, the “global convergence” no longer holds; the method converges only “locally”, that is only if the initial approximations  $x_0$  and  $x_1$  are sufficiently close to a root.

We need a relation between three consecutive errors

$$\begin{aligned} x_{n+1} - \alpha &= x_n - \alpha - \frac{f(x_n)}{f[x_{n-1}, x_n]} = (x_n - \alpha) \left( 1 - \frac{f(x_n) - f(\alpha)}{(x_n - \alpha)f[x_{n-1}, x_n]} \right) \\ &= (x_n - \alpha) \left( 1 - \frac{f[x_n, \alpha]}{f[x_{n-1}, x_n]} \right) = (x_n - \alpha) \frac{f[x_{n-1}, x_n] - f[x_n, \alpha]}{f[x_{n-1}, x_n]} \\ &= (x_n - \alpha)(x_{n-1} - \alpha) \frac{f[x_n, x_{n-1}, \alpha]}{f[x_{n-1}, x_n]}. \end{aligned}$$

Hence,

$$(x_{n+1} - \alpha) = (x_n - \alpha)(x_{n-1} - \alpha) \frac{f[x_n, x_{n-1}, \alpha]}{f[x_{n-1}, x_n]}, \quad n \in \mathbb{N}^* \quad (7.5.2)$$

From (7.5.2) it follows that if  $\alpha$  is a simple root ( $f(\alpha) = 0, f'(\alpha) \neq 0$ ) and if  $x_n \rightarrow \alpha$ , then convergence is faster than linear, at least if  $f \in C^2$  near  $\alpha$ . How fast is convergence?

We replace the ratio of divided difference in (7.5.2) by a constant, which is almost true when  $n$  is large. Letting then  $e_k = |x_k - \alpha|$ , we have

$$e_{n+1} = e_n e_{n-1} C, \quad C > 0$$

Multiplying both sides by  $C$  and defining  $E_n = Ce_n$  gives

$$E_{n+1} = E_n E_{n-1}, \quad E_n \rightarrow 0.$$

Taking logarithms on both sides, and defining  $y_n = \log \frac{1}{E_n}$  we obtain

$$y_{n+1} = y_n + y_{n-1}, \quad (7.5.3)$$

the well-known difference equation for the Fibonacci sequence.

The solution is

$$y_n = c_1 t_1^n + c_2 t_2^n,$$

where  $c_1, c_2$  are constants and

$$t_1 = \frac{1}{2}(1 + \sqrt{5}), \quad t_2 = \frac{1}{2}(1 - \sqrt{5}).$$

Since  $y_n \rightarrow \infty$ , we have  $c_1 \neq 0$  and  $y_n \sim c_1 t_1^n$ , as  $n \rightarrow \infty$ , since  $|t_2| < 1$ . Putting them back,  $\frac{1}{E_n} \sim e^{c_1 t_1^n}$ ,  $\frac{1}{e_n} \sim C e^{c_1 t_1^n}$ , so

$$\frac{e_{n+1}}{e_n} \sim \frac{C^{t_1} e^{c_1 t_1^n t_1}}{C e^{c_1 t_1^{n+1}}} = C^{t_1 - 1}, \quad n \rightarrow \infty.$$

The order of convergence, therefore, is  $t_1 = \frac{1 + \sqrt{5}}{2} \approx 1.61803 \dots$  (the golden ratio).

**Theorem 7.5.1.** Let  $\alpha$  be a simple zero of  $f$ . Let  $I_\varepsilon = \{x \in \mathbb{R} : |x - \alpha| < \varepsilon\}$  and assume  $f \in C^2[I_\varepsilon]$ . Define, for sufficiently small  $\varepsilon$

$$M(\varepsilon) = \max_{\substack{s \in I_\varepsilon \\ t \in I_\varepsilon}} \left| \frac{f''(s)}{2f'(t)} \right|. \quad (7.5.4)$$

Assume  $\varepsilon$  so small that

$$\varepsilon M(\varepsilon) < 1. \quad (7.5.5)$$

Then the secant method converges to the unique root  $\alpha \in I_\varepsilon$  for any starting values  $x_0 \neq x_1$  with  $x_0 \in I_\varepsilon$ ,  $x_1 \in I_\varepsilon$ .

**Remark 7.5.2.** Note that  $\lim_{\varepsilon \rightarrow 0} M(\varepsilon) = \left| \frac{f''(\alpha)}{2f'(\alpha)} \right| < \infty$ , so that (7.5.5)) can certainly be satisfied for  $\varepsilon$  small enough. The local nature of convergence is thus quantified by the requirement  $x_0, x_1 \in I_\varepsilon$ .  $\diamond$

*Proof.* First of all, observe that  $\alpha$  is the only zero of  $f$  in  $I_\varepsilon$ . This follows from Taylor's formula applied at  $x = \alpha$ :

$$f(x) = f(\alpha) + (x - \alpha)f'(\alpha) + \frac{(x - \alpha)^2}{2}f''(\xi)$$

where  $f(\alpha) = 0$  and  $\xi \in (x, \alpha)$  (or  $(\alpha, x)$ ). Thus, if  $x \in I_\varepsilon$ , then also  $\xi \in I_\varepsilon$ , and we have

$$f(x) = (x - \alpha)f'(\alpha) \left[ 1 + \frac{x - \alpha}{2} \frac{f''(\xi)}{f'(\alpha)} \right]$$

Here, if  $x \neq \alpha$ , all three factors are different from zero, the last one since by assumption

$$\left| \frac{x - \alpha}{2} \frac{f''(\xi)}{f'(\alpha)} \right| \leq \varepsilon M(\varepsilon) < 1.$$

Thus,  $f$  on  $I_\varepsilon$  can only vanish at  $x = \alpha$ .

Next we show that for all  $n$ ,  $x_n \in I_\varepsilon$  and two consecutive iterates are distinct, unless  $f(x_n) = 0$  for some  $n$ , in which case  $x_n = \alpha$  and the method converge in a finite number of steps. We prove this by induction: assume that  $x_{n-1}, x_n \in I_\varepsilon$  and  $x_n \neq x_{n-1}$ . (By assumption this is true for  $n = 1$ .) Then, from known properties of divided differences, and by our assumption that  $f \in C^2[I_\varepsilon]$ , we have

$$f[x_{n-1}, x_n] = f'(\xi_1), \quad f[x_{n-1}, x_n, \alpha] = \frac{1}{2}f''(\xi_2), \quad \xi_i \in I_\varepsilon, \quad i = 1, 2.$$

Therefore, by (7.5.2),

$$|x_{n+1} - \alpha| \leq \varepsilon^2 \left| \frac{f''(\xi_n)}{2f'(\xi_1)} \right| \leq \varepsilon \varepsilon M(\varepsilon) < \varepsilon,$$

that is,  $x_{n+1} \in I_\varepsilon$ . Furthermore, by the relation between three consecutive errors (7.5.2),  $x_{n+1} \neq x_n$  unless  $f(x_n) = 0$ , hence  $x_n = \alpha$ .

Finally, using again (7.5.2) we have

$$|x_{n+1} - \alpha| \leq |x_n - \alpha| \varepsilon M(\varepsilon)$$

which, applied repeatedly, yields

$$|x_{n+1} - \alpha| \leq |x_n - \alpha| \varepsilon M(\varepsilon) \leq \dots \leq [\varepsilon M(\varepsilon)]^{n-1} |x_1 - \alpha|.$$

Since  $\varepsilon M(\varepsilon) < 1$ , it follows that the method converges and  $x_n \rightarrow \alpha$  as  $n \rightarrow \infty$ .  $\square$

Since only one evaluation of  $f$  is required in each iteration step, the secant method has the efficiency index  $p = \frac{1+\sqrt{5}}{2} \approx 1.61803 \dots$ . An implementation of this method is given in MATLAB Source 7.2.

## 7.6 Newton's method

Newton's method can be thought of as a limit case of the secant method, when  $x_{n-1} \rightarrow x_n$ . The result is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.6.1}$$

where  $x_0$  is some appropriate initial approximation. Another, more fruitful interpretation is that of linearization of the equation  $f(x) = 0$  at  $x = x_n$ :

$$f(x) \approx f(x_n) + (x - x_n)f'(x_n) = 0.$$

Viewed in this manner, Newton's method can be vastly generalized to nonlinear equations of all kinds (nonlinear equations, functional equations, in which case the derivative  $f'$  is to be understood as a Fréchet derivative, and the iteration is

$$x_{n+1} = x_n - [f'(x_n)]^{-1} f(x_n) \tag{7.6.2}$$

The study of error in Newton's method is virtually the same as the one for the secant method.

$$\begin{aligned} x_{n+1} - \alpha &= x_n - \alpha - \frac{f(x_n)}{f'(x_n)} \\ &= (x_n - \alpha) \left[ 1 - \frac{f(x_n) - f(\alpha)}{(x_n - \alpha)f'(x_n)} \right] \\ &= (x_n - \alpha) \left( 1 - \frac{f[x_n, \alpha]}{f[x_n, x_n]} \right) \\ &= (x_n - \alpha)^2 \frac{f[x_n, x_n, \alpha]}{f[x_n, x_n]}. \end{aligned} \tag{7.6.3}$$

**MATLAB Source 7.2** Secant method for nonlinear equations in  $\mathbb{R}$ 

```

function [z,ni]=secant(f,x0,x1,ea,er,Nmax)
%SECANT - secant method in R
%input
%f - function
%x0,x1 - starting values
%ea,er - absolute and relative error, respectively
%Nmax - maximim number of iterations
%output
%z - approximate root
%ni - actual no. of iterations

if nargin<6, Nmax=50; end
if nargin<5, er=0; end
if nargin<4, ea=1e-3; end
xv=x0; fv=f(xv); xc=x1; fc=f(xc);
for k=1:Nmax
    xn=xc-fc*(xc-xv)/(fc-fv);
    if abs(xn-xc)<ea+er*xn %success
        z=xn;
        ni=k;
        return
    end
    %prepare next iteration
    xv=xc; fv=fc; xc=xn; fc=feval(f,xn);
end
%failure
error('maximum iteration number exceeded')

```

Therefore, if  $x_n \rightarrow \alpha$ , then

$$\lim_{n \rightarrow \infty} \frac{x_{n+1} - \alpha}{(x_n - \alpha)^2} = \frac{f''(\alpha)}{2f'(\alpha)}$$

that is, Newton's method has the order of convergence  $p = 2$  if  $f''(\alpha) \neq 0$ . The efficiency index of Newton method is  $\sqrt{2} = 1.41421 \dots$ , because it requires at each step two function evaluations (the function and its derivative).

For the convergence of Newton's method we have the following result.

**Theorem 7.6.1.** Let  $\alpha$  be a simple root of the equation  $f(x) = 0$  and  $I_\varepsilon = \{x \in \mathbb{R} : |x - \alpha| \leq \varepsilon\}$ . Assume that  $f \in C^2[I_\varepsilon]$ . Define

$$M(\varepsilon) = \max_{\substack{s \in I_\varepsilon \\ t \in I_\varepsilon}} \left| \frac{f''(s)}{2f'(t)} \right| \quad (7.6.4)$$

If  $\varepsilon$  is so small that

$$2\varepsilon M(\varepsilon) < 1, \quad (7.6.5)$$

then for every  $x_0 \in I_\varepsilon$ , Newton's method is well defined and converges quadratically to the only root  $\alpha \in I_\varepsilon$ .

The extra factor 2 in (7.6.5) comes from the requirement that  $f'(x) \neq 0$  for  $x \in I_\varepsilon$ .

The stopping criterion for Newton's method

$$|x_n - x_{n-1}| < \varepsilon$$

is based on the following result.

**Proposition 7.6.2.** *Let  $(x_n)$  be the sequence of approximations generated by Newton's method. If  $\alpha$  is a simple root in  $[a, b]$ ,  $f \in C^2[a, b]$  and the method is convergent, then there exists an  $n_0 \in \mathbb{N}$  such that*

$$|x_n - \alpha| \leq |x_n - x_{n-1}|, \quad n > n_0.$$

*Proof.* We shall first show that

$$|x_n - \alpha| \leq \frac{1}{m_1} |f(x_n)|, \quad m_1 \leq \inf_{x \in [a, b]} |f'(x)|. \quad (7.6.6)$$

Using Lagrange Theorem,  $f(\alpha) - f(x_n) = f'(\xi)(\alpha - x_n)$ , where  $\xi \in (\alpha, x_n)$  (or  $(x_n, \alpha)$ ). Relations  $f(\alpha) = 0$  and  $|f'(x)| \geq m_1$ , for  $x \in (a, b)$ , imply that  $|f(x_n)| \geq m_1 |\alpha - x_n|$ , that is (7.6.6).

Based on Taylor formula, we have

$$f(x_n) = f(x_{n-1}) + (x_n - x_{n-1})f'(x_{n-1}) + \frac{1}{2}(x_n - x_{n-1})^2 f''(\mu), \quad (7.6.7)$$

where  $\mu \in (x_{n-1}, x_n)$  or  $\mu \in (x_n, x_{n-1})$ . Due to the way which we obtain an approximation in Newton's method, we have  $f(x_{n-1}) + (x_n - x_{n-1})f'(x_{n-1}) = 0$  and from (7.6.7) we obtain

$$|f(x_n)| = \frac{1}{2}(x_n - x_{n-1})^2 |f''(\mu)| \leq \frac{1}{2}(x_n - x_{n-1})^2 \|f''\|_\infty,$$

and based on (7.6.6) it follows that

$$|\alpha - x_n| \leq \frac{\|f''\|_\infty}{2m_1} (x_n - x_{n-1})^2.$$

Since we assumed the convergence of the method, there exists a  $n_0 \in \mathbb{N}$  such that

$$\frac{\|f''\|_\infty}{2m_1} (x_n - x_{n-1}) < 1, \quad n > n_0,$$

and hence

$$|x_n - \alpha| \leq |x_n - x_{n-1}|, \quad n > n_0.$$

□

The geometric interpretation of Newton's method is given in Figure 7.3, and an implementation is given in MATLAB Source 7.3.

The choice of starting value is, in general, a difficult task. In practice, one chooses a value, and if after a fixed maximum number of iterations the desired accuracy, tested by an usual stopping criterion, is not attained, another starting value is chosen. For example, if the root is isolated in a certain interval  $[a, b]$ , and  $f''(x) \neq 0$ ,  $x \in (a, b)$ , a choice criterion is  $f(x_0)f''(x_0) > 0$ . Another criterion is: Let  $f \in C^2[a, b]$  be such that

- $f$  is convex (or concave) on  $[a, b]$ ;
- $f(a)f(b) < 0$ ;
- the tangents at the endpoints of  $[a, b]$  intersect the real line within  $[a, b]$ .

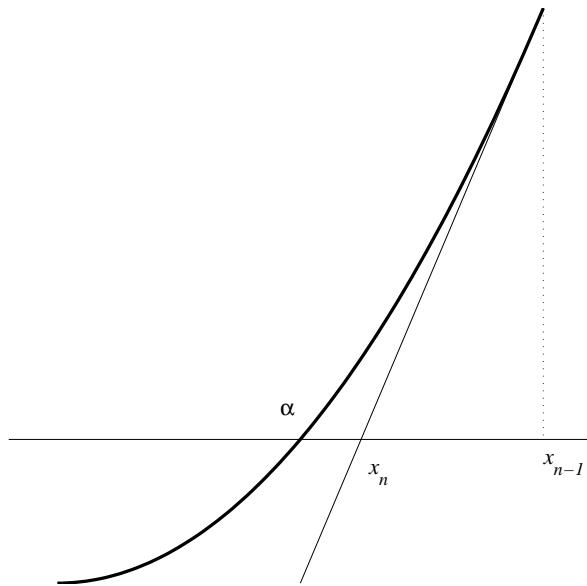


Figure 7.3: Newton's method

Then, Newton's method converges globally, i.e. for any  $x_0 \in [a, b]$ .

**Example 7.6.3.** We wish to compute  $\alpha = \sqrt{a}$ ,  $a > 0$ . Starting from the equation

$$f(x) = x^2 - a = 0,$$

(7.6.1) becomes

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right), \quad n = 0, 1, 2, \dots \quad (7.6.8)$$

This method was used by Babylonians long before Newton. Because the convexity of  $f$ , it is clear that the iteration (7.6.8) converges to the positive square root for each  $x_0 > 0$  and is monotonically decreasing (except for the first step in the case  $0 < x_0 < \alpha$ ). This is an elementary example of *global convergence*.  $\diamond$

**Example 7.6.4 (Cycle).** Let  $f(x) = \sin(x)$ ,  $|x| < \frac{\pi}{2}$ . There is exactly one root in this interval,  $\alpha = 0$ . Newton's method becomes

$$x_{n+1} = x_n - \tan x_n, \quad n = 0, 1, 2, \dots \quad (7.6.9)$$

It exhibits a strange behavior (see Figure 7.4). If  $x_0 = x^*$ , where is the smallest positive root of

$$\tan x^* = 2x^*, \quad (7.6.10)$$

---

**MATLAB Source 7.3** Newton method for nonlinear equations in  $\mathbb{R}$ 

---

```

function [z,ni]=Newtons(f,fd,x0,ea,er,Nmax)
%NEWTONS - Newton method in R
%Input
%f - function
%fd - derivative
%x0 - starting value
%ea,er - absolute and relative error, respectively
%Nmax - maximum number of iterations
%Output
%z - approximate solution
%ni - actual no. of iterations

if nargin<6, Nmax=50; end
if nargin<5, er=0; end
if nargin<4, ea=1e-3; end
xv=x0;
for k=1:Nmax
    xc=xv-f(xv)/fd(xv);
    if abs(xc-xv)<ea+er*abs(xc) %success
        z=xc;
        ni=k;
        return
    end
    xv=xc; %prepare next iteration
end
%failure
error('maximum iteration number exceeded')

```

---

then  $x_1 = -x^*$ ,  $x_2 = x^*$ ; that is Newton's method cycles forever. This is called a *cycle*. For this starting value, Newton's method does not converge, let alone to  $\alpha = 0$ . It does converge, however, for any starting value  $x_0$  with  $|x_0| < x^*$ , generating a sequence of alternatively increasing and decreasing approximations  $x_n$  converging necessarily to  $\alpha = 0$ . The value of the critical number  $x^*$  can be computed by Newton's method applied to (7.6.10). The result is  $x^* = 1.165561185207211 \dots$ . Here is an example of *local convergence*, since convergence does not hold for all  $x_0 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ . (If  $x_0 = \frac{\pi}{2}$ , the first derivative vanishes.) What is interesting, in double precision floating-point representation the cycle does not appear. The example (M-file `cycletest.m`):

```

%cycle test
f1=@(x) tan(x)-2*x;
f1d=@(x) cos(x).^(2)-2;
[x0,ni]=Newtons(f1,f1d,3*pi/7,0,eps,200)
[x,n2]=Newtons(@(sin,@cos,x0,0,eps,200)
[x1,n3]=Newtons(@(sin,@cos,x0-eps,eps,eps,200)

```

provides the following results:

$x_0 =$

```

1.165561185207211
ni =
    7
x =
21.991148575128552
n2 =
    27
x1 =
    0
n3 =
    26

```

Can you explain why? (Hint: compute  $f'(x_0)$  and  $f'(x_0 - \epsilon)$ .)  $\diamond$

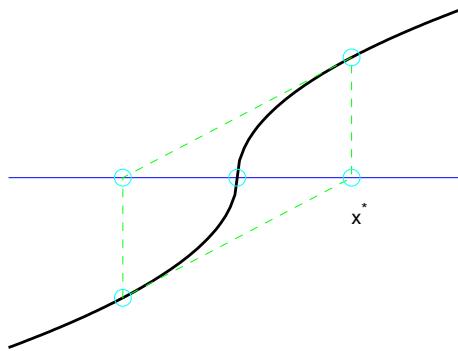


Figure 7.4: A cycle in Newton's method

For another example on cycles see Problem 7.2.

**Example 7.6.5.**  $f(x) = x^{20} - 1$ ,  $x > 0$ . There is exactly one positive simple root,  $\alpha = 1$ . Newton's method yields the iteration

$$x_{n+1} = \frac{19}{20}x_n + \frac{1}{20x_n^{19}}, \quad n = 0, 1, 2, \dots \quad (7.6.11)$$

which provides a good example to illustrate that unless one starts sufficiently close to the desired root, it may take a long time to approach it. If we take  $x_0 = \frac{1}{2}$ ; then  $x_1 \approx \frac{2^{19}}{20} = 2.62144 \times 10^4$ , a very large number. It takes a long time to arrive in a small vicinity of  $\alpha = 1$ , since for  $x_n$  large, one has

$$x_{n+1} \approx \frac{19}{20}x_n, \quad x_n \gg 1.$$

At each step the approximation is reduced only by a fraction  $\frac{19}{20} = 0.95$ . It takes about 200 steps to get back to near the desired root. But once we come close to  $\alpha = 1$ , the iteration speeds up dramatically and converges to the root quadratically. Since  $f$  is convex, we actually have global convergence on  $[0, \infty]$ , but as we have seen, this is not very useful.  $\diamond$

## 7.7 Fixed Point Iteration

Often, in applications, a nonlinear equation presents itself in the form of a *fixed point problem*: find  $x$  such that

$$x = \varphi(x). \quad (7.7.1)$$

A number  $\alpha$  satisfying this equation is called a *a fixed point* of  $\varphi$ . Any equation  $f(x) = 0$  can, in fact, (in many different ways) be written equivalently in the form (7.7.1). For example, if  $f'(x) \neq 0$  in the interval of interest, we can take

$$\varphi(x) = x - \frac{f(x)}{f'(x)}. \quad (7.7.2)$$

If  $x_0$  is an initial approximation of a fixed point  $\alpha$  of (7.7.1), the *fixed point iteration* generates a sequence of approximates by

$$x_{n+1} = \varphi(x_n). \quad (7.7.3)$$

If it converges, it clearly converges to a fixed point of  $\varphi$  if  $\varphi$  is continuous. Note that (7.7.3) is precisely Newton's method for solving  $f(x) = 0$  if  $\varphi$  is defined by (7.7.2). So Newton's method can be viewed as a fixed point iteration, but not the secant method.

For any iteration of the form (7.7.3), assuming that  $x_n \rightarrow \alpha$  when  $n \rightarrow \infty$ , it is straightforward to determine the order of convergence. Suppose indeed that at the fixed point  $\alpha$  we have

$$\varphi'(\alpha) = \varphi''(\alpha) = \cdots = \varphi^{(p-1)}(\alpha) = 0, \quad \varphi^p(\alpha) \neq 0 \quad (7.7.4)$$

We assume that  $\varphi \in C^p$  on a neighborhood of  $\alpha$ . This defines the integer  $p \geq 1$ . We then have by Taylor's theorem

$$\begin{aligned} \varphi(x_n) &= \varphi(\alpha) + (x_n - \alpha)\varphi'(\alpha) + \cdots + \frac{(x_n - \alpha)^{p-1}}{(p-1)!}\varphi^{(p-1)}(\alpha) \\ &\quad + \frac{(x_n - \alpha)^p}{p!}\varphi^{(p)}(\xi_n) = \varphi(\alpha) + \frac{(x_n - \alpha)^p}{p!}\varphi^{(p)}(\xi_n), \end{aligned}$$

where  $\xi_n$  is between  $\alpha$  and  $x_n$ . Since  $\varphi(x_n) = x_{n+1}$  and  $\varphi(\alpha) = \alpha$  we get

$$\frac{x_{n+1} - \alpha}{(x_n - \alpha)^p} = \frac{1}{p!}\varphi^{(p)}(\xi_n).$$

As  $x_n \rightarrow \alpha$ , since  $\xi_n$  is trapped between  $x_n$  and  $\alpha$ , we conclude by the continuity of  $\varphi^{(p)}$  at  $\alpha$ , that

$$\lim_{n \rightarrow \infty} \frac{x_{n+1} - \alpha}{(x_n - \alpha)^p} = \frac{1}{p!}\varphi^{(p)}(\alpha) \neq 0. \quad (7.7.5)$$

This shows that convergence is exactly of order  $p$ , and the asymptotic error constant is

$$c = \frac{1}{p!}\varphi^{(p)}(\alpha). \quad (7.7.6)$$

Combining this with the usual local convergence argument, we obtain the following result.

**Theorem 7.7.1.** *Let  $\alpha$  be a fixed point of  $\varphi$  and  $I_\varepsilon = \{x \in \mathbb{R} : |x - \alpha| \leq \varepsilon\}$ . Assume  $\varphi \in C^p[I_\varepsilon]$  satisfies (7.7.4). If*

$$M(\varepsilon) := \max_{t \in I_\varepsilon} |\varphi'(t)| < 1 \quad (7.7.7)$$

*then the fixed point iteration converges to  $\alpha$ , for any  $x_0 \in I_\varepsilon$ . The order of convergence is  $p$ , and the asymptotic error constant is given by (7.7.6).*

## 7.8 Newton's Method for Multiple zeros

If  $\alpha$  is a zero of multiplicity  $m$ , then the convergence order of Newton method is only one. Indeed, let

$$\varphi(x) = x - \frac{f(x)}{f'(x)}.$$

Since

$$\varphi'(x) = \frac{f(x)f''(x)}{[f'(x)]^2}$$

the process should be convergent if  $\varphi'(\alpha) = 1 - 1/m < 1$ .

One way to avoid multiple zeros is to solve the modified equation

$$u(x) := \frac{f(x)}{f'(x)} = 0$$

which has the same roots as  $f$ , but simple. Newton's method for the modified problem has the form

$$x_{k+1} = x_k - \frac{u(x_k)}{u'(x_k)} = \frac{f(x_k)f'(x_k)}{[f'(x_k)]^2 - f(x_k)f''(x_k)}. \quad (7.8.1)$$

Since  $\alpha$  is a simple zero of  $u$ , the convergence of (7.8.1) is always quadratic. The only theoretical disadvantage of (7.8.1) is the additionally required second derivative of  $f$  and the slightly higher cost of the determination of  $x_{k+1}$  from  $x_k$ . In practice, this is a weakness, since the denominator of (7.8.1) could be very small on a neighborhood of  $\alpha$  when  $x_k \rightarrow \alpha$ .

Quadratic convergence for zeros of higher multiplicities can be achieved not only by modifying the problem, but also by modifying the method. In a neighborhood of a zero  $\alpha$  with multiplicity  $m$ , the relation

$$f(x) = (x - \alpha)^m \varphi(x) \approx (x - \alpha)^m \cdot c, \quad (7.8.2)$$

holds. This leads to

$$\frac{f(x)}{f'(x)} \approx \frac{x - \alpha}{m} \Rightarrow \alpha \approx x - m \frac{f(x)}{f'(x)}.$$

The accordingly modified sequence

$$x_{k+1} := x_k - m \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \dots \quad (7.8.3)$$

converges quadratically even at multiple zeros, provided the correct value of the multiplicity  $m$  is used in (7.8.3).

The efficiency of the Newton variant (7.8.3) depends critically on the correctness of the value  $m$  used. If this value cannot be determined analytically, then a good estimate should at least be used.

Provided that

$$|x_k - \alpha| < |x_{k-1} - \alpha| \wedge |x_k - \alpha| < |x_{k-2} - \alpha|$$

$x_k$  can be substituted for  $\alpha$  in (7.8.2):

$$\begin{aligned} f(x_{k-1}) &\approx (x_{k-1} - x_k)^m \cdot c \\ f(x_{k-2}) &\approx (x_{k-2} - x_k)^m \cdot c. \end{aligned}$$

Then this system is solved with respect to  $m$ :

$$m \approx \frac{\log [f(x_{k-1})/f(x_{k-2})]}{\log [(x_{k-1} - x_k)/(x_{k-2} - x_k)]}.$$

This estimate of the multiplicity can be used, for example, in (7.8.3).

## 7.9 Algebraic Equations

There are many iterative methods specifically designed to solve algebraic equations. Here we only describe how Newton's method applies to this context, essentially confining ourselves to a discussion of an efficient way to evaluate simultaneously the value of a polynomial and its first derivative. In the special case where all zeros of the polynomial are known to be real and simple, we describe an improved variant of Newton's method.

**Newton's method applied to algebraic equations.** We consider an algebraic equation of degree  $d$ ,

$$f(x) = 0, \quad f(x) = x^d + a_{d-1}x^{d-1} + \cdots + a_0, \quad (7.9.1)$$

where the leading coefficient is assumed (without restricting generality) to be 1 and where we may also assume  $a_0 \neq 0$  without loss of generality. For simplicity we assume all coefficients to be real.

To apply Newton's method to (7.9.1), one needs good methods for evaluating a polynomial and its derivative.

*Horner's scheme* is good for this purpose:

```
bd := 1; cd := 1;
for k = d - 1 downto 1 do
    bk := tbk+1 + ak;
    ck := tcck+1 + bk;
end for
b0 := tb1 + a0;
```

Then  $f(t) = b_0$ ,  $f'(t) = c_1$ .

We proceed as follows:

One applies Newton's method, computing simultaneous  $f(x_n)$  and  $f'(x_n)$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Then we apply Newton's method to the polynomial  $\frac{f(x)}{x - \alpha}$ . For complex roots, one begins with  $x_0$  complex and all computations are done in complex arithmetic. It is possible to divide by quadratic factors and to compute entirely in real arithmetic – Bairstow's method. This method for decreasing the degree could lead to large errors. A way of improvement is to use the approximated roots as starting values for Newton's method to the original polynomial.

## 7.10 Newton's method for systems of nonlinear equations

Newton's method can be easily adapted to deal with systems of nonlinear equations

$$F(x) = 0, \quad (7.10.1)$$

where  $F : \Omega \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ , and  $x, F(x) \in \mathbb{R}^n$ . The system (7.10.1) can be written explicitly

$$\begin{cases} F_1(x_1, \dots, x_n) = 0 \\ \vdots \\ F_n(x_1, \dots, x_n) = 0 \end{cases}$$

Let  $F'(x^{(k)})$  be the Jacobian matrix of  $F$  in  $x^{(k)}$ :

$$J := F'(x^{(k)}) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1}(x^{(k)}) & \dots & \frac{\partial F_1}{\partial x_n}(x^{(k)}) \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1}(x^{(k)}) & \dots & \frac{\partial F_n}{\partial x_n}(x^{(k)}) \end{bmatrix}. \quad (7.10.2)$$

The quantity  $1/f'(x)$  is replaced by the inverse of the Jacobian in  $x^{(k)}$ :

$$x^{(k+1)} = x^{(k)} - [F'(x^{(k)})]^{-1}F(x^{(k)}). \quad (7.10.3)$$

We write iteration under the form

$$x^{(k+1)} = x^{(k)} + w^{(k)}. \quad (7.10.4)$$

Note that  $w_k$  is the solution of the system having  $n$  equations and  $n$  unknowns

$$F'(x^{(k)})w^{(k)} = -F(x^{(k)}). \quad (7.10.5)$$

It is more efficient and convenient that, instead of computing the inverse Jacobian to solve the system (7.10.5) and of using the form (7.10.4) of iteration.

**Theorem 7.10.1.** *Let  $\alpha$  be a solution of equation  $F(x) = 0$  and suppose that in closed ball  $B(\delta) \equiv \{x : \|x - \alpha\| \leq \delta\}$ , there exists the Jacobi matrix of  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , it is nonsingular and satisfies a Lipschitz condition*

$$\|F'(x) - F'(y)\|_{\infty} \leq c\|x - y\|_{\infty}, \quad \forall x, y \in B(\delta), \quad c > 0.$$

We set  $\gamma = c \max \{ \|F'(x)]^{-1}\|_{\infty} : \|\alpha - x\|_{\infty} \leq \delta \}$  and  $0 < \varepsilon < \min \{ \delta, \gamma^{-1} \}$ . Then for any initial approximation  $x^{(0)} \in B(\varepsilon) := \{x : \|x - \alpha\|_{\infty} \leq \varepsilon\}$  Newton method is convergent, and the vectors  $e^{(k)} := \alpha - x^{(k)}$  satisfy the following inequalities:

- (a)  $\|e^{(k+1)}\|_{\infty} \leq \gamma\|e^{(k)}\|_{\infty}^2$
- (b)  $\|e^{(k)}\|_{\infty} \leq \gamma^{-1}(\gamma\|e^{(0)}\|_{\infty})^{2^k}$ .

*Proof.* If  $F'$  is continuous on the segment joining the points  $x, y \in \mathbb{R}^n$ , Lagrange's Theorem implies

$$F(x) - F(y) = J_k(x - y),$$

where

$$J_k = \begin{bmatrix} \frac{\partial F_1}{\partial x_1}(\xi_1) & \dots & \frac{\partial F_1}{\partial x_n}(\xi_1) \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1}(\xi_n) & \dots & \frac{\partial F_n}{\partial x_n}(\xi_n) \end{bmatrix} \Rightarrow$$

$$\begin{aligned} e^{(k+1)} &= e^{(k)} - [F'(x^{(k)})]^{-1}(F(\alpha) - F(x^{(k)})) = e^{(k)} - [F'(x^{(k)})]^{-1}J_k e^{(k)} \\ &= [F'(x^{(k)})]^{-1}(F'(x^{(k)}) - J_k)e^{(k)} \end{aligned}$$

and (a) follows. From Lipschitz condition one gets

$$\|F'(x^{(k)}) - J_k\|_{\infty} \leq c \max_{j=1,n} \|x^{(k)} - \xi^{(j)}\| \leq c\|x^{(k)} - \alpha\|$$

Thus, if  $\|\alpha - x^{(k)}\|_{\infty} \leq \varepsilon$ , then  $\|\alpha - x^{(k+1)}\|_{\infty} \leq (\gamma\varepsilon)\varepsilon \leq \varepsilon$ . Since (a) holds for any  $k$ , (b) follows immediately.  $\square$

**MATLAB Source 7.4** Newton method in  $\mathbb{R}$  and  $\mathbb{R}^n$ 

```

function [z,ni]=Newton(f,fd,x0,ea,er,nmax)
%NEWTON - Newton method for nonlinear equations in R and R^n
%call [z,ni]=Newton(f,fd,x0,ea,er,nmax)
%Input
%f - function
%fd - derivative
%x0 - starting value
%ea,er - absolute and relative error, respectively
%nmax - maximum number of iterations
%Output
%z - approximate solution
%ni - actual no. of iterations

if nargin < 6, nmax=50; end
if nargin < 5, er=0; end
if nargin < 4, ea=1e-3; end
xp=x0(:); %previous x
for k=1:nmax
    xc=xp-fd(xp)\f(xp);
    if norm(xc-xp,inf)<ea+er*norm(xc,inf)
        z=xc; %success
        ni=k;
        return
    end
    xp=xc;
end
error('maximum iteration number exceeded')

```

---

MATLAB Source 7.4 gives an implementation of Newton's method that functions for scalar equations and also for systems.

**Example 7.10.2.** Consider the nonlinear system:

$$\begin{aligned} 3x_1 - \cos(x_1 x_2) - \frac{1}{2} &= 0, \\ x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06 &= 0, \\ e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3} &= 0 \end{aligned}$$

Here are the function and the jacobian in MATLAB:

```

function y=fs3(x)
y=[3*x(1)-cos(x(2)*x(3))-1/2;...
    x(1)^2-81*(x(2)+0.1)^2+sin(x(3))+1.06;...
    exp(-x(1)*x(2))+20*x(3)+(10*pi-3)/3];

function y=fs3d(x)

```

```

y=[3,x(3)*sin(x(2)*x(3)), x(2)*sin(x(2)*x(3));...
2*x(1), -162*(x(2)+0.1), cos(x(3));...
-x(2)*exp(-x(1)*x(2)), -x(1)*exp(-x(1)*x(2)), 20];

```

Finally, we give the call example:

```
>> [z,ni]=Newton(@fs3,@fs3d,x0,1e-9);
```

Applying Newton's method with starting value  $x^{(0)} = [0.1, 0.1, -0.1]^T$  one obtains the value given in Table 7.1. The desired accuracy,  $10^{-9}$ , is attained after 5 iterations.  $\diamond$

$k$	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$	$\ x^{(k)} - x^{(k-1)}\ _\infty$
0	0.1000000000	0.1000000000	-0.1000000000	—
1	0.4998696729	0.0194668485	-0.5215204719	0.42152
2	0.5000142402	0.0015885914	-0.5235569643	0.0178783
3	0.5000001135	0.0000124448	-0.5235984501	0.00157615
4	0.5000000000	0.0000000008	-0.5235987756	1.2444e-005
5	0.5000000000	0.0000000000	-0.5235987756	7.75786e-010
6	0.5000000000	-0.0000000000	-0.5235987756	1.11022e-016

Table 7.1: Results for Example 7.10.2

## 7.11 Quasi-Newton Methods

An important weakness of Newton's method for solution of systems of nonlinear equation is the necessity to compute the Jacobian matrix and to solve a linear  $n \times n$  system having this matrix. To illustrate the size of such a weakness, let us evaluate the amount of computation associated to an iteration of Newton method. The Jacobian matrix associated to a system of  $n$  nonlinear equation  $F(x) = 0$  requires the evaluation of the  $n^2$  partial derivatives of the  $n$  component of  $F$ . In most situations, evaluation of partial derivatives is not convenient and often impossible. The computational effort required by an iteration of Newton's method is at least  $n^2 + n$  scalar function evaluations ( $n^2$  for Jacobian and  $n$  for  $F$ ) and  $O(n^3)$  flops for the solution of nonlinear system. This amount of computation is prohibitive, excepting small values of  $n$ , and scalar functions which can be evaluated easily. So, it is natural to focus our attention to reduce the number of evaluation and to avoid the solution of a linear system at each step.

With the scalar secant method, the next iteration,  $x^{(k+1)}$ , is obtained as the solution of the linear equation

$$\bar{l}_k = f(x^{(k)}) + (x - x^{(k)}) \frac{f(x^{(k)} + h_k) - f(x^{(k)})}{h_k} = 0.$$

Here the linear function  $\bar{l}_k$  can be interpreted in two ways:

1.  $\bar{l}_k$  is an approximation of the tangent equation

$$l_k(x) = f(x^{(k)}) + (x - x^{(k)}) f'(x^{(k)});$$

2.  $\bar{l}_k$  is the linear interpolation of  $f$  between the points  $x^{(k)}$  and  $x^{(k+1)}$ .

By extending the scalar secant method to  $n$  dimensions, different generalization of secant method are obtained depending on the interpretation of  $\bar{l}_k$ . The first interpretation leads to the discrete Newton method, and the second one to interpolation methods.

The discrete Newton method is obtained replacing  $F'(x)$  in Newton's method (7.10.3) by a discrete approximation  $A(x, h)$ . The partial derivatives in the Jacobian matrix (7.10.2) are replaced by divided differences

$$A(x, h)e_i := [F(x + h_i e_i) - F(x)]/h_i, \quad i = \overline{1, n}, \quad (7.11.1)$$

where  $e_i \in \mathbb{R}^n$  is the  $i$ -th unit vector and  $h_i = h_i(x)$  is the step length of the discretization. A possible choice of step length is

$$h_i := \begin{cases} \varepsilon|x_i|, & \text{if } x_i \neq 0; \\ \varepsilon, & \text{otherwise,} \end{cases}$$

with  $\varepsilon := \sqrt{\text{eps}}$ , where  $\text{eps}$  is the machine epsilon.

### 7.11.1 Linear Interpolation

In linear interpolation, each of the tangent planes is replaced by a (hyper-)plane which interpolates the component function  $F_i$  of  $F$  at  $n + 1$  given points  $x^{k,j}$ ,  $j = \overline{0, n}$ , located in a neighborhood of  $x^{(k)}$ , that is one determines vectors  $a^{(i)}$  and scalars  $\alpha_i$  in such a way that for

$$L_i(x) = \alpha_i + a^{(i)T}x, \quad i = \overline{1, n} \quad (7.11.2)$$

the following relations hold

$$L_i(x^{k,j}) = F_i(x^{k,j}), \quad i = \overline{1, n}, \quad j = \overline{0, n}.$$

The next iterate  $x^{(k+1)}$  is obtained by intersecting the  $n$  hyperplanes (7.11.2) in  $\mathbb{R}^{n+1}$  and the hyperplane  $y = 0$ .  $x^{(k+1)}$  is the solution of the linear system of equations

$$L_i(x) = 0, \quad i = \overline{1, n}. \quad (7.11.3)$$

Depending on the selection of interpolation points  $x^{k,j}$ , numerous different methods are derived, and among them the best known are Brown's method and Brent's method. Brown's method combines the processes of approximating  $F'$  and that of solving the system of linear equations (7.11.3) using Gaussian elimination. Brent's method uses a QR factorization for solving (7.11.3). Both methods are members of a class of methods quadratically convergent (like Newton's method) but require only  $(n^2 + 3n)/2$  function evaluations per iteration.

In a comparative study Moré and Cosnard [67] found that Brent's method is often better than Brown's method, and that the discrete Newton method is usually the most efficient if the F-evaluations do not require too much computational effort.

### 7.11.2 Modification Method

Iterative methods of exceptionally efficiency can be constructed by using an approximation  $A_k$  of  $F'(x^{(k)})$ , which is derived from  $A_{k-1}$  by a rank 1 modification, i.e., by adding a matrix of rank 1:

$$A_{k+1} := A_k + u^{(k)} \left[ v^{(k)} \right]^T, \quad u^{(k)}, v^{(k)} \in \mathbb{R}^n, \quad k = 0, 1, 2, \dots$$

According to the Sherman-Morrison formula (see [24])

$$\left( A + uv^T \right)^{-1} = A^{-1} - \frac{1}{1 + v^T A^{-1} u} A^{-1} u v^T A^{-1}$$

for  $B_{k+1} := A_{k+1}^{-1}$  the recursion

$$B_{k+1} = B_k - \frac{B_k u^{(k)} \left[ v^{(k)} \right]^T B_k}{1 + [v^{(k)}]^T B_k u^{(k)}}, \quad k = 0, 1, 2, \dots,$$

holds, provided that  $1 + \left[ v^{(k)} \right]^T B_k u^{(k)} \neq 0$ . Thus, the necessity of solving a system of linear equations in every iteration is avoided; a matrix vector operation suffices. Accordingly, a reduction of the computational effort from  $O(n^3)$  to  $O(n^2)$  occurs. There is, however, a major drawback: the convergence is no longer quadratic (as it is in Newton's, Brown or Brent method); it is only superlinear:

$$\lim_{k \rightarrow \infty} \frac{\|x^{(k+1)} - \alpha\|}{\|x^{(k)} - \alpha\|} = 0. \quad (7.11.4)$$

Broyden's chooses the vectors  $u^{(k)}$  and  $v^{(k)}$  using the principle of secant approximation. For the scalar case, the approximation  $a_k \approx f'(x^{(k)})$  is uniquely defined by

$$a_{k+1}(x^{(k+1)} - x^{(k)}) = f(x^{(k+1)}) - f(x^{(k)}).$$

However, for  $n > 1$ , the approximation

$$A_{k+1}(x^{(k+1)} - x^{(k)}) = F(x^{(k+1)}) - F(x^{(k)}) \quad (7.11.5)$$

(called quasi-Newton equation) is no longer uniquely defined; any other matrix of the form

$$\bar{A}_{k+1} := A_{k+1} + pq^T,$$

where  $p, q \in \mathbb{R}^n$  and  $q^T(x^{(k+1)} - x^{(k)})$  verifies the equations (7.11.5). On the other hand,

$$y_k := F(x^{(k)}) - F(x^{(k-1)}) \text{ and } s_k := x^{(k)} - x^{(k-1)}$$

only contain information about the partial derivative of  $F$  in the direction of  $s_k$ , not about the partial derivative in directions orthogonal to  $s_k$ . On this direction, the effect of  $A_{k+1}$  and  $A_k$  should be the same

$$A_{k+1}q = A_k q, \quad \forall q \in \{v : v \neq 0, v^T s_k = 0\}. \quad (7.11.6)$$

Starting from an initial approximation  $A_0 \approx F'(x^{(0)})$  (the differential quotient given by (7.11.1) could be such an example), one generates the sequence  $A_1, A_2, \dots$ , uniquely determined by formulas (7.11.5) and (7.11.6) (Broyden [9], Dennis and Moré [24]).

For the corresponding sequence  $B_0 = A_0^{-1} \approx [F(x^{(0)})]^{-1}$ ,  $B_1, B_2, \dots$ , the Sherman-Morrison formula can be used to obtain the recursion

$$B_{k+1} := B_k + \frac{(s_{k+1} - B_k y_{k+1})s_{k+1}^T B_k}{s_{k+1}^T B_k y_{k+1}}, \quad k = 0, 1, 2, \dots$$

which requires only matrix-vector multiplication operations and thus only  $O(n^2)$  computational work. With the matrices  $B_k$  one can define the Broyden's method by the iteration

$$x^{(k+1)} := x^{(k)} - B_k F(x^{(k)}), \quad k = 0, 1, 2, \dots$$

This method converges superlinearly in terms of (7.11.4), if the update vectors  $s_k$  converge (as  $k \rightarrow \infty$ ) to the update vectors of the Newton's. This is a good illustration of the importance of local linearization principle for the solution of nonlinear equation.

MATLAB Source 7.5 gives an implementation of Broyden method.

**MATLAB Source 7.5** Broyden method for nonlinear systems

---

```

function [z,ni]=Broyden1(f,fd,x0,ea,er,nmax)
%Broyden1 - Broyden method for nonlinear systems
%call [z,ni]=Broyden1(f,x0,ea,er,nmax)
%Input
%f - function
%fd - derivative
%x0 - starting approximation
%ea - absolute error
%er - relative error
%nmax - maximum number of iterations
%Output
%z - approximate solution
%ni - actual no. of iterations

if nargin < 6, nmax=50; end
if nargin < 5, er=0; end
if nargin < 4, ea=1e-3; end
x=zeros(length(x0),nmax+1);
F=x;
x(:,1)=x0(:);
F(:,1)=f(x(:,1));
B=inv(fd(x));
x(:,2)=x(:,1)+B*F(:,1);
for k=2:nmax
    F(:,k)=f(x(:,k));
    y=F(:,k)-F(:,k-1); s=x(:,k)-x(:,k-1);
    B=B+((s-B*y)*s'*B)/(s'*B*y);
    x(:,k+1)=x(:,k)-B*F(:,k);
    if norm(x(:,k+1)-x(:,k),inf)<ea+er*norm(x(:,k+1),inf)
        z=x(:,k+1); %success
        ni=k;
        return
    end
end
error('maximum iteration number exceeded')

```

---

**Example 7.11.1.** The nonlinear system:

$$\begin{aligned}3x_1 - \cos(x_1 x_2) - \frac{1}{2} &= 0, \\x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06 &= 0, \\e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3} &= 0\end{aligned}$$

was solved in Example 7.10.2 by using Newton method. By applying Broyden's method with the same function and jacobian as in Example 7.10.2 and the starting values  $x^{(0)} = [0.1, 0.1, -0.1]^T$  one obtains the results in Table 7.2. The solution in MATLAB is

```
>> [z, ni] = Broyden1(@fs3, @fs3d, x0, 1e-9);  
The desired accuracy,  $10^{-9}$ , is achieved after 8 iterations.  $\diamond$ 
```

$k$	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$	$\ x^{(k)} - x^{(k-1)}\ _\infty$
0	0.1000000000	0.1000000000	-0.1000000000	—
1	-0.2998696729	0.1805331515	0.3215204719	0.42152
2	0.5005221618	0.0308676159	-0.5197695490	0.84129
3	0.4999648231	0.0130913099	-0.5229213211	0.0177763
4	0.5000140840	0.0018065260	-0.5235454326	0.0112848
5	0.5000009766	0.0001164840	-0.5235956998	0.00169004
6	0.5000000094	0.0000011019	-0.5235987460	0.000115382
7	0.5000000000	0.0000000006	-0.5235987756	1.10133e-006
8	0.5000000000	0.0000000000	-0.5235987756	5.69232e-010
9	0.5000000000	-0.0000000000	-0.5235987756	4.29834e-013

Table 7.2: Results in Example 7.11.1

## 7.12 Nonlinear Equations in MATLAB

MATLAB has few functions to find the zeros of a univariate function. We have already seen that if the function is polynomial, then `roots(p)` returns all the roots of `p`, where `p` is the coefficient vector decreasingly ordered on the power of variable.

`fzero` function finds a root of a univariate function. The algorithm used by `fzero` is a combination of several methods: bisection, secant and inverse quadratic interpolation [66]. The simplest invocation of `fzero` is `x = fzero(fun, x0)`, with `x0` a scalar, which attempts to find a zero of `fun` near `x0`. For example,

```
>> fzero(@(x) cos(x)-x, 0)  
ans =  
0.7391
```

The convergence tolerance and the display of output in `fzero` are controlled by a third input argument, the structure `options`, which is best defined using the `optimset` function. Four of the fields of the `options` structure are used:

- `Display` specifies the level of reporting, with values `off` for no output, `iter` for output at each iteration, `final` for just the final output, and `notify` to display the result only if the method does not converge (default);
- `TolX` is a convergence tolerance;
- `FunValCheck` determines whether function values are checked for complex or NaN values;
- `OutputFcn` specify a user defined function that is called at each iteration.

For the previous example, using `Display` with value `final`, we obtain:

```
>> fzero(@(x) cos(x)-x, 0, optimset('Display','final'))
Zero found in the interval [-0.905097, 0.905097]
ans =
    0.7391
```

The input argument `x0` can be a 2-vector; at its components the function must have values of opposite sign. Such an argument is useful when the function has singularities.

We shall set `Display` to `final` using

```
os=optimset('Display','final');
```

Consider the example (message is omitted):

```
>> [x,fval]=fzero('tan(x)-x', 1, os)
...
x =
    1.5708
fval =
-1.2093e+015
```

The second output argument is the function value at `x`, the purported singularity. Since  $f(x) = \tan x - x$  has a singularity at  $\pi/2$  (see Figure 7.5), we can give pass to `fzero` a starting interval that encloses a zero but not a singularity:

```
>> [x,fval]=fzero('tan(x)-x', [-1,1], os)
Zero found in the interval: [-1, 1].
x =
    0
fval =
    0
```

We can pass additional parameters `p1, p2, ...` to `f` function using calls such

```
x = fzero(f, x0, options, p1, p2, ...)
```

but it is more convenient to use anonymous functions.

MATLAB has no function for the solution a nonlinear system. However, an attempt at solving such a system could be made by minimizing the sum of squares of the residual. The Optimization Toolbox contains a nonlinear equation solver.

Function `fminsearch` searches for a local minimum of a real function of  $n$  real variables. A possible call is `x=fminsearch(f, x0, options)`. The fields in `options` structure are those supported by `fzero` plus `MaxFunEvals` (the maximum number of function evaluations allowed), `MaxIter` (the maximum number of iterations allowed), `TolFun` (a termination tolerance on the function value). Both `TolX` and `TolFun` default to  $1e-4$ .

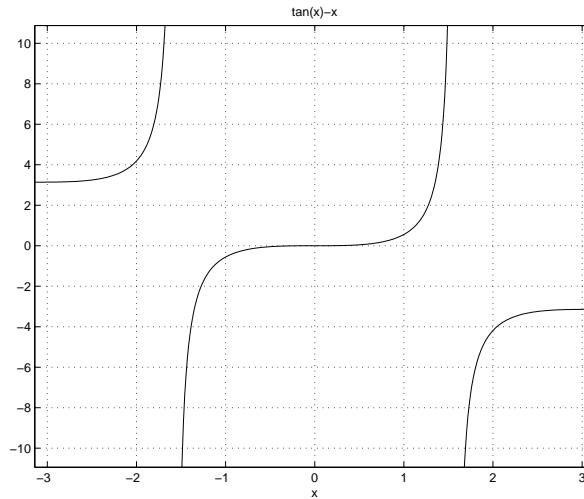


Figure 7.5: Singularity of  $f(x) = \tan x - x$ , obtained with `ezplot('tan(x)-x', [-pi, pi]), grid`

**Example 7.12.1.** The nonlinear system in Example 7.10.2 and 7.11.1, that is

$$\begin{aligned}f_1(x_1, x_2, x_3) &:= 3x_1 - \cos(x_1 x_2) - \frac{1}{2} = 0, \\f_2(x_1, x_2, x_3) &:= x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06 = 0, \\f_3(x_1, x_2, x_3) &:= e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3} = 0\end{aligned}$$

could be solved by trying minimization of the sum of squares of left-hand sides:

$$F(x_1, x_2, x_3) = [f_1(x_1, x_2, x_3)]^2 + [f_2(x_1, x_2, x_3)]^2 + [f_3(x_1, x_2, x_3)]^2.$$

The function to be minimized is given in M-file `fminob.m`:

```
function y = fminob(x)
y=(3*x(1)-cos(x(2)*x(3))-1/2)^2+(x(1)^2-81*(x(2)+0.1)^2+...
sin(x(3))+1.06)^2+(exp(-x(1)*x(2))+20*x(3)+(10*pi-3)/3)^2;
```

We chose the starting vector  $x^{(0)} = [0.5, 0.5, 0.5]^T$  and the tolerance  $10^{-9}$  for  $x$  and for the function. We give below MATLAB commands and their result

```
>> os=optimset('Display','final','TolX',1e-9,'TolFun',1e-9);
>> [xm,fval]=fminsearch(@fminob,[0.5,0.5,0.5]',os)
Optimization terminated:
the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-009 and F(X) satisfies the
convergence criteria using OPTIONS.TolFun of 1.000000e-009
xm =
0.49999999959246
```

```

0.00000000001815
-0.52359877559440
fval =
1.987081116616629e-018

```

Because of complicated objective function, the running time is larger than that for Newton's or Broyden's method and the precision is worse. The computed approximation can be used as starting vector for Newton's method.  $\diamond$

Function `fminsearch` is based on the Nelder-Mead simplex algorithm, a direct search method that uses function values but not derivatives. The method can be very slow to converge, or may fail to converge to a local minimum. However, it has the advantage of being insensitive to discontinuities in the function. More sophisticated minimization functions can be found in the Optimization Toolbox.

For the minimization of a univariate function MATLAB provides the `fminbnd` function. Example

```

>> [x, fval] = fminbnd(@(x) sin(x)-cos(x), -pi, pi)
x =
-0.7854
fval =
-1.4142

```

## 7.13 Applications

### 7.13.1 Analysis of the state equation of a real gas

The mathematical model and implementation hint for this application is from [72]. For a mole of a perfect gas, the state equation  $Pv = RT$  establishes a relation between the pressure  $P$  of the gas (in Pascals [ $Pa$ ]), the specific volume  $v$  (in cubic meters per kilogram [ $m^3 K g^{-1}$ ]) and its temperature  $T$  (in Kelvin [ $K$ ]),  $R$  being the universal gas constant, expressed in [ $J K g^{-1} K^{-1}$ ] (joules per kilogram per Kelvin).

For a real gas, the deviation from the state equation of perfect gases is due to van der Waals and takes into account the intermolecular interaction and the space occupied by molecules of finite size.

Denoting by  $\alpha$  and  $\beta$  the gas constants according to the van der Waals model, in order to determine the specific volume  $v$  of the gas, once  $P$  and  $T$  are known, we must solve the nonlinear equation

$$f(v) = (P + \alpha/v^2)(v - \beta) - RT = 0. \quad (7.13.1)$$

For this purpose we shall apply Newton's method in the case of carbon dioxide ( $CO_2$ ), at the pressure of  $P = 10[atm]$  (that is  $1013250[Pa]$ ) and at the temperature of  $T = 300[K]$ .

The constants are  $\alpha = 188.33[Pam^6 Kg^{-2}]$  and  $\beta = 9.77 \cdot 10^{-4}[m^3 Kg^{-1}]$ . Assuming the gas is perfect, the computed solution is  $v \approx 0.056[m^3 Kg^{-1}]$ .

To approximate the solution we shall use Newton's method and MATLAB function `fzero`. Here is a MATLAB script that plots  $f$  and solve approximately the equation.

```

%set parameters
alpha=188.33;
beta_p=9.77e-4;
P=1013250;
R=8.3144721515/0.044;

```

```
%R=11.0163766063776
T=300;
%plot f
f = @(v) (P+alpha./v.^2).* (v-beta_p)-R*T;
fd= @(v) P+alpha./v.^2-2*(v-beta_p)*alpha./v.^3;
v=linspace(1e-3,0.1,5000);
plot([0,0.1],[0,0],'k-',v,f(v),'k-')
%Newton method
for v0=[1e-4,1e-3,1e-2,1e-1]
    [vv,nit]=Newton(f,fd,v0,0,eps,200);
    v0, vv, nit
end
%MATLAB fzero
opt=optimset('TolX',eps,'Display','iter');
[xv,fv]=fzero(f,[1e-4,0.1],opt)
```

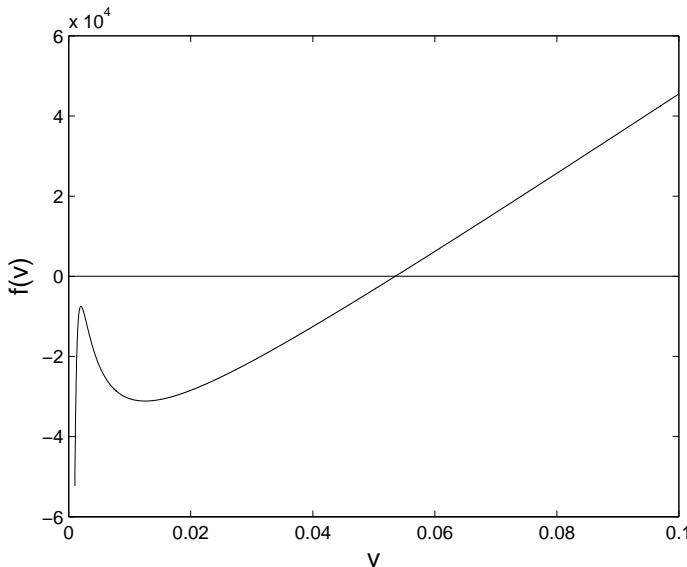


Figure 7.6: Graph of the function  $f$  in (7.13.1)

Newton's method for a relative error equal to machine epsilon and starting values  $v_0$  given in the table requires a number of iteration denoted by  $Nit$

$v_0$	$Nit$	$v_0$	$Nit$	$v_0$	$Nit$	$v_0$	$Nit$
$10^{-4}$	30	$10^{-3}$	36	$10^{-2}$	7	$10^{-1}$	5

The computed approximation of  $v$  is  $v^* = 0.053515529144585$ . To analyze the causes of the strong dependence of  $Nit$  on  $v_0$  let examine the derivative  $f'(v) = P - \alpha v^{-2} + 2\alpha\beta v^{-3}$ . For  $v > 0$ ,  $f'(v) = 0$  at  $v_M \approx 1.99 \cdot 10^{-3} [m^3 Kg^{-1}]$  (a relative maximum) and at  $v_m \approx 1.25 \cdot 10^{-2} [m^3 Kg^{-1}]$

(a relative minimum) (see the graph of  $f$  in Figure 7.6). The choice of  $v_0$  in the interval  $(0, v_m)$  leads to a slow convergence.

The results for each iteration of `fzero` are

Func-count	x	f(x)	Procedure
2	0.1	45510.4	initial
3	0.0997264	45238.3	interpolation
4	0.0543725	814.9	interpolation
5	0.0543725	814.9	bisection
6	0.0534599	-52.8251	interpolation
7	0.0535155	-0.0545071	interpolation
8	0.0535155	6.49088e-008	interpolation
9	0.0535155	0	interpolation

```
Zero found in the interval [0.0001, 0.1]
xv =
    0.053515529144585
fv =
    0
```

### 7.13.2 Nonlinear heat transfer in a wire

Consider a thin wire of length  $L$  and radius  $r$ . Let the ends of the wire have a fixed temperature of 900 and let the surrounding region be  $u_{sur} = 300$ . Suppose the surface of the wire is being cooled via radiation.

The continuous model for the heat transfer is [102]

$$\begin{aligned} -(Ku_x)_x &= c(u_{sur}^4 - u^4) \\ u(0) &= u(L) = 900, \end{aligned}$$

where  $c = 2\varepsilon\sigma/r$ ,  $\varepsilon$  is the emissivity of the surface and  $\sigma$  is the Stefan-Boltzmann constant. For simplicity assume  $K$  is a constant and will be incorporated into  $c$ . Consider the nonlinear differential equation  $-u_{xx} = f(u)$ . Applying the finite difference discretization, as in Section 4.9.1, for  $h = L/(n+1)$  and  $u_i \approx u(ih)$  with  $u_0 = u_{n+1} = 900$  we obtain

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f(u_i), \quad i = 1, \dots, n.$$

Thus, this discrete model has  $n$  unknowns,  $u_i$ , and  $n$  equations

$$F_i(u) \equiv h^2 f(u_i) + u_{i-1} - 2u_i + u_{i+1} = 0.$$

The Jacobian matrix is easily computed and must be tridiagonal because each  $F_i(u)$  only depends on  $u_{i-1}$ ,  $u_i$  and  $u_{i+1}$

$$F'_i(u) = \begin{bmatrix} h^2 f(u_1) - 2 & 1 & & & \\ 1 & h^2 f(u_2) - 2 & \ddots & & \\ & \ddots & \ddots & \ddots & 1 \\ & & & 1 & h^2 f(u_n) - 2 \end{bmatrix}.$$

For the Stefan cooling model where the absolute temperature is positive  $f'(u) < 0$ . Thus, the Jacobian matrix is strictly diagonally dominant and must be nonsingular so that the solve step can be done in Newton's method. The MATLAB function HeatTransfer contains the code for the solution based on Newton's method.

```
function [x,y]=HeatTransfer(L,c,n)

h = L/(n+1);
x = (0:n+1)*h;
f = @(u) c*(300^4 - u.^4);
df = @(u) -4*c*u.^3;
u0 = 900*ones(n,1);
[z,ni] = Newton(@F,@Jac,u0,1e-4,0,150);
y=[900;z;900];

function y=F(u)
    ux=[900;u;900];
    y=h^2*f(u)+ux(1:end-2)-2*ux(2:end-1)+ux(3:end);
end
function y=Jac(u)
    y=diag(h^2*df(u)-2)+diag(ones(n-1,1),1)+diag(ones(n-1,1),-1);
end
end
```

We have experimented with  $c = 10^{-8}, 10^{-7}$  and  $10^{-6}$ . The curves in Figure 7.7 indicate the larger the  $c$  the more the cooling, that is, the lower the temperature. We obtained the figure with the code

```
%THT - test heat transfer
c = [1e-8, 1e-7, 1e-6];
n=30;
x=zeros(n+2,3);
y=x;
L=1;

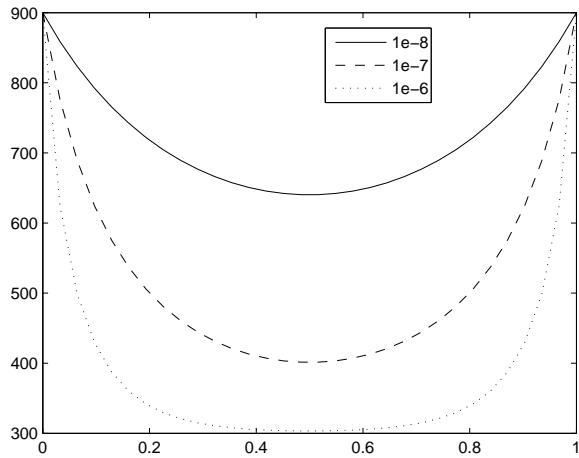
for k=1:length(c)
    [x(:,k),y(:,k)]=HeatTransfer(L,c(k),n);
end
plot(x(:,1),y(:,1),'k-',x(:,2),y(:,2),'k--',...
      x(:,3),y(:,3),'k:');
legend('1e-8','1e-7','1e-6',0)
```

## Problems

**Problem 7.1.** Find the first 10 positive values,  $x$ , for which  $x = \operatorname{tg} x$ .

**Problem 7.2.** Investigate the behavior of Newton and secant method for the function

$$f(x) = \operatorname{sign}(x-a)\sqrt{|x-a|}.$$

Figure 7.7: Temperatures for Variable  $c$ 

**Problem 7.3 (Adapted after[66]).** Consider the polynomial

$$x^3 - 2x - 5.$$

Wallis uses this example to present Newton method in front of French Academy. It has a real root within the interval  $(2, 3)$  and a pair of complex conjugated roots.

- (a) Use Maple or Symbolic Math toolbox to compute the roots. The results are ugly. Convert them to numerical values.
- (b) Determine all the roots using MATLAB function `roots`.
- (c) Find the real root with MATLAB function `fzero`.
- (d) Find all the roots using Newton method (for complex roots use complex starting values).
- (e) Can you use bisection or false position method to find a complex root? Why or why not?

**Problem 7.4.** Solve the systems numerically

$$\begin{aligned} f_1(x, y) &= 1 - 4x + 2x^2 - 2y^3 = 0 \\ f_2(x, y) &= -4 + x^4 + 4y + 4y^4 = 0, \end{aligned}$$

$$\begin{aligned} f_1(x_1, x_2) &= x_1^2 - x_2 + 0.25 = 0 \\ f_2(x_1, x_2) &= -x_1 + x_2^2 + 0.25 = 0, \end{aligned}$$

$$\begin{aligned} f_1(x_1, x_2) &= 2x_1 + x_2 - x_1x_2/2 - 2 = 0 \\ f_2(x_1, x_2) &= x_1 + 2x_2^2 - \cos(x_2)/2 - \frac{3}{2} = 0. \end{aligned}$$

**Problem 7.5.** Solve the system numerically

$$\begin{aligned}9x^2 + 36y^2 + 4z^2 - 36 &= 0, \\x^2 - 2y^2 - 20z &= 0, \\x^2 - y^2 + z^2 &= 0\end{aligned}$$

*Hint.* There are four solutions. Good starting values:  $[\pm 1, \pm 1, 0]^T$ .

**Problem 7.6.** Consider the system, inspired from chemical industry

$$f_i := \beta a_i^2 + a_i - a_{i-1} = 0.$$

The system has  $n$  equations and  $n + 2$  unknowns. We shall set  $a_0 = 5$ ,  $\beta = a_n = 0.5$  mol/liter. Solve the system for  $n = 10$  and starting value  $x = [1; -0.1; 0.1]'$ .



# CHAPTER 8

---

## Eigenvalues and Eigenvectors

---

In the following we study the problem of determining eigenvalues (and eigenvectors) of a square matrix  $A \in \mathbb{R}^{n \times n}$ , that is, to find the numbers  $\lambda \in \mathbb{C}$  and the vectors  $x \in \mathbb{C}^n$  such that

$$Ax = \lambda x. \quad (8.0.1)$$

**Definition 8.0.1.** A number  $\lambda \in \mathbb{C}$  is called an eigenvalue of the matrix  $A \in \mathbb{R}^{n \times n}$ , if there is a vector  $x \in \mathbb{C}^n \setminus \{0\}$  called an eigenvector (or a right eigenvector) such that  $Ax = \lambda x$ .

**Remark 8.0.2.** 1. The requirement  $x \neq 0$  is important, since the null vector is an eigenvector corresponding to any eigenvalue.  
2. Even if  $A$  is real, some of its eigenvalues may be complex. For real matrices, these occur always in conjugated pairs.  $\diamond$

### 8.1 Eigenvalues and Polynomial Roots

Any eigenvalue problem could be reduced to a problem of finding zeros of a polynomial: the eigenvalues of a matrix  $A \in \mathbb{R}^{n \times n}$  are the roots of a *characteristic polynomial*

$$p_A \lambda = \det(A - \lambda I), \quad \lambda \in \mathbb{C}$$

since the above determinant is zero if and only if the system  $(A - \lambda I)x = 0$  has a nontrivial solution, that is,  $\lambda$  is an eigenvalue.

A naive method for the solution of eigenproblems is the computation of characteristic polynomial and then the computation of its roots. But the computation of a determinant is, in general, a complex and unstable problem, so matrix transformation is more appropriate. Conversely, the problem of computing

polynomial roots can be formulated as an eigenvalue problem. Let  $p \in \mathbb{P}_n$  a polynomial with real coefficients, that can be written (by mean of its roots  $z_1, \dots, z_n$ , which could be complex) as

$$p(x) = a_n x^n + \dots + a_0 = a_n(x - z_1) \dots (x - z_n), \quad a_n \in \mathbb{R}, \quad a_n \neq 0.$$

In the vector space  $\mathbb{P}_{n-1}$  “modulo  $p$  multiplication”

$$\mathbb{P}_{n-1} \ni q \mapsto r \quad xq(x) = \alpha p(x) + r(x), \quad r \in \mathbb{P}_n \quad (8.1.1)$$

is a *linear transform*, and since

$$x^n = \frac{1}{a_n} p(x) - \sum_{j=0}^{n-1} \frac{a_j}{a_n} x^j, \quad x \in \mathbb{R},$$

we shall represent  $p$  in basis  $1, x, \dots, x^{n-1}$  by mean of so-called Frobenius's *companion matrix* (of size  $n \times n$ )

$$M = \begin{bmatrix} 0 & & & -\frac{a_0}{a_n} \\ 1 & 0 & & -\frac{a_1}{a_n} \\ \ddots & \ddots & \ddots & \vdots \\ & 1 & 0 & -\frac{a_{n-2}}{a_n} \\ & & 1 & -\frac{a_{n-1}}{a_n} \end{bmatrix}, \quad (8.1.2)$$

Let  $v_j = (v_{jk} : k = \overline{1, n}) \in \mathbb{C}^n$ ,  $j = \overline{1, n}$  chosen so that

$$\ell_j(x) = \frac{p(x)}{x - z_j} = a_n \prod_{k \neq j} (x - z_k) = \sum_{k=1}^n v_{jk} x^{k-1}, \quad j = \overline{1, n},$$

then

$$\sum_{k=1}^n (Mv_j - z_j v_j)_k x^{k-1} = x \ell_j(x) - z_j \ell_j(x) = (x - z_j) \ell_j(x) = p(x) \approx 0,$$

and thus  $Mv_j = z_j v_j$ ,  $j = \overline{1, n}$ .

Hence, eigenvalues of  $M$  are roots of  $p$ .

The Frobenius matrix given by (8.1.2) is only a way (there exists many other) to represent the “multiplication” in (8.1.1); any other basis of  $\mathbb{P}_{n-1}$  provides a matrix  $M$  whose eigenvalues are roots of  $p$ . The only device for polynomial handling is “remainder division”.

## 8.2 Basic Terminology and Schur Decomposition

As the following example shows

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \quad p_A(\lambda) = \lambda^2 + 1 = (\lambda + i)(\lambda - i),$$

a real matrix may have complex eigenvalues. Therefore (at least from theoretical point of view) it is convenient to deal with complex matrices  $A \in \mathbb{C}^{n \times n}$ .

**Definition 8.2.1.** Two matrices,  $A, B \in \mathbb{C}^{n \times n}$  are called similar if there exists a nonsingular matrix  $T \in \mathbb{C}^{n \times n}$ , such that

$$A = TBT^{-1}.$$

**Lemma 8.2.2.** *If  $A, B \in \mathbb{C}^{n \times n}$  are similar, their eigenvalues are the same.*

*Proof.* Let  $\lambda \in \mathbb{C}$  be an eigenvalue of  $A = TBT^{-1}$  and  $x \in \mathbb{C}^n$  its eigenvector. Then

$$B(T^{-1}x) = T^{-1}ATT^{-1}x = T^{-1}Ax = \lambda T^{-1}x$$

and hence,  $\lambda$  is also an eigenvalue of  $B$ .  $\square$

The following important result from linear algebra holds, which we state without proof. (For a proof see [41].)

**Theorem 8.2.3 (Jordan normal form).** *Any matrix  $A \in \mathbb{C}^{n \times n}$  is similar to a matrix*

$$J = \begin{bmatrix} J_1 & & \\ & \ddots & \\ & & J_k \end{bmatrix}, \quad J_\ell = \begin{bmatrix} \lambda_\ell & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_\ell \end{bmatrix} \in \mathbb{C}^{n_\ell \times n_\ell}, \quad \sum_{\ell=1}^k n_\ell = n,$$

called Jordan normal form of  $A$ .

**Definition 8.2.4.** *A matrix is called diagonalizable, if all its Jordan blocks  $J_\ell$  have their dimension equal to one, that is,  $n_\ell = 1$ ,  $\ell = \overline{1, n}$ . A matrix is called nonderogatory if each eigenvalue  $\lambda_\ell$  appears in exactly one Jordan block, on diagonal.*

**Remark 8.2.5.** If a matrix  $A \in \mathbb{R}^{n \times n}$  has  $n$  simple eigenvalues, then it is diagonalizable and also nonderogatory, and suitable for numerical treatment.  $\diamond$

**Theorem 8.2.6 (Schur decomposition).** *For every matrix  $A \in \mathbb{C}^{n \times n}$  there exists a unitary matrix  $U \in \mathbb{C}^{n \times n}$  and an upper triangular matrix*

$$R = \begin{bmatrix} \lambda_1 & * & \dots & * \\ & \ddots & \ddots & \vdots \\ & & \ddots & * \\ & & & \lambda_n \end{bmatrix} \in \mathbb{C}^{n \times n},$$

such that  $A = URU^*$ .

**Remark 8.2.7.** 1. The diagonal elements of  $R$  are eigenvalues of  $A$ . Since  $A$  and  $R$  are similar, they have the same eigenvalues.

2. We have a stronger form of similarity between  $A$  and  $R$ : they are *unitary-similar*.  $\diamond$

*Proof of theorem 8.2.6.* By induction on  $n$ . The case  $n = 1$  is trivial. Suppose the theorem is true for  $n \in \mathbb{N}$  and let  $A \in \mathbb{C}^{(n+1) \times (n+1)}$ . Let  $\lambda \in \mathbb{C}$  be an eigenvalue of  $A$  and  $x \in \mathbb{C}^{n+1}$ ,  $\|x\|_2 = 1$ , the corresponding eigenvector. We take  $u_1 = x$  and we choose  $u_2, \dots, u_{n+1}$  such that  $u_1, \dots, u_{n+1}$  to be an orthonormal basis for  $\mathbb{C}^{n+1}$ , or equivalently, the matrix  $U = [u_1, \dots, u_{n+1}]$  to be unitary. Thus,

$$U^*AUe_1 = U^*Au_1 = U^*Ax = \lambda U^*x = \lambda e_1,$$

that is

$$U^*AU = \begin{bmatrix} \lambda & * \\ 0 & B \end{bmatrix}, \quad B \in \mathbb{C}^{n \times n}.$$

By induction hypothesis, there exists a unitary matrix  $V \in \mathbb{C}^{n \times n}$ , such that  $B = VSV^*$ , where  $S \in \mathbb{C}^{n \times n}$  is an upper triangular matrix. Therefore,

$$A = U \begin{bmatrix} \lambda_1 & * \\ 0 & VSV^* \end{bmatrix} U^* = U \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & V \end{bmatrix}}_{=:U} \underbrace{\begin{bmatrix} \lambda_1 & * \\ 0 & S \end{bmatrix}}_{=:R} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & V^* \end{bmatrix}}_{=:U^*} U^*,$$

which completes the proof.  $\square$

Let us give now two direct consequences of Schur decomposition.

**Corollary 8.2.8.** *To each Hermitian matrix  $A \in \mathbb{C}^{n \times n}$  there corresponds an orthogonal matrix  $U \in \mathbb{C}^{n \times n}$  such that*

$$A = U \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} U^*, \quad \lambda_j \in \mathbb{R}, \quad j = \overline{1, n}.$$

*Proof.* The matrix  $R$  in Theorem 8.2.6 verifies  $R = U^*AU$ . Since

$$R^* = (U^*AU) = U^*A^*U = U^*AU = R,$$

$R$  must be diagonal, and its diagonal elements are real (being Hermitian).  $\square$

In other words, Corollary 8.2.8 guarantees that any Hermitian matrix is unitary-diagonalizable and has a basis which consists of orthonormal eigenvectors. Moreover, all eigenvalues of a Hermitian matrix are real. It is interesting, not only from theoretical point of view, what kind of matrices are unitary-diagonalizable.

**Theorem 8.2.9.** *A matrix  $A \in \mathbb{C}^{n \times n}$  is unitary-diagonalizable, that is, there exists a unitary matrix  $U \in \mathbb{C}^{n \times n}$  such that*

$$A = U \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} U^*, \quad \lambda_j \in \mathbb{R}, \quad j = 1, \dots, n. \quad (8.2.1)$$

*if and only if  $A$  is normal, i.e.*

$$AA^* = A^*A. \quad (8.2.2)$$

*Proof.* We set  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ . By (8.2.1),  $A$  has the form  $A = U\Lambda U^*$ , so

$$AA^* = U\Lambda U^*U\Lambda^*U^* = U|\Lambda|^2U^* \text{ and } A^*A = U^*\Lambda^*U^*U\Lambda U^* = U|\Lambda|^2U^*,$$

that is, (8.2.2). For the converse, we use the Schur decomposition of  $A$  in form  $R = U^*AU$ . Then

$$|\lambda_1|^2 = (R^*R)_{11} = (RR^*)_{11} = |\lambda_1|^2 + \sum_{k=2}^n |r_{1k}|^2,$$

which implies  $r_{12} = \dots = r_{1n} = 0$ . By induction, for  $j = \overline{2, n}$ ,

$$(R^*R)_{jj} = |\lambda_j|^2 + \sum_{k=1}^{j-1} |r_{kj}|^2 = (RR^*)_{jj} = |\lambda_j|^2 + \sum_{k=j+1}^n |r_{jk}|^2,$$

and for this reason  $R$  must be diagonal.  $\square$

A Schur decomposition for real matrices, that is, the so-called *real Schur decomposition* is a little bit more complicated.

**Theorem 8.2.10.** *For any matrix  $A \in \mathbb{R}^{n \times n}$  there exists an orthogonal matrix  $U \in \mathbb{R}^{n \times n}$  such that*

$$A = U \begin{bmatrix} R_1 & * & \dots & * \\ * & \ddots & \ddots & \vdots \\ & * & \ddots & * \\ & & * & R_k \end{bmatrix} U^*, \quad (8.2.3)$$

where either  $R_j \in \mathbb{R}^{1 \times 1}$ , or  $R_j \in \mathbb{R}^{2 \times 2}$ , with two complex conjugated eigenvalues,  $j = \overline{1, k}$ .

A real Schur decomposition transforms  $A$  into an upper Hessenberg matrix

$$U^T A U = \begin{bmatrix} * & \dots & \dots & * \\ * & \ddots & & \vdots \\ & \ddots & \ddots & \vdots \\ & & * & * \end{bmatrix}.$$

*Proof.* If all eigenvalues of  $A$  are real then we proceed the same way as for complex Schur decomposition. Thus, let  $\lambda = \alpha + i\beta$ ,  $\beta \neq 0$ , a complex eigenvalue of  $A$  and  $x + iy$  its eigenvector. Then

$$A(x + iy) = \lambda(x + iy) = (\alpha + i\beta)(x + iy) = (\alpha x - \beta y) + i(\beta x + \alpha y)$$

or in matrix form

$$A \underbrace{\begin{bmatrix} x & y \end{bmatrix}}_{\in \mathbb{R}^{n \times 2}} = \begin{bmatrix} x & y \end{bmatrix} \underbrace{\begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}}_{:= R}.$$

Since  $R = \alpha^2 + \beta^2 > 0$ , ( $\beta \neq 0$ ),  $\text{span}\{x, y\}$  is an  $A$ -invariant bidimensional subspace of  $\mathbb{R}^n$ . Then we choose  $u_1, u_2$  such that they form a basis of this space, completed by  $u_3, \dots, u_n$  such that all these vectors be an orthonormal basis of  $\mathbb{R}^n$  and after a reasoning which is analogous to that of complex case we get

$$U^T A U = \begin{bmatrix} R & * \\ 0 & B \end{bmatrix},$$

and the induction proceeds as for complex Schur decomposition.  $\square$

## 8.3 Vector Iteration

*Vector iteration* (also called *power method*) is the simplest method when an eigenvalue and its eigenvector is needed.

Starting with an arbitrary  $y^{(0)} \in \mathbb{C}^n$  one constructs the sequence  $y^{(k)}$ ,  $k \in \mathbb{N}$  based on the following iteration

$$\begin{aligned} z^{(k)} &= A y^{(k-1)}, \\ y^{(k)} &= \frac{z^{(k)}}{|z^{(k)}|} \frac{|z_{j_*}^{(k)}|}{\|z^{(k)}\|_\infty}, \quad j_* = \min \left\{ 1 \leq j \leq n : |z_j^{(k)}| \geq \left(1 - \frac{1}{k}\right) \|z^{(k)}\|_\infty \right\}. \end{aligned} \quad (8.3.1)$$

Under certain condition, this sequence converges to the eigenvector corresponding to the dominant eigenvalue.

**Proposition 8.3.1.** Let  $A \in \mathbb{C}^{n \times n}$  a diagonalizable matrix whose eigenvalues verify the condition

$$|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|.$$

Then the sequence  $y^{(k)}$ ,  $k \in \mathbb{N}$ , converges to a multiple of the normed eigenvector corresponding to the eigenvalue  $\lambda_1$ , for almost every starting vector  $y^{(0)}$ .

*Proof.* Let  $x_1, \dots, x_n$  be the orthonormal eigenvectors of  $A$  corresponding to the eigenvalues  $\lambda_1, \dots, \lambda_n$  – they exist,  $A$  is diagonalizable. We express  $y^{(0)}$  as

$$y^{(0)} = \sum_{j=1}^n \alpha_j x_j, \quad \alpha_j \in \mathbb{C}, \quad j = \overline{1, n},$$

and we state that

$$A^k y^{(0)} = \sum_{j=1}^n \alpha_j A^k x_j = \sum_{j=1}^n \alpha_j \lambda_j^k x_j = \lambda_1^k \sum_{j=1}^n \alpha_j \left(\frac{\lambda_j}{\lambda_1}\right)^k x_j.$$

Since  $|\lambda_1| > |\lambda_j|$ ,  $j = \overline{2, n}$  this implies

$$\lim_{k \rightarrow \infty} \lambda_1^{-k} A^k y^{(0)} = \alpha_1 x_1 + \lim_{k \rightarrow \infty} \sum_{j=2}^n \alpha_j \left(\frac{\lambda_j}{\lambda_1}\right)^k x_j = \alpha_1 x_1,$$

and also

$$\lim_{k \rightarrow \infty} |\lambda_1|^{-k} \|A^k y^{(0)}\|_\infty = \left\| \sum_{j=1}^n \alpha_j \left(\frac{\lambda_j}{\lambda_1}\right)^k \alpha_j x_j \right\| = |\alpha_1| \|x_1\|_\infty.$$

If  $\alpha_1 = 0$ , and thus  $y^{(0)}$  is in hyperplane

$$x_1^+ = \{x \in \mathbb{C}^n, x^* x_1 = 0\},$$

then both limits are zero, and we cannot derive any conclusion on the convergence of  $y^{(k)}$ ,  $k \in \mathbb{N}$ ; this hyperplane has the measure zero, so in the sequel we shall suppose  $\alpha_1 \neq 0$ .

Equation (8.3.1) implies  $y^{(k)} = \gamma_k A^k y^{(0)}$ ,  $\gamma_k \in \mathbb{C}$  and moreover  $\|y^{(k)}\|_\infty = 1$ , so

$$\lim_{k \rightarrow \infty} |\lambda_1|^k |\gamma_k| = \lim_{k \rightarrow \infty} \frac{1}{|\lambda_1|^{-1} \|A^k y^{(0)}\|} = \frac{1}{|\alpha_1| \|x_1\|_\infty}.$$

Thus

$$y^{(k)} = \gamma_k A^k y^{(0)} = \underbrace{\frac{\gamma_k \lambda_1^k}{|\gamma_k \lambda_1^k|}}_{=: e^{-2\pi i \theta_k}} \underbrace{\frac{\alpha_1 x_1}{|\alpha_1| \|x_1\|_\infty}}_{=: \alpha x_1} + O\left(\frac{|\lambda_2|^k}{|\lambda_1|^k}\right), \quad k \in \mathbb{N}, \quad (8.3.2)$$

where  $\theta_k \in [0, 1]$ . Now, it is the time to use the "strange" relation (8.3.1): let  $j$  be the least subscript such that  $|(\alpha x_1)_j| = \|\alpha x_1\|_\infty$ ; then, by (8.3.2), for a sufficiently large  $k$ , it holds in (8.3.1)  $j^* = j$  too. Therefore it holds

$$\lim_{k \rightarrow \infty} y_j^{(k)} = 1 \Rightarrow \lim_{k \rightarrow \infty} e^{2\pi i \theta_k} = \lim_{k \rightarrow \infty} \frac{y_j^{(k)}}{(\alpha x_1)_j} = \frac{1}{(\alpha x_1)_j}.$$

Substituting this in (8.3.2) we conclude the convergence of  $y^{(k)}$ ,  $k \in \mathbb{N}$ .  $\square$

We could also apply vector iteration to compute all eigenvalues and eigenvectors, provided that the eigenvalues of  $A$  have different modulus. For this purpose we find the largest modulus eigenvalue  $\lambda_1$  of  $A$  and the corresponding eigenvector  $x_1$ , and we proceed to

$$A^{(1)} = A - \lambda_1 x_1 x_1^T.$$

The matrix  $A^{(1)}$  is diagonalizable and has the same orthonormal eigenvectors as  $A$ , excepting that  $x_1$  is the eigenvector corresponding to the eigenvalue 0, and does not play any role for the iteration, provided that the starting vector is not a multiple of  $x_1$ . By applying once more vector iteration to  $A^{(1)}$  one obtains the second largest modulus eigenvalue  $\lambda_2$  and the corresponding eigenvector; the iteration

$$A^{(j)} = A^{(j-1)} - \lambda_j x_j x_j^T, \quad j = \overline{1, n}, \quad A^{(0)} = A$$

computes successively all the eigenvalues and eigenvectors of  $A$ , if its eigenvalues are different in modulus.

**Remark 8.3.2 (Drawbacks of vector iteration).** 1. The method works only if there exists a dominant eigenvector, that is only when there exists a unique eigenvector corresponding to the dominant eigenvalue. For example, the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix},$$

transforms vector  $[x_1 \ x_2]^T$  into the vector  $[x_2 \ x_1]^T$  and the convergence holds only if the starting vector is an eigenvector.

2. The method works well only for “suitable” starting vectors. It sounds gorgeous that all vectors which are not in a certain hyperplane are good, but the things are more complicated. If the dominant eigenvalue of a real matrix is complex and the starting values are real, then the iteration run indefinitely, without finding an eigenvector.
3. We could perform all computation in complex, but this grows seriously the computational complexity (with a factor of two for addition and six for multiplication, respectively).
4. The speed of convergence depends on ratio

$$\frac{|\lambda_2|}{|\lambda_1|} < 1,$$

which may be indefinitely close to 1. If the dominance of dominant eigenvector is not sufficiently emphasized, the convergence is very slow.  $\diamond$

Taking into account the above remarks, we conclude that vector iteration method is not enough good.

## 8.4 QR Method – the Theory

The practical method for eigenproblems is the QR method, due to Francis [29] and Kublanovskaya [53], a unitary extension of Rutishauser’s LR method [75]. We begin with the complex case.

The iterative method is very simple: one starts with  $A^{(0)} = A$  and one computes iteratively, using QR decomposition,

$$A^{(k)} = Q_k R_k, \quad A^{(k+1)} = R_k Q_k, \quad k \in \mathbb{N}_0. \quad (8.4.1)$$

With a bit of luck, or as mathematicians say, under certain hypothesis, this sequence will converge to a matrix whose diagonal elements are the eigenvalues of  $A$ .

**Lemma 8.4.1.** *The matrices  $A^{(k)}$ , built by (8.4.1),  $k \in \mathbb{N}$ , are orthogonal-similar to  $A$  (and, obviously have the same eigenvalues as  $A$ ).*

*Proof.* The following statement holds

$$A^{(k+1)} = Q_k^* Q_k R_k Q_k = Q_k^* A^{(k)} Q_k = \cdots = \underbrace{Q_k^* \dots Q_0^*}_{=:U_k^*} A \underbrace{Q_0 \dots Q_k}_{=:U_k}.$$

□

In order to prove the convergence, we shall interpret QR iteration as a generalization of vector iteration (8.3.1) (without the strange norming process) to vector spaces. For this purpose, we shall write the orthonormal base  $u_1, \dots, u_m \in \mathbb{C}^n$  of a  $m$ -dimensional subspace  $U \subset \mathbb{C}^n$ ,  $m \leq n$ , as column vectors of a unitary matrix  $U \in \mathbb{R}^{n \times m}$  and we shall iterate the subspace (i.e. matrices) over the QR decomposition

$$U_{k+1} R_k = A U_k, \quad k \in \mathbb{N}_0, \quad U_0 \in \mathbb{C}^n. \quad (8.4.2)$$

This implies immediately

$$U_{k+1}(R_k \dots R_0) = A U_k(R_{k-1} \dots R_0) = A^2 U_{k-1}(R_{k-2} \dots R_0) = \dots = A^{k+1} U_0. \quad (8.4.3)$$

If we define, for  $m = n$ ,  $A^{(k)} = U_k^* A U_k$ , then by (8.4.2), the following relation holds

$$\begin{aligned} A^{(k)} &= U_k^* A U_k = U_k^* U_{k+1} R_k \\ A^{(k+1)} &= U_{k+1}^* A U_{k+1} = U_{k+1}^* A U_k U_k^* U_{k+1} \end{aligned}$$

and setting  $Q_k := U_k^* U_{k+1}$ , we obtain the iteration rule (8.4.1). We choose  $U_0 = I$  as starting matrix.

**Definition 8.4.2.** A phase matrix  $\Theta \in \mathbb{C}^{n \times n}$  is a diagonal matrix with form

$$\Theta = \begin{bmatrix} e^{-i\theta_1} & & & \\ & \ddots & & \\ & & e^{-i\theta_n} & \end{bmatrix}, \quad \theta_j \in [0, 2\pi), \quad j = \overline{1, n}.$$

**Proposition 8.4.3.** Suppose  $A \in \mathbb{C}^{n \times n}$  has eigenvalues with distinct moduli,  $|\lambda_1| > |\lambda_2| > \dots > |\lambda_n| > 0$ . If the matrix  $X^{-1}$  in normal Jordan form  $A = X \Lambda X^{-1}$  of  $A$  has a LU decomposition

$$X^{-1} = ST, \quad S = \begin{bmatrix} 1 & & & \\ * & 1 & & \\ \vdots & \ddots & \ddots & \\ * & \dots & * & 1 \end{bmatrix}, \quad T = \begin{bmatrix} * & \dots & * \\ & \ddots & \vdots \\ & & * \end{bmatrix},$$

the there exists phase matrices  $\Theta_k$ ,  $k \in \mathbb{N}_0$ , such that the matrix sequence  $(\Theta_k U_k)$ ,  $k \in \mathbb{N}$  is convergent.

**Remark 8.4.4 (On proposition 8.4.3).** 1. The convergence of the matrix sequence  $(\Theta_k U_k)$  means that if the corresponding orthonormal bases converge to an orthonormal basis of  $\mathbb{C}^n$ , we have also the convergence of the corresponding vector spaces.

2. The existence of a LU decomposition for  $X^{-1}$  introduces no additional constraints: since  $X^{-1}$  is invertible, there exists a permutation  $P$ , such that

$$X^{-1}P^T = (PX)^{-1} = LU$$

and  $PX$  is invertible. This means that the matrix  $\hat{A} = P^TAP$ , which is the result of line and column permutation of  $A$  has the same eigenvalues of  $A$ , fulfill the hypothesis of Proposition 8.4.3.

3. The proof of Proposition 8.4.3 is a modification of proof in [90, pag. 54–56] for the convergence of LR, whose origin can be found in Wilkinson’s <sup>1</sup> book [103]. What is the LR method? It is analogous to QR method, but the QR decomposition of  $A$  is replaced by an LU decomposition,  $A^{(k)} = L_k R_k$ , and then one builds  $A^{(k+1)} = R_k L_k$ . Under certain conditions, this method converges to an upper triangular matrix. ◇

Before the proof of 8.4.3, let us see why the convergence of the sequence  $(U_k)$  implies the convergence of QR method. Namely, if we have  $\|U_{k+1} - U_k\| \leq \varepsilon$  or equivalently

$$U_{k+1} = U_k + E, \quad \|E\|_2 \leq \varepsilon,$$

then

$$Q_k = U_{k+1}^* U_k = (U_k + E)^* U_k = I + E^* U_k = I + F, \quad \|F\|_2 \leq \|E\|_2 \underbrace{\|U_k\|_2}_{=1} \leq \varepsilon,$$

and simultaneously

$$A^{(k+1)} = R_k Q_k = R_k(I + F) = R_k + G, \quad \|G\|_2 \leq \varepsilon \|R_k\|_2,$$

hence  $A^{(k)}$ ,  $k \in \mathbb{N}$ , converges also to an upper triangular matrix, only if the norms of  $R_k$ ,  $k \in \mathbb{N}$  are uniformly bounded. This is the case, since

$$\|R_k\|_2 = \|Q_k^* A^{(k)}\|_2 = \|A^{(k)}\|_2 = \|Q_{k-1}^* \dots Q_0^* A Q_0 \dots Q_{k-1}\| = \|A\|_2.$$

We need also an auxiliary result on the “uniqueness” of QR decomposition.

**Lemma 8.4.5.** *Let  $U, V \in \mathbb{C}^{n \times n}$  be unitary matrices and  $R, S \in \mathbb{C}^{n \times n}$  be invertible upper triangular matrices. Then  $UR = VS$  if and only if there exists a phase matrix*

$$\Theta = \begin{bmatrix} e^{-i\theta_1} & & & \\ & \ddots & & \\ & & e^{-i\theta_n} & \end{bmatrix}, \quad \theta_j \in [0, 2\pi), j = \overline{1, n},$$

such that  $U = V\Theta^*$ ,  $R = \Theta S$ .



1

James Hardy Wilkinson (1919-1986), English mathematician. Contribution to numerical analysis, numerical linear algebra and computer science. He received many awards for his outstanding work. He was elected a Fellow of the Royal Society in 1969. He received the A. M. Turing award from the Association of Computing Machinery and the J. von Neumann award from the Society for Industrial and Applied Mathematics both in 1970. Beside the large numbers of papers on his theoretical work on numerical analysis, Wilkinson developed computer software, working on the production of libraries of numerical routines. The NAG (Numerical Algorithms Group) began work in 1970 and much of the linear algebra routines were due to Wilkinson.

*Proof.* Since  $UR = V\Theta^*\Theta S = VS$ , the sufficiency is trivial. For the necessity, from  $UR = VS$  it follows that  $V^*U = SR^{-1}$  must be an upper triangular matrix such that  $(V^*U)^* = U^*V = RS^{-1}$ . Hence  $\Theta = V^*U$  is a unitary diagonal matrix and it holds  $U = VV^*U = V\Theta$ .  $\square$

*Proof of Proposition 8.4.3.* Let  $A = X\Lambda X^{-1}$  be the Jordan normal form of  $A$ , where  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ . For  $U_0 = I$  and  $k \in \mathbb{N}_0$

$$U_k \left( \prod_{j=k-1}^0 R_j \right) = (X^{-1}\Lambda X)^k = X\Lambda^k X^{-1} = X\Lambda^k ST = X \underbrace{(\Lambda^k S \Lambda^{-k})}_{=:L_k} \Lambda^k T,$$

where  $L_k$  is a lower triangular matrix with entries

$$(L_k)_{jm} = \left( \frac{\lambda_j}{\lambda_m} \right)^k, \quad 1 \leq m \leq j \leq n \quad (8.4.4)$$

such that for  $k \in \mathbb{N}$

$$|L_k - I| \leq \left( \max_{1 \leq m < j \leq n} |s_{jm}| \right) \left( \max_{1 \leq m < j \leq n} \left| \frac{\lambda_j}{\lambda_m} \right| \right)^k \begin{bmatrix} 0 & & & \\ 1 & \ddots & & \\ \vdots & \ddots & \ddots & \\ 1 & \dots & 1 & 0 \end{bmatrix}, \quad (k \in \mathbb{N}). \quad (8.4.5)$$

Let  $\widehat{U}_k \widehat{R}_k = X L_k$  be the QR decomposition of  $X L_k$ , that, due to (8.4.5) and Lemma 8.4.5 converges, up to a phase matrix, to a QR decomposition  $X = UR$  of  $X$ . Now we apply Lemma 8.4.5 to the identity

$$U_k \left( \prod_{j=k-1}^0 R_j \right) \widehat{Q}_k \widehat{R}_k \Lambda^k T;$$

there exist phase matrices  $\Theta_k$ , such that

$$U_k = \widehat{Q}_k \Theta_k^* \text{ and } \left( \prod_{j=k-1}^0 R_j \right) = \Theta_k \widehat{R}_k \Lambda^k T,$$

hence there exist phase matrices  $\widehat{\Theta}_k$ , such that  $U_k \widehat{\Theta}_k \rightarrow U$ , when  $k \rightarrow \infty$ .  $\square$

Let us examine shortly the ‘‘error term’’ in (8.4.4), whose sub-diagonal entries verifies

$$|L_k|_{jm} \leq \left( \frac{|\lambda_j|}{|\lambda_m|} \right) |s_{jm}|, \quad 1 \leq m < j \leq n.$$

Therefore, it holds

*The farther is the sub-diagonal element to the diagonal, the faster is the convergence of that element to zero.*

## 8.5 QR Method – the Practice

### 8.5.1 Classical QR method

We have seen that QR method generates a matrix sequence  $A^{(k)}$  that under certain conditions converges to an upper triangular matrix with eigenvalues of  $A$  on diagonal. We may apply this method to real matrices.

**Example 8.5.1.** Let

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 2 & 1 \end{bmatrix}.$$

Its eigenvalues are

$$\lambda_1 \approx 4.56155, \quad \lambda_2 = -1, \quad \lambda_3 \approx 0.43845.$$

Using a rough MATLAB implementation of QR method we obtain the values in Table 8.1 for the subdiagonal entries. Note that after  $k$  iterative steps, the entries  $a_{m\ell}^{(k)}$ ,  $\ell < m$ , approach to 0 like  $|\lambda_\ell/\lambda_k|$

#iterations	$a_{21}$	$a_{31}$	$a_{32}$
10	6.64251e-007	-2.26011e-009	0.00339953
20	1.70342e-013	-1.52207e-019	8.9354e-007
30	4.36711e-020	-1.02443e-029	2.34578e-010
40	1.11961e-026	-6.89489e-040	6.15829e-014

Table 8.1: Results for Example 8.5.1

do.  $\diamond$

**Example 8.5.2.** The matrix

$$\begin{bmatrix} 1 & 5 & 7 \\ 3 & 0 & 6 \\ 4 & 3 & 1 \end{bmatrix}$$

has the eigenvalues

$$\lambda_1 \approx 9.7407, \quad \lambda_2 \approx -3.8703 + 0.6480i, \quad \lambda_3 \approx -3.8703 - 0.6480i.$$

In this case, QR method does not converge to an upper triangular matrix. After 100 iterations we obtain the matrix

$$A^{(100)} \approx \begin{bmatrix} 9.7407 & -4.3355 & 0.94726 \\ 8.552e-039 & -4.2645 & 0.7236 \\ 3.3746e-039 & -0.79491 & -3.4762 \end{bmatrix},$$

that correctly provides the real eigenvalue. Additionally, the lower right  $2 \times 2$  matrix provide the complex eigenvalues  $-3.8703 \pm 0.6480i$ .  $\diamond$

The second example leads us to the following strategy: if the sub-diagonal entries do not disappear, it is recommendable to examine the corresponding  $2 \times 2$  matrix.

**Definition 8.5.3.** If  $A \in \mathbb{R}^{n \times n}$  has the QR decomposition  $A = QR$ , then a RQ transformation of  $A$  is defined by  $A_* = RQ$ .

What problems appear in a practical usage of QR method? Since the complexity of QR decomposition is  $\Theta(n^3)$ , it is not advisable to use a method based on such an iterative step. In order to avoid the problem, we convert the initial matrices into a matrix whose QR decomposition could be computed faster. Such examples are upper Hessenberg matrices, whose QR decomposition could be computed using  $n$  Givens rotations, a total of  $O(n^2)$  flops: since only entries  $h_{j-1,j}$ ,  $j = \overline{2, n}$  must be eliminated, we shall find the angles  $\phi_2, \dots, \phi_n$ , such that

$$G(n-1, n; \phi_n) \dots G(1, 2; \phi_2)H = R$$

and it holds

$$H_* = RG^T(1, 2; \phi_2) \dots G^T(n-1, n; \phi_n). \quad (8.5.1)$$

This is the idea implemented in MATLAB Source 8.1.

---

### MATLAB Source 8.1 The RQ transform of a Hessenberg matrix

---

```
function HH=HessenRQ(H)
%HESSENRO - computes RQ transform of a Hessenberg matrix
%using Givens rotations
%input H - Hessenberg matrix
%output HH - RQ transform of H

[m,n]=size(H);
Q=eye(m,n);

for k=2:n
    a=H(k-1:k,k-1);
    an=sqrt(a'*a); %Euclidean norm
    c=sign(a(2))*abs(a(1))/an; %sine
    s=sign(a(1))*abs(a(2))/an; %cosine
    Jm=eye(n);
    Jm(k-1,k-1)=c; Jm(k,k)=c;
    Jm(k-1,k)=s; Jm(k,k-1)=-s;
    H=Jm*H;
    Q=Q*Jm';
end
HH=H*Q;
```

---

**Lemma 8.5.4.** *If  $H \in \mathbb{R}^{n \times n}$  is an upper Hessenberg matrix, the matrix  $H_*$  is upper Hessenberg, too.*

*Proof.* The conclusion is a direct consequence of (8.5.1) representation. Right multiplication by a Givens matrix,  $G^T(j, j+1, \phi_{j+1})$ ,  $j = \overline{1, n-1}$  means a combination of  $j$ th and  $(j+1)$ th columns that creates nonzero values only in the first sub-diagonal –  $R$  is upper triangular.  $\square$

Let see how to convert the initial matrix to a Hessenberg form. For this purpose we shall use (for variation) Householder transformations. Let us suppose we have already found a matrix  $Q_k$ , such that

the first  $k$  columns of the transformed matrix are already in Hessenberg form, that is,

$$Q_k A Q_k^T = \left[ \begin{array}{ccc|cc|ccc} * & \dots & * & * & & * & \dots & * \\ * & \dots & * & * & & * & \dots & * \\ \ddots & \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ & * & * & * & \dots & * & \dots & * \\ \hline & & & a_1^{(k)} & & * & \dots & * \\ & & & \vdots & & \vdots & \ddots & \vdots \\ & & & a_{n-k-1}^{(k)} & & * & \dots & * \end{array} \right].$$

Then we determine  $\hat{y} \in \mathbb{R}^{n-k-1}$  and  $\alpha \in \mathbb{R}$  (which results automatically), such that

$$H(\hat{y}) \begin{bmatrix} a_1^{(k)} \\ \vdots \\ a_{n-k-1}^{(k)} \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix} \Rightarrow U_{k+1} := \begin{bmatrix} I_{k+1} & H(\hat{y}) \end{bmatrix}$$

and we get

$$\underbrace{U_{k+1} Q_k}_{=:Q_{k+1}} \underbrace{A Q_k U_{k+1}}_{=:Q_{k+1}^T} = \left[ \begin{array}{ccc|cc|ccc} * & \dots & * & * & & * & \dots & * \\ * & \dots & * & * & & * & \dots & * \\ \ddots & \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ & * & * & * & \dots & * & \dots & * \\ \hline & & & \alpha & * & \dots & * \\ & & & 0 & * & \dots & * \\ & & & \vdots & \ddots & \vdots & \\ & & & 0 & * & \dots & * \end{array} \right] U_{k+1};$$

the upper left unit matrix  $I_{k+1}$  in matrix  $U_{k+1}$  takes care to have on the first  $k+1$  columns a Hessenberg structure. MATLAB Source 8.2 gives a method for conversion of a matrix into upper Hessenberg form. To conclude, our QR method will be a two step method:

1. Convert  $A$  into Hessenberg form using an orthogonal transformation:

$$H^{(0)} = Q A Q^T, \quad Q^T Q = Q Q^T = I.$$

2. Do QR iterations

$$H^{(k+1)} = H_*^{(k)}, \quad k \in \mathbb{N}_0,$$

hoping that all the sub-diagonal elements converge to zero.

Since sub-diagonal entries converge slowest, we can use the maximum of modulus as stopping criterion. This leads us to the *simple QR method*, see MATLAB Source 8.3, M-file `QRMETHOD1.m`. Of course, for complex eigenvalues this method iterates infinitely.

**Example 8.5.5.** We apply the new method to the matrix in Example 8.5.1. For various given tolerances  $\varepsilon$ , we get the results given in Table 8.2. Note that one gains a new decimal digit for sub-diagonal entries at each three iterations.  $\diamond$

**MATLAB Source 8.2 Reduction to upper Hessenberg form**

---

```

function [A,Q]=hessen_h(A)
%HESSEN_H - Householder reduction to Hessenberg form

[m,n]=size(A);
v=zeros(m,m);
Q=eye(m,m);
for k=1:m-2
    x=A(k+1:m,k);
    vk=mysign(x(1))*norm(x,2)*[1;zeros(length(x)-1,1)]+x;
    vk=vk/norm(vk);
    A(k+1:m,k:m)=A(k+1:m,k:m)-2*vk*(vk'*A(k+1:m,k:m));
    A(1:m,k+1:m)=A(1:m,k+1:m)-2*(A(1:m,k+1:m)*vk)*vk';
    v(k+1:m,k)=vk;
end
if nargout==2
    Q=eye(m,m);
    for j=1:m
        for k=m:-1:1
            Q(k:m,j)=Q(k:m,j)-2*v(k:m,k)*(v(k:m,k)'*Q(k:m,j));
        end
    end
end

```

---

**MATLAB Source 8.3 Pure (simple) QR Method**

---

```

function [lambda,it]=QRMethod1(A,t)
%QRMETHOD1 - Computes eigenvalues of a real matrix
%naive implementation
%Input
%    A - matrix
%    t - tolerance
%Output
%    lambda - eigenvalues - diagonal of R
%    it - no. of iterations

H=hessen_h(A);
it=0;
while norm(diag(H,-1),inf) > t
    H=HessenRQ(H);
    it=it+1;
end
lambda=diag(H);

```

---

$\varepsilon$	#iterations	$\lambda_1$	$\lambda_2$	$\lambda_3$
$10^{-3}$	11	4.56155	-0.999834	0.438281
$10^{-4}$	14	4.56155	-1.00001	0.438461
$10^{-5}$	17	4.56155	-0.999999	0.438446
$10^{-10}$	31	4.56155	-1	0.438447

Table 8.2: Results for Example 8.5.5

We can try a speedup of our method by decomposing a problem into subproblems. If we have a Hessenberg matrix with form

$$H = \left[ \begin{array}{cccc|ccccc} * & \dots & \dots & * & & & & & \\ * & \ddots & & \vdots & & & & & \\ & \ddots & \ddots & \vdots & & & & & \\ & & * & * & & & & & \\ \hline & & & & * & \dots & \dots & * & \\ & & & & * & \ddots & & \vdots & \\ & & & & & \ddots & \ddots & \vdots & \\ & & & & & & * & * & \end{array} \right] = \begin{bmatrix} H_1 & * \\ & H_2 \end{bmatrix},$$

then the eigenvalue problem for  $H$  may be decomposed into an eigenvalue problem for  $H_1$  and one for  $H_2$ .

According to [39], a sub-diagonal  $h_{j+1,j}$  entry is considered to be “small enough” if

$$|h_{j+1,j}| \leq \text{eps}(|h_{jj}| + |h_{j+1,j+1}|). \quad (8.5.2)$$

We shall do something simpler, namely we shall decompose a matrix if its least modulus sub-diagonal entries is less than a given tolerance. The procedure is as follows: the function for computing eigenvalues using QR iterations finds a decomposition into two matrices  $H_1$  and  $H_2$  that calls itself recursively for each of these matrices.

If one of these matrices is  $1 \times 1$ , the eigenvalue is trivially computed, and if it is  $2 \times 2$ , then its characteristic polynomial is

$$\begin{aligned} p_A(x) &= \det(A - xI) = x^2 - \text{trace}(A)x + \det(A) \\ &= x^2 + \underbrace{(a_{11} + a_{22})}_=:b x + \underbrace{(a_{11}a_{22} - a_{12}a_{21})}_=:c. \end{aligned}$$

If its discriminant  $b^2 - 4c$  is positive, then  $A$  have two real and distinct eigenvalues

$$x_1 = \frac{1}{2} \left( -b - \text{sgn}(b)\sqrt{b^2 - 4c} \right) \text{ and } x_2 = \frac{c}{x_1},$$

otherwise its eigenvalues are complex, namely

$$\frac{1}{2} \left( -b \pm i\sqrt{4c - b^2} \right);$$

thus we can deal with complex eigenvalues. The function `Eigen2x2` returns the eigenvalues of a  $2 \times 2$  matrix. The idea is implemented in M-file `QRSplit1a.m`, MATLAB source 8.4. The M-file `QRIter.m` contains the QR iterations, and the file `Eigen2x2.m` deals with the complex case.

---

**MATLAB Source 8.4 QRSplit1a – QR method with partition and treatment of  $2 \times 2$  matrices**


---

```

function [lambda, It]=QRSPPLIT1(A, t)
%QRSPPLIT1 eigenvalues with partition and special treatment
% of 2x2 matrices
%Input
%A - matrix
%t - tolerance
%Output
%lambda - eigenvalues
%It - no. of iterations

[m, n]=size (A) ;
if n==1
    It=0;
    lambda=A;
    return
elseif n==2
    It=0;
    lambda=Eigen2x2 (A) ;
    return
else
    H=hessen_h (A); %convert to Hessenberg form
    [H1, H2, It]=QRITER(H, t); %decomposition H->H1, H2
    %recursive call
    [l1, It1]=QRSPPLIT1(H1, t);
    [l2, It2]=QRSPPLIT1(H2, t);
    It=It+It1+It2;
    lambda=[l1; l2];
end

```

---

**Example 8.5.6.** Let us consider again the matrix in Example 8.5.2; we apply the algorithm implemented in MATLAB Source 8.4 to it. The results are given in Table 8.3.  $\diamond$

### 8.5.2 Spectral shift

Hessenberg matrices allow us to execute each iteration into a shorter time. We shall try to reduce the number of iterations, that is to increase the speed of convergence, since

*The convergence rate of sub-diagonal entries  $h_{j+1,j}$  has order of growth*

$$\left( \frac{\lambda_{j+1}}{\lambda_j} \right)^k, \quad j = \overline{1, n-1}.$$

The keyword is here *spectral shift*. One observes that for  $\mu \in \mathbb{R}$  the matrix  $A - \mu I$  has the eigenvalues  $\lambda_1 - \mu, \dots, \lambda_n - \mu$ . For an arbitrary invertible matrix  $B$  the matrix  $B(A - \mu I)B^{-1} + \mu I$  has the

**MATLAB Source 8.5** Compute eigenvalues of a  $2 \times 2$  matrix

```
function lambda=Eigen2x2(A)
%EIGEN2X2 - Compute eigenvalues of a 2x2 matrix
%A - 2x2 matrix
%lambda - eigenvalues

b=trace(A); c=det(A);
d=b^2/4-c;
if d > 0
    if b == 0
        lambda = [sqrt(c); -sqrt(c)];
    else
        x = (b/2+sign(b)*sqrt(d));
        lambda=[x; c/x];
    end
else
    lambda=[b/2+i*sqrt(-d); b/2-i*sqrt(-d)];
end
```

---

**MATLAB Source 8.6** QR iterations on a Hessenberg matrix

```
function [H1,H2,It]=QRIter(H,t)
%Qriter - perform QR iteration on Hessenberg matrix
%until the least subdiagonal element is < t
%Input
% H - Hessenberg matrix
% t - tolerance
%Output
%H1, H2 - descomposition over least element
%It - no. of iterations

It=0; [m,n]=size(H);
[m, j]=min(abs(diag(H,-1)));
while m>t
    It=It+1;
    H=HessenRQ(H);
    [m, j]=min(abs(diag(H,-1)));
end
H1=H(1:j,1:j);
H2=H(j+1:n, j+1:n);
```

---

$\varepsilon$	#iterații	$\lambda_1$	$\lambda_2$	$\lambda_3$
$10^{-3}$	12	9.7406	$-3.8703 + 0.6479i$	$-3.8703 - 0.6479i$
$10^{-4}$	14	9.7407	$-3.8703 + 0.6479i$	$-3.8703 - 0.6479i$
$10^{-5}$	17	9.7407	$-3.8703 + 0.6480i$	$-3.8703 - 0.6480i$
$10^{-5}$	19	9.7407	$-3.8703 + 0.6480i$	$-3.8703 - 0.6480i$
$10^{-5}$	22	9.7407	$-3.8703 + 0.6480i$	$-3.8703 - 0.6480i$

Table 8.3: Results for Example 8.5.6

eigenvalues  $\lambda_1, \dots, \lambda_n$  – one may shift the spectrum of matrices forward and backwards by means of a similarity transformation. One sorts the eigenvalues  $\mu_1, \dots, \mu_n$  such that

$$|\mu_1 - \mu| > |\mu_2 - \mu| > \dots > |\mu_n - \mu|, \quad \{\mu_1, \dots, \mu_n\} = \{\lambda_1, \dots, \lambda_n\},$$

and if  $\mu$  is close to  $\mu_n$ , then if QR method start with  $H^0 = A - \mu I$ , the subdiagonal entry  $h_{n-1,n}^{(n)}$  converge very fast to zero. It is better if spectral shift is performed at each step individually. In addition, we may choose heuristically as approximation for  $\mu$  the value  $h_{nn}^{(k)}$ . One gets the following iterative scheme

$$H^{(k+1)} = (H^{(k)} - \mu_k I)_* + \mu_k I, \quad \mu_k := h_{nn}^{(k)}, \quad k \in \mathbb{N}_0,$$

with the starting matrix  $H^0 = QAQ^T$ . The M-file `QRsplit2.m`, MATLAB Source 8.7, gives a variant of the method which treats complex eigenvalues. It calls `QRIter2`. (See MATLAB Source 8.8).

**Remark 8.5.7.** If the shift value  $\mu$  is sufficiently close to an eigenvalue  $\lambda$ , then the matrix could be decomposed in a single iterative step.  $\diamond$

### 8.5.3 Double shift QR method

It can be shown that spectral shift QR method converges *quadratically*, that is the error is, for  $\rho < 1$ ,

$$O(\rho^{2k}) \text{ instead of } O(\rho^k).$$

This nice idea works only for real eigenvalues; for complex eigenvalue it is problematic. Nevertheless, we can exploit the fact that eigenvalues appear in conjugated pairs. This leads us to “double shift methods”:

*Instead of shifting the spectrum with an eigenvalue, approximated heuristically by  $h_{n,n}^{(k)}$ , we could rather perform two shifts in a step, namely with eigenvalues of*

$$B = \begin{bmatrix} h_{n-1,n-1}^{(k)} & h_{n-1,n}^{(k)} \\ h_{n-1,n}^{(k)} & h_{n,n}^{(k)} \end{bmatrix}.$$

There are two possibilities: either both eigenvalues  $\mu$  and  $\mu'$  of  $B$  are real and we proceed as above, or we have a pair of complex conjugated eigenvalues,  $\mu$  and  $\bar{\mu}$ . As we shall see, the second case could be also treated in real arithmetic. Let  $Q_k, Q'_k \in \mathbb{C}^{n \times n}$  and  $R_k, R'_k \in \mathbb{C}^{n \times n}$  the matrices of complex QR decomposition

$$\begin{aligned} Q_k R_k &= H^{(k)} - \mu I, \\ Q'_k R'_k &= R_k Q_k + (\mu - \bar{\mu}) I. \end{aligned}$$

---

**MATLAB Source 8.7** Spectral shift QR method, partition and treatment of complex eigenvalues
 

---

```

function [lambda, It]=QRSSplit2(A, t)
%QRSPLIT2 eigenvalues with partition and special treatment
% of 2x2 matrix
%Input
%A - matrix
%t - tolerance
%Output
%lambda - eigenvalues
%It - no. of iterations

[m, n]=size(A);
if n==1
    It=0;
    lambda=A;
    return
elseif n==2
    It=0;
    lambda=Eigen2x2(A);
    return
else
    H=hessen_h(A); %convert to Hessenberg
    [H1, H2, It]=QRIter2(H, t); %decomposition H->H1, H2
    %recursive call
    [l1, It1]=QRSSplit2(H1, t);
    [l2, It2]=QRSSplit2(H2, t);
    It=It+It1+It2;
    lambda=[l1; l2];
end

```

---

Then it holds

$$\begin{aligned}
 H^{(k+1)} &:= R'_k Q'_k + \mu I = (Q'_k)^*(R_k Q_k + (\mu - \bar{\mu})I)Q'_k + \bar{\mu}I \\
 &= Q_k^* R_k Q_k Q'_k + \mu I = (Q'_k)^* Q_k^* (H^{(k)} - \mu I) Q_k Q'_k + \mu I \\
 &= \underbrace{(Q_k Q'_k)^*}_{=U^*} \underbrace{H^{(k)}}_{=U} \underbrace{Q_k Q'_k}_{=U}.
 \end{aligned}$$

Using the matrix  $S = R'_k R_k$  we have

$$\begin{aligned}
 US &= Q_k Q'_k R'_k R_k = Q_k (R_k Q_k + (\mu - \bar{\mu})I) R_k \\
 &= Q_k R_k Q_k R_k + (\mu - \bar{\mu}) Q_k R_k = (H^{(k)} - \mu I)^2 + (\mu - \bar{\mu})(H^{(k)} - \mu I) \\
 &= (H^{(k)})^2 - 2\mu H^{(k)} + \mu^2 I + (\mu - \bar{\mu})H^{(k)} - (\mu^2 - \mu\bar{\mu})I \\
 &= (H^{(k)})^2 - (\mu + \bar{\mu})H^{(k)} + \mu\bar{\mu}I =: X
 \end{aligned} \tag{8.5.3}$$

If  $\mu = \alpha + i\beta$ , then  $\mu + \bar{\mu} = 2\alpha$  and  $\mu\bar{\mu} = |\mu|^2 = \alpha^2 + \beta^2$ , hence the matrix  $X$  in the righthand side of (8.5.3) is real, so it has a real QR decomposition  $X = QR$  and by Lemma 8.4.5 there exist a phase

---

**MATLAB Source 8.8 QR iteration and partition**

---

```

function [H1,H2,It]=QRIter2(H,t)
%QRITER - perform QR iteration on Hessenberg matrix
%until the least subdiagonal element is < t
%Input
% H - Hessenberg matrix
% t - tolerance
%Output
%H1, H2 - descomposition over least element
%It - no. of iterations

It=0; [m,n]=size(H);
II=eye(n);
[m, j]=min(abs(diag(H,-1)));
while m > t
    It=It+1;
    H=HessenRQ(H-H(n,n)*II)+H(n,n)*II;
    [m, j]=min(abs(diag(H,-1)));
end
H1=H(1:j,1:j);
H2=H(j+1:n, j+1:n);

```

---

matrix  $\Theta \in \mathbb{C}^{n \times n}$  such that  $U = \Theta Q$ . If we perform real iteration further, we obtain *double shift QR method*

$$\begin{aligned}
Q_k R_k &= (H^{(k)})^2 - (h_{n-1,n-1}^{(k)} + h_{n,n}^{(k)}) H^{(k)} \\
&\quad + \left( (h_{n-1,n-1}^{(k)} h_{n,n}^{(k)} - h_{n-1,n}^{(k)} H_{n,n-1}^{(k)}) I, \right. \\
H^{(k+1)} &= Q_k^T H^{(k)} Q_k.
\end{aligned} \tag{8.5.4}$$

- Remark 8.5.8 (Double shift QR method).**
1. The matrix  $X$  in (8.5.3) is no more a Hessenberg matrix, since it has an additional diagonal. Nevertheless, one could easily compute the QR decomposition of  $X$ , using only  $2n - 3$  Jacobi rotations, instead of  $n - 1$ , the number required by a Hessenberg matrix.
  2. Because of its high complexity, the multiplication  $Q_k^T H^{(k)} Q_k$  is no more an effective method for our iteration; we can fix this drawback, see for example [29] or [81, pages 272–278].
  3. Naturally,  $H^{(k+1)}$  could be converted to Hessenberg form.
  4. Double shift QR method is useful only when  $A$  has complex eigenvalues; for symmetric matrices it is not advantageous.  $\diamond$

Double shift QR method with partitioning and treatment of  $2 \times 2$  matrices is given in MATLAB Source 8.9, file `QRSSplit3`. It calls `QRDouble`.

**Example 8.5.9.** We apply sources 8.7 and 8.9 to matrices in Examples 8.5.1 and 8.5.2. One gets the results in Table 8.4. The good behavior of double shift QR method can be explained by the idea to obtain two eigenvalues simultaneously.  $\diamond$

**MATLAB Source 8.9** Double shift QR method with partition and treating  $2 \times 2$  matrices

```

function [lambda, It]=QRSSplit3(A,t)
%QRSPLIT3 compute eigenvalues with QR method, partition, shift
%and special treatment of 2x2 matrices
%Input
%A - matrix
%t - tolerance
%Output
%lambda - eigenvalues
%It - no. of iterations

[m,n]=size(A);
if n==1
    It=0;
    lambda=A;
    return
elseif n==2
    It=0;
    lambda=Eigen2x2(A);
    return
else
    H=hessen_h(A); %convert to Hessenberg
    [H1,H2,It]=QRDouble(H,t); %decomposition H->H1,H2
    %recursiv call
    [l1,It1]=QRSSplit3(H1,t);
    [l2,It2]=QRSSplit3(H2,t);
    It=It+It1+It2;
    lambda=[l1;l2];
end

```

$\varepsilon$	#iterations in $\mathbb{R}$		#iterations in $\mathbb{C}$	
	alg. 8.7	alg. 8.9	alg. 8.7	alg. 8.9
1e-010	1	1	9	4
1e-020	9	2	17	5
1e-030	26	3	45	5

Table 8.4: Comparisons in Example 8.5.9

---

**MATLAB Source 8.10** Double shift QR iterations and Hessenberg transformation

---

```

function [H1,H2,It]=QRDouble(H,t)
%QRDOUBLE - perform double step QR iteration and inverse
%transform on Hessenberg matrix until the least
%subdiagonal element is < t
%Input
% H - Hessenberg matrix
% t - tolerance
%Output
%H1, H2 - descomposition over least element
%It - no. of iterations

It=0; [m,n]=size(H);
II=eye(n);
[m, j]=min(abs(diag(H,-1)));
while m>t
    It=It+1;
    X = H*H ... % X matrix
    - (H(n-1,n-1) + H(n,n)) * H ...
    + (H(n-1,n-1)*H(n,n) - H(n,n-1)*H(n-1,n))*II;
    [Q,R]=qr(X);
    H=hessen_h(Q'*H*Q);
    [m, j]=min(abs(diag(H,-1)));
end
H1=H(1:j,1:j);
H2=H(j+1:n,j+1:n);

```

---

## 8.6 Eigenvalues and Eigenvectors in MATLAB

MATLAB uses LAPACK routines to compute eigenvalues and eigenvectors. The eigenvalues of  $A$  are computed with the `eig` function: `e = eig(A)` assigns the eigenvalues to the vector `e`. More generally, after the call `[V,D]=eig(A)`, the diagonal  $n$ -by- $n$  matrix `D` contains eigenvalues on the diagonal and the columns of the  $n$ -by- $n$  matrix `V` are eigenvectors. It holds  $A*V=V*D$ . Not every matrix has  $n$  linearly independent eigenvectors, so the matrix `V` returned by `eig` may be singular (or, because of roundoff, nonsingular but very ill conditioned). The matrix in the following example has a double eigenvalue 1 and only one linear independent eigenvector:

```

>> [V,D]=eig([2, -1; 1,0])
V =
    0.7071    0.7071
    0.7071    0.7071
D =
    1         0
    0         1

```

MATLAB normalizes so that each column of `V` has unit 2-norm (this is possible, since if  $x$  is an eigenvector then so is any nonzero multiple of  $x$ ).

For Hermitian matrices MATLAB returns eigenvalues sorted in increasing order and the matrix of eigenvectors is unitary to working precision:

```
>> [V,D]=eig([2,-1;-1,1])
V =
   -0.5257    -0.8507
   -0.8507     0.5257
D =
   0.3820         0
         0    2.6180
>> norm(V'*V-eye(2))
ans =
  2.2204e-016
```

The following example computes the eigenvalues of the (non-Hermitian) Frank matrix:

```
>> F = gallery('frank',5)
F =
   5     4     3     2     1
   4     4     3     2     1
   0     3     3     2     1
   0     0     2     2     1
   0     0     0     1     1
>> e = eig(F)'
e =
  10.0629    3.5566    1.0000    0.0994    0.2812
```

if  $\lambda$  is an eigenvalue of  $F$ , then so is  $1/\lambda$ .

```
>> 1./e
ans =
  0.0994    0.2812    1.0000   10.0629    3.5566
```

The reason is that the characteristic polynomial is anti-palindromic:

```
>> poly(F)
ans =
  1.0000  -15.0000   55.0000  -55.0000   15.0000  -1.0000
```

Thus,  $\det(F - \lambda I) = -\lambda^5 \det(F - \lambda^{-1} I)$ .

If  $\lambda$  is an eigenvalue of  $A$ , a nonzero vector  $y$  such that  $y^* A = \lambda y^*$  is a *left eigenvector*. Use  $[W, D] = \text{eig}(A.^*)$ ;  $W = \text{conj}(W)$  to compute the left eigenvectors of  $A$ . If  $\lambda$  is a simple eigenvalue of  $A$  with a right eigenvector  $x$  and a left eigenvector  $y$  such that  $\|x\|_2 = \|y\|_2 = 1$ , then the *condition number of the eigenvalue*  $\lambda$  is  $\kappa(\lambda, A) = \frac{1}{|y^* z|}$ . The condition number of the eigenvector matrix is an upper bound for the individual eigenvalue condition numbers.

Function `condeig` computes condition numbers for the eigenvalues. The command `c=condeig(A)` returns a vector of condition numbers for the eigenvalue of  $A$ . The call `[V, D, s] = condeig(A)` is equivalent to: `[V, D] = eig(A), s = condeig(A)`. A large condition number indicates an eigenvalue that is sensitive to perturbations in the matrix. The following example displays eigenvalues of the sixth order Frank matrix in the first row and their condition numbers in the second:

```

>> A = gallery('frank', 6);
>> [V, D, s] = condeig(A);
>> [diag(D)'; s']
ans =
12.9736    5.3832    1.8355    0.5448    0.0771    0.1858
1.3059    1.3561    2.0412   15.3255   43.5212   56.6954

```

Let us explain shortly how `eig` function actions. It works in several stages. First, when  $A$  is nonsymmetric, it balances the matrix, that is, it carries out a similarity transform  $A \leftarrow Y^{-1}AY$ , where  $Y$  is a permutation of a diagonal matrix chosen to give  $A$  rows and columns of approximately equal norm. The motivation for balancing is that it can lead to a more accurate computed eigensystem. However, balancing can worsen rather than improve the accuracy (see `doc eig` for an example), so it may be necessary to turn balancing off with `eig(A, 'nobalance')`. Also note that if  $A$  is symmetric, `eig(A, 'nobalance')` ignores the `nobalance` option since  $A$  is already balanced.

After balancing, `eig` reduces  $A$  to Hessenberg form, then uses the QR algorithm to reach Schur form, after which eigenvectors are computed by substitution if required. The Hessenberg factorization is computed by  $H = \text{hess}(A)$  or  $[Q, H] = \text{hess}(A)$ . The last form returns also the transformation matrix  $Q$ . The commands  $T = \text{schur}(A)$  or  $[Q, T] = \text{schur}(A)$ , produce the real Schur decomposition if  $A$  is real and the real Schur decomposition if  $A$  is complex. The complex Schur form of a real matrix can be obtained for with `schur(A, 'complex')`.

If  $A$  is real and symmetric (complex Hermitian),  $[V, D] = \text{eig}(A)$  reduces initially to symmetric (Hermitian) tridiagonal form then iterates to produce a diagonal Schur form, resulting in an orthogonal (unitary)  $V$  and a real, diagonal  $D$ .

MATLAB can solve generalized eigenvalue problems: given two square matrices of order  $n$ ,  $A$  and  $B$ , find the scalars  $\lambda$  and vectors  $x \neq 0$  such that  $Ax = \lambda Bx$ . The generalized eigenvalues are computed by  $e = \text{eig}(A, B)$ , while  $[V, D] = \text{eig}(A, B)$  computes an  $n$ -by- $n$  diagonal matrix  $D$  and an  $n$ -by- $n$  matrix  $V$  of eigenvectors such that  $A \cdot V = B \cdot V \cdot D$ . The theory of the generalized eigenproblem is more complicated than that of the standard eigenproblem, with the possibility of zero, finitely many or infinitely many eigenvalues and of eigenvalues that are infinitely large. When  $B$  is singular `eig` may return computed eigenvalues containing NaNs. We illustrate with

```

>> A = gallery('triw', 3), B = magic(3)
A =
1     -1     -1
0      1     -1
0      0      1
B =
8      1      6
3      5      7
4      9      2
>> [V, D] = eig(A, B); V, eigvals = diag(D)'
V =
-1.0000   -1.0000    0.3526
0.4844   -0.4574    0.3867
0.2199   -0.2516   -1.0000
eigvals =
0.2751    0.0292   -0.3459

```

Function `polyeig` solves the polynomial eigenvalue problem  $(\lambda^p A_p + \lambda^{p-1} A_{p-1} + \dots + \lambda A_1 + A_0)x = 0$ , where the  $A_i$  are given square coefficient matrices. The generalized eigenproblem is

obtained for  $p = 1$ ; if we take further  $A_0 = I$  we get the standard eigenproblem. The quadratic eigenproblem  $(\lambda^2 A + \lambda B + C)x = 0$  corresponds to  $p = 2$ . If  $A_p$  is  $n$ -by- $n$  and nonsingular then there are  $pn$  eigenvalues. MATLAB's syntax is `e = polyeig(A0, A1, ..., Ap)` or `[X, e] = polyeig(A0, A1, ..., Ap)`, with  $e$  a  $pn$ -vector of eigenvalues and  $X$  an  $n$ -by- $pn$  matrix whose columns are the corresponding eigenvectors. Example:

```
>> A = eye(2); B = [20 -10; -10 20]; C = [15 -5; -5 15];
>> [X,e] = polyeig(C,B,A)
X =
    0.7071    0.7071    0.7071    0.7071
   -0.7071    0.7071   -0.7071    0.7071
e =
   -29.3178
   -8.8730
   -0.6822
   -1.1270
```

The *singular value decomposition* (SVD) of an  $m$ -by- $n$  matrix  $A$  is the factorization

$$A = U\Sigma V^*, \quad (8.6.1)$$

where  $U$  is  $m \times m$  and unitary,  $V$  is  $n \times n$  and unitary, and  $\Sigma$  is  $m \times n$  and diagonal with positive real entries  $\sigma_{ii}$ , such that  $\sigma_{11} \geq \sigma_{2,2} \geq \sigma_{\min(m,n)} \geq 0$ .

There exist two kind of SVD: the full or complete SVD, given by (8.6.1), and the reduced or economical SVD, that accept a rectangular  $m \times n$  matrix, and returns the  $m \times n$  matrix  $U$ , the diagonal  $n \times n$  matrix  $\Sigma$  and the unitary  $n \times n$  matrix  $V$ .

SVD is a useful instrument for the analysis of mappings defined on a space and having values into a different space, possible with a different dimensions. The rank, null space, range space of a matrix are computed via SVD. SVD has many applications in Statistics and Image Processing and helps to a better understanding of linear algebra concepts. If  $A$  is symmetric and positive definite, SVD (8.6.1) and eigenvalue decomposition agree. In contrast to eigenvalue decomposition, SVD always exists.

Consider the matrix

```
A =
    9      4
    6      8
    2      7
```

Its full (complete) SVD is

```
>> [U, S, V] = svd(A)
U =
   -0.6105    0.7174    0.3355
   -0.6646   -0.2336   -0.7098
   -0.4308   -0.6563    0.6194

S =
  14.9359      0
      0    5.1883
      0      0
```

```
V =
-0.6925    0.7214
-0.7214   -0.6925
```

and the reduced SVD

```
>> [U,S,V]=svd(A,0)
U =
-0.6105    0.7174
-0.6646   -0.2336
-0.4308   -0.6563

S =
14.9359      0
0        5.1883

V =
-0.6925    0.7214
-0.7214   -0.6925
```

In both cases  $U \cdot S \cdot V'$  is equal to  $A$ , modulo roundoff.

The generalized singular value decomposition of an  $m \times p$  matrix  $A$  and an  $n \times p$  matrix  $B$  can be written

$$A = U C X^*, \quad B = V S X^*, \quad C^* C + S^* S = I,$$

where  $U$  and  $V$  are unitary and  $C$  and  $S$  are real diagonal matrices with nonnegative diagonal elements. The numbers  $C(i,i)/S(i,i)$  are the generalized singular values. This decomposition is computed by  $[U,V,X,C,S] = \text{gsvd}(A,B)$ . See `help gsvd` for more details about the dimensions of the factors. For details on generalized SVD, see [39].

## 8.7 Applications

### 8.7.1 Solving mass-spring systems

Demmel presents in [23] an application of the concepts of eigenvalue and eigenvector to a problem of *mechanical vibrations*. Consider the damped mass spring system in Figure 8.1. Newton's law  $F = ma$  leads us to the system

$$\begin{aligned} m_i \ddot{x}_i(t) &= k_i (x_{i-1}(t) - x_i(t)) \\ &\quad + k_{i+1} (x_{i+1}(t) - x_i(t)) \\ &\quad - b_i \dot{x}_i(t). \end{aligned}$$

The first term is the force on mass  $i$  from spring  $i$ , the second is the force on mass  $i$  from spring  $i+1$  and the last term is the force on mass  $i$  from damper  $i$ . In matrix form, our equation is

$$M \ddot{x}(t) = -B \dot{x}(t) - K x(t), \quad (8.7.1)$$

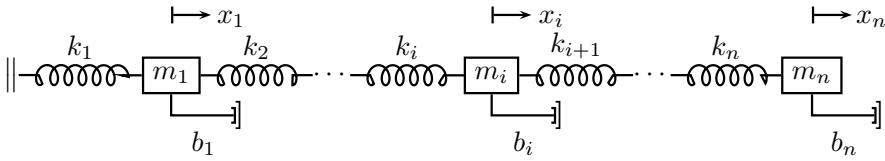


Figure 8.1: Damped, vibrating mass-spring system. Here,  $x_i$  is the position of the  $i$ th mass,  $m_i$  is the  $i$ th mass,  $k_i$  is the spring constant of the  $i$ th spring and  $b_i$  is the damping constant of  $i$ th damper

where  $M = \text{diag}(m_1, \dots, m_n)$ ,  $B = \text{diag}(b_1, \dots, b_n)$ , and

$$K = \begin{bmatrix} k_1 + k_2 & -k_2 & & & \\ -k_2 & k_2 + k_3 & -k_3 & & \\ & \ddots & \ddots & \ddots & \\ & & -k_{n-1} & k_{n-1} + k_n & -k_n \\ & & & -k_n & k_n \end{bmatrix}.$$

We assume that all the masses  $m_i$  are positive.  $M$  is called the *mass matrix*,  $B$  is the *damping matrix*, and  $K$  is the *stiffness matrix*. First, we convert this second-order differential equation to a first-order differential equation. If we introduce

$$y(t) = \begin{bmatrix} \dot{x}(t) \\ x(t) \end{bmatrix},$$

our equation becomes

$$\begin{aligned} \dot{y}(t) &= \begin{bmatrix} \ddot{x}(t) \\ \dot{x}(t) \end{bmatrix} = \begin{bmatrix} -M^{-1}B\dot{x}(t) - M^{-1}Kx(t) \\ \dot{x}(t) \end{bmatrix} \\ &= \begin{bmatrix} -M^{-1}B & M^{-1}K \\ I & 0 \end{bmatrix} \begin{bmatrix} \dot{x}(t) \\ x(t) \end{bmatrix} \\ &= \begin{bmatrix} -M^{-1}B & M^{-1}K \\ I & 0 \end{bmatrix} y(t) \equiv Ay(t). \end{aligned} \quad (8.7.2)$$

To solve  $\dot{y}(t) = Ay(t)$ , we assume that  $y(0)$  is given (i.e. the initial positions  $x(0)$  and velocities  $\dot{x}(t)$  are given).

One way to express the solution of this differential equation is  $y(t) = e^{At}y(0)$ , where  $e^{At}$  is the matrix exponential. We shall give the solution in the special case where  $A$  is diagonalizable; this will be true for almost all choices of  $m_i$ ,  $k_i$ , and  $b_i$ .

When  $A$  is diagonalizable, we can write  $A = SAS^{-1}$ , where  $\Lambda = (\lambda_1, \dots, \lambda_n)$ . Then  $\dot{y}(t) = Ay(t)$  is equivalent to  $\dot{y}(t) = S\Lambda S^{-1}y(t)$  or  $S^{-1}\dot{y}(t) = \Lambda S^{-1}y(t)$  or  $\dot{z}(t) = \Lambda z(t)$ , where  $z(t) = S^{-1}y(t)$ . This diagonal system of differential equations  $\dot{z}_i(t) = \lambda_i z_i(t)$  has solutions  $z_i(t) = e^{\lambda_i t} z_i(0)$ , so  $y(t) = S\text{diag}(e^{\lambda_1 t}, \dots, e^{\lambda_n t})S^{-1}y(0) = Se^{At}S^{-1}y(0)$ .

```
function [T, Locations, Velocities]=massspring(n,m,b,k,x0,x,v,dt,tfinal)
% MASSSPRING Solve vibrating mass-spring system using eigenvalues
%
% Inputs
%   N = number of bodies
```

```

% M = column vector of n masses
% B = column vector of n damping constants
% K = column vector of n spring constants
% X0= column vector of n rest positions of bodies
% X = column vector of n initial displacements from rest
% V = column vector of n initial velocities of bodies
% DT = time step
% TFINAL = final time to integrate system
%
% Outputs
% graph body positions, T, LOCATIONS, VELOCITIES
%

% Compute mass matrix
M = diag(m);
% Compute damping matrix
B = diag(b);
% Compute stiffness matrix
if n>1,
    K = diag(k)+diag([k(2:n);k(n)])-diag(k(2:n),1)-diag(k(2:n),-1);
else
    K = k;
end
% Compute matrix governing motion, find eigenvalues and eigenvectors
A = [[-inv(M)*B, -inv(M)*K]; [eye(n), zeros(n)]];
[V,D]=eig(A);
dD = diag(D);
iV = inv(V);
i=1;
Y = [v;x];
T = 0;
steps = round(tfinal/dt);
% Compute positions and velocities
for i = 2:steps
    t = (i-1)*dt;
    T = [T,t];
    Y(1:2*n,i) = V * ( diag(exp(t*dD)) * ( iV * [v;x] ) );
end
Y = real(Y);
hold off, clf
subplot(2,1,1)
Locations = Y(n+1:2*n,:)+x0*ones(1,i);
attr={'k-','r--','g-.','b:'};
for j=1:n,
    color=attr{rem(j,4)+1};
    plot(T,Locations(j,:),color),
    hold on
    axis([0,tfinal,min(min(Locations)),max(max(Locations))])
end

```

```

title('Positions')
xlabel('Time')
grid
subplot(2,1,2)
Velocities = Y(1:n,:);
for j=1:n,
    color=attr{rem(j,4)+1};
    plot(T,Velocities(j,:),color),
    hold on
    axis([0,tfinal,min(min(Velocities)),max(max(Velocities))])
end
title('Velocities')
xlabel('Time')
grid

```

The MATLAB function `massspring` finds time moments, positions and the velocities of the system components, and plot the graph. The call sequence is

```

n=4;
b=0.4*ones(4,1);
m=[2,1,1,2]';
k=ones(4,1);
x0=(1:4)';
x=[-0.25,0,0,0.25]';
v=[-1,0,0,1]';
dt=0.1;;
tfinal=20;
[T,P,V]=massspring(n,m,b,k,x0,x,v,dt,tfinal);

```

See Figure 8.2 for output.

## 8.7.2 Computing natural frequencies of a rectangular membrane

One of the most useful applications of eigenvalue problems occurs in natural frequency calculation of linear systems. In [105] one considers the finite difference approximation for the natural frequencies of a rectangular membrane and the approximate results are compared to exact values. Consider a tightly stretched elastic membrane occupying a region  $R \subset \mathbb{R}^2$  bounded by a curve  $L$  on which the traverse deflection is zero. The PDE and boundary conditions governing the traverse motion  $U(x, y, t)$  are

$$\begin{aligned} T(U_{xx} + U_{yy}) &= \rho U_{tt}, \quad (x, y) \in R \\ U(x, y, t) &= 0, \quad (x, y) \in L, \end{aligned}$$

where  $T$  is the membrane tension and  $\rho$  is the density. The natural vibration modes are motion states where all points of the system simultaneously move with the same frequency, which says  $U(x, y, t) = u(x, y) \sin \Omega t$ . It follows that  $u(x, y)$  satisfies

$$\begin{aligned} u_{xx} + u_{yy} &= -\omega^2 u, \quad (x, y) \in R \\ u(x, y) &= 0, \quad (x, y) \in L, \end{aligned}$$

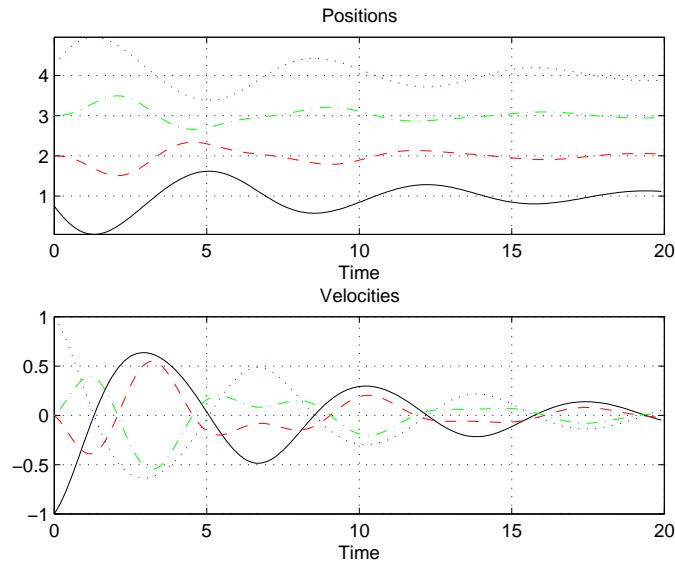


Figure 8.2: Positions and velocities of a mass-spring system

where  $\omega = \sqrt{\frac{P}{t}}\Omega$ . In the simple case of a rectangular membrane lying in the region such that  $0 \leq x \leq a$  and  $0 \leq y \leq b$ , the natural frequencies and mode shapes are given by

$$\omega_{mn} = \sqrt{\left(\frac{n\pi}{a}\right)^2 + \left(\frac{m\pi}{b}\right)^2}, \quad u_{nm} = \sin \frac{n\pi x}{a} \sin \frac{m\pi y}{b}$$

where  $n, m \in \mathbb{N}^*$ . How closely these values can be reproduced when the partial differential equation is replaced by a second order finite difference approximation defined on a rectangular grid? We introduce the grid points expressed as

$$\begin{aligned} x(i) &= (i-1)\Delta_x, \quad i = 1, \dots, N \\ y(j) &= (j-1)\Delta_y, \quad j = 1, \dots, M, \end{aligned}$$

where  $\Delta_x = a/(N-1)$ ,  $\Delta_y = b/(M-1)$ , and let  $u(i, j)$  be the value of  $u$  at  $x(i), y(j)$ . The Helmholtz equation is replaced by an algebraic eigenvalue problem of the form

$$\begin{aligned} \Delta_y^2 [u(i-1, j) - 2u(i, j) + u(i+1, j)] + \\ \Delta_x^2 [u(i, j-1) - 2u(i, j) + u(i, j+1)] &= \lambda u(i, j) \end{aligned}$$

where

$$\lambda = (\Delta_x \Delta_y \omega)^2$$

and associated homogeneous boundary conditions

$$u(1, j) = u(N, j) = u(i, 1) = u(i, M) = 0.$$

This combination of equations can be written in matrix form as

$$Au = \lambda u, \quad Bu = 0.$$

We used the MATLAB function `null` to solve the boundary conditions equations. If  $Q$  is the null space of  $B$  (with orthonormal columns), we write  $u = Qz$  and substitute into the eigenvalue equations. Multiplying both sides by  $Q^T$ , we obtain a standard eigenvalue problem of the form  $Cz = \lambda z$ , where  $C = Q^T A Q$ . The eigenvector matrix of the original problem is obtained as  $u = QV$ , where  $V$  is the eigenvector matrix of  $C$ , and the eigenvalues of the original matrix are the eigenvalues of  $C$  ( $C$  and  $A$  are similar).

The function `recmemnfr` form and solve the algebraic equations just discussed. To avoid the tedious indexing into  $n^2$  dimensional matrices the MATLAB functions `ind2sub` and `sub2ind` are useful.

```

function [w,wex,modes,x,y,nx,ny,ax,by]=recmemnfr(...
    ax,by,nx,ny)
%RECMEMNFR - natural frequencies of a rectangular membrane
% [w,wex,modes,x,y,nx,ny,ax,by]=recmemfr(a,b,nx,ny,noplt)
% ~~~~~
% % ax, by - membrane side lengths
% nx,ny - number of points
% w - vector of (nx-2)*(ny-2) computed frequencies
% wex - vector of exact frequencies
% modes - three dimensional array containing the mode
% shapes for various frequencies. The array
% size is [nx,ny,(nx-2)*(ny-2)] denoting
% the x direction, y direction, and the
% frequency numbers matching components of the
% w vector. The i'th mode shape is obtained
% as reshape(vecs(:,i),n,m)
% x,y - vectors defining the finite difference grid

if nargin==0; ax=2; nx=20; by=1; ny=10; end
dx=ax/(nx-1); dy=by/(ny-1);
na=(1:nx-1)'/ax; nb=(1:ny-1)/by;

% Compute exact frequencies for comparison
wex=pi*sqrt(repmat(na.^2,1,ny-1)+repmat(nb.^2,nx-1,1));
wex=sort(wex(:)');
x=linspace(0,ax,nx);
y=linspace(0,by,ny); neig=(nx-2)*(ny-2); nvar=nx*ny;
% Form equations to fix membrane edges
k=0; s=[nx,ny]; c=zeros(2*(nx+ny),nvar);
for j=1:nx
    m=sub2ind(s,[j,j],[1,ny]); k=k+1;
    c(k,m(1))=1; k=k+1; c(k,m(2))=1;
end
for j=1:ny
    m=sub2ind(s,[1,nx],[j,j]); k=k+1;
    c(k,m(1))=1; k=k+1; c(k,m(2))=1;
end

% Form frequency equations at interior points
k=0; a=zeros(neig,nvar); b=a;

```

```

phi=(dx/dy)^2; psi=2*(1+phi);
for i=2:nx-1
    for j=2:ny-1
        m=sub2ind(s,[i-1,i,i+1,i,i],[j,j,j,j-1,j+1]);
        k=k+1; a(k,m(1))=-1; a(k,m(2))=psi; a(k,m(3))=-1;
        a(k,m(4))=-phi; a(k,m(5))=-phi; b(k,m(2))=1;
    end
end

% Compute frequencies and mode shapes
q=null(c); A=a*q; B=b*q; [modes, lam]=eig(B\A);
[lam, k]=sort(diag(lam)); w=sqrt(lam)'/dx;
modes=q*modes(:,k); modes=reshape(modes (:), nx, ny, neig);

```

The calling sequence and the plotting code for the first 50 frequencies is given below

```

% Plot first fifty approximate and exact frequencies
[w,wex,modes,x,y,nx,ny,ax,by]=recmemnfr;

m=1:min([50,length(w),length(wex)]);
pcter=100*(wex(m)-w(m))./wex(m);

clf; plot(m,wex(m),'k-',m,w(m),'k.',m,pcter,'k--')
xlabel('frequency number');
ylabel('frequency and % error')
legend('exact frequency','approx. frequency',...
    'percent error',2)
s=['MEMBRANE FREQUENCIES FOR AX / BY = ',...
    num2str(ax/by,5),' AND ',num2str(nx*ny),...
    ' GRID POINTS'];
title(s), grid on, shg

```

See Figure 8.3 for the graph of exact and approximate frequencies and the error.

## Problems

**Problem 8.1.** Compute the eigenvalues of Hilbert matrix for  $n = 10, 11, \dots, 20$  and the corresponding condition numbers.

**Problem 8.2.** The matrices

```

P=gallery('pascal',12)
F=gallery('frank',12)

```

have the property that, if  $\lambda$  is an eigenvalue, then,  $1/\lambda$  is an eigenvalue too. How well do the computed eigenvalues preserve this property? Use `condeig` two explain the different behavior of these two matrices.

**Problem 8.3.** What is the largest eigenvalue of `magic(n)`? Why?

**Problem 8.4.** Try the following command sequence:

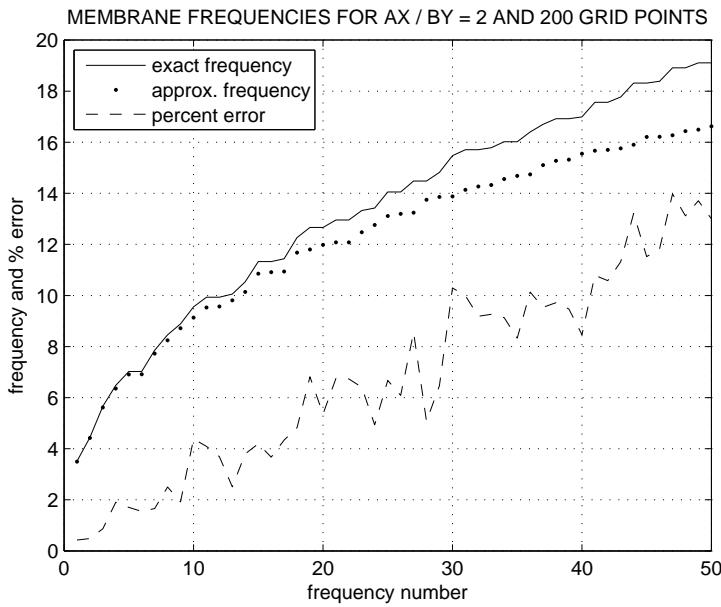


Figure 8.3: Approximate and exact frequencies for a rectangular membrane

```
n=100;
d=ones(n,1);
A=diag(d,1)+diag(d,-1);
e=eig(A);
plot(-n/2:n/2,e,'.')
```

Do you recognize the resulted curve? Could you find a formula for the eigenvalues of this matrix?

**Problem 8.5.** Let  $T_N$  be the matrix obtained by finite-difference discretization of univariate Poisson equation (problem 4.7). Their eigenvalues are

$$\lambda_j = 2 \left( 1 - \cos \frac{\pi j}{N+1} \right),$$

and their eigenvectors  $z_j$  have the components

$$z_j(k) = \sqrt{\frac{2}{N+1}} \sin \frac{jk\pi}{N+1}.$$

Give a graphical representation of eigenvalues and eigenvectors of  $T_{21}$ .

**Problem 8.6.** (a) Implement power method (vector iteration).

(b) Test the function in (a) for the matrix and the starting vector

$$A = \begin{pmatrix} 6 & 5 & -5 \\ 2 & 6 & -2 \\ 2 & 5 & -1 \end{pmatrix}, \quad x = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}.$$

(c) Approximate the spectral radius  $\rho(A)$  of the matrix

$$A = \begin{pmatrix} 2 & 0 & -1 \\ -2 & -10 & 0 \\ -1 & -1 & 4 \end{pmatrix},$$

using power method and starting vector  $[1, 1, 1]^T$ .

**Problem 8.7.** Find the eigenvalues of the matrix

$$\begin{pmatrix} 190 & 66 & -84 & 30 \\ 66 & 303 & 42 & -36 \\ 336 & -168 & 147 & -112 \\ 30 & -36 & 28 & 291 \end{pmatrix}$$

by using double shift QR method. Compare your result to that provided by `eig`.

**Problem 8.8.** Find the SVD for the following matrices:

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 7 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 5 \\ 4 \end{bmatrix}.$$

# CHAPTER 9

---

## Numerical Solution of Ordinary Differential Equations

---

### 9.1 Differential Equations

Let us consider the initial value (Cauchy<sup>1</sup>) problem: determine a vector valued  $y \in C^1[a, b]$ ,  $y : [a, b] \rightarrow \mathbb{R}^d$ , such that

$$(CP) \quad \begin{cases} \frac{dy}{dx} = f(x, y), & x \in [a, b], \\ y(a) = y_0. \end{cases} \quad (9.1.1)$$

We shall emphasize two important classes of such problems:

- (i) for  $d = 1$  we have a single first-order differential equation

$$\begin{cases} y' = f(x, y), \\ y(a) = y_0. \end{cases}$$

- (ii) for  $d > 1$  we have a system of first order differential equation

$$\begin{cases} \frac{dy^i}{dx} = f^i(x, y^1, y^2, \dots, y^d), & i = \overline{1, d}, \\ y^i(a) = y_0^i, & i = \overline{1, d}. \end{cases}$$

---

Augustin Louis Cauchy (1789-1857), French mathematician, active in Paris, is considered to be the father of modern analysis. He provided a firm foundation for analysis by basing it on a rigorous concept of limit. He is also the creator of complex analysis, where "Cauchy's formula" play a central role. In addition, Cauchy's name is attached to pioneering contributions to the theory of ordinary and partial differential equations, mainly in existence and uniqueness problems. Like other great mathematicians of 18th and 19th centuries, his work encompasses geometry, algebra, number theory, mechanics and theoretical physics.



**Remark 9.1.1.** Let us consider a single  $d$ -th order differential equation,

$$u^{(n)} = g(x, u, u', \dots, u^{(n-1)}),$$

with the initial condition  $u^{(i)}(a) = u_0^i$ ,  $i = \overline{0, d-1}$ . This problem is easily brought into the form (9.1.1) by defining

$$y^i = u^{(i-1)}, \quad i = \overline{1, d}.$$

Then

$$\begin{aligned} \frac{dy^1}{dx} &= y^2, & y^1(a) &= u_0^0, \\ \frac{dy^2}{dx} &= y^3, & y^2(a) &= u_0^1, \\ &\dots & & \\ \frac{dy^{d-1}}{dx} &= y^d, & y^{d-1}(a) &= u_0^{d-2}, \\ \frac{dy^d}{dx} &= g(x, y^1, y^2, \dots, y^d), & y^d(a) &= u_0^{d-1}. \end{aligned} \tag{9.1.2}$$

which has the form (9.1.1) with very special (linear) functions  $f^1, f^2, \dots, f^{d-1}$ , and  $f^d(x, y) = g(x, y)$ .  $\diamond$

We recall from the theory of differential equation the following basic existence and uniqueness result.

**Theorem 9.1.2.** Assume that  $f(x, y)$  is continuous in the first variable for  $x \in [a, b]$  and with respect to the second satisfies a uniform Lipschitz condition

$$\|f(x, y) - f(x, y^*)\| \leq L\|y - y^*\|, \quad y, y^* \in \mathbb{R}^d, \tag{9.1.3}$$

where  $\|\cdot\|$  is some vector norm. Then the initial value problem (CP) has a unique solution  $y(x)$ ,  $a \leq x \leq b$ ,  $\forall y_0 \in \mathbb{R}^d$ . Moreover,  $y(x)$  depends continuously on  $a$  and  $y_0$ .

The Lipschitz condition (9.1.3) certainly holds if all functions  $\frac{\partial f^i}{\partial y^j}(x, y)$ ,  $i, j = \overline{1, d}$  are continuous in the  $y$ -variables and bounded on  $[a, b] \times \mathbb{R}^d$ . This is the case for linear systems of differential equations, where

$$f^i(x, y) = \sum_{j=1}^d a_{ij}(x)y^j + b_i(x), \quad i = \overline{1, d}$$

and  $a_{ij}(x), b_i(x)$  are continuous functions on  $[a, b]$ .

Often the Lipschitz condition (9.1.3) holds in some compact neighbor of  $a$  in which  $y(x)$  remains in a compact  $D$ .

## 9.2 Numerical Methods

One can distinguish between *analytic approximation methods* and *discrete variable methods*. In the former one tries to find approximations  $y_a(x) \approx y(x)$  to the exact solutions, valid for all  $x \in [a, b]$ . This usually take the form of a truncated series expansion, either in powers of  $x$ , in Chebyshev polynomials, or in some other system of basis functions. In discrete-variable methods, one attempts to find approximations  $u_n \in \mathbb{R}^d$  of  $y(x_n)$  only at discrete points  $x_n \in [a, b]$ . The abscissas  $x_n$  may be predetermined

(e.g., equally spaced on  $[a, b]$ ), or, more likely, are generated dynamically as a part of the integration process.

If desired, one can then from these discrete approximations  $\{u_n\}$  obtain again an approximation  $y_a(x)$  defined for all  $x \in [a, b]$ , either by interpolation or, by a continuation mechanism built into the approximation method itself. We are concerned only with discrete one step methods, that is methods in which  $u_{n+1}$  is determined solely from a knowledge of  $x_n$ ,  $u_n$  and the step  $h$  to proceed from  $x_n$  to  $x_{n+1} = x_n + h$ . In a  $k$ -step method ( $k > 1$ ) knowledge of  $k - 1$  additional points  $(x_{n-j}, u_{n-j})$ ,  $j = 1, 2, \dots, k - 1$ , is required to advance the solution.

When describing a single step of a one-step method, it suffices to show how one proceeds from a generic point  $(x, y)$ ,  $x \in [a, b]$ ,  $y \in \mathbb{R}^d$  to the “next” point  $(x + h, y_{next})$ . We refer to this as the *local description* of the one-step method. This also includes a discussion of the local accuracy, that is how closely  $y_{next}$  agrees at  $x + h$  with the solution of the differential equation. A one-step method solving the initial value problem (9.1.1) effectively generates a grid function  $\{u_n\}$ ,  $u_n \in \mathbb{R}^d$ , on a grid  $a = x_0 < x_1 < x_2 < \dots < x_{N-1} < x_N = b$  covering the interval  $[a, b]$ , whereby  $u_n$  is intended to approximate the exact solution  $y(x)$  at  $x = x_n$ . The point  $(x_{n+1}, u_{n+1})$  is obtained from the point  $(x_n, u_n)$  by applying a one-step method with an appropriate step  $h_n = x_{n+1} - x_n$ . This is referred to as the *global description* of a one-step method. Questions of interest here are the behavior of the global error  $u_n - y(x_n)$ , in particular stability and convergence, and the choice of  $h_n$  to proceed from one grid point  $x_n$  to the next,  $x_{n+1} = x_n + h_n$ .

## 9.3 Local Description of One-Step Methods

Given a generic point  $x \in [a, b]$ ,  $y \in \mathbb{R}^d$ , we define a single step of a one step method by

$$y_{next} = y + h\Phi(x, y; h), \quad h > 0. \quad (9.3.1)$$

The function  $\Phi : [a, b] \times \mathbb{R}^d \times \mathbb{R}_+ \rightarrow \mathbb{R}^d$  may be thought as the approximate increment per unit step, or the approximate difference quotient, and it defines the method. Along with (9.3.1), we consider the solution  $u(t)$  of the differential equation (9.1.1) passing through the point  $(x, y)$ , that is, the local initial value problem

$$\begin{cases} \frac{du}{dt} = f(t, u) \\ u(t) = y \end{cases} \quad t \in [t, t + h] \quad (9.3.2)$$

We call  $u(t)$  the *reference solution*. The vector  $y_{next}$  in (9.3.1) is intended to approximate  $u(x + h)$ . How successfully is this done it is measured by the truncation error defined as follows.

**Definition 9.3.1.** *The truncation error of the method  $\Phi$  at the point  $(x, y)$  is defined by*

$$T(x, y; h) = \frac{1}{h}[y_{next} - u(x + h)]. \quad (9.3.3)$$

Thus the truncation error is a vector valued function of  $d + 2$  variables. Using (9.3.1) and (9.3.2), we can write for it alternatively,

$$T(x, y; h) = \Phi(x, y; h) - \frac{1}{h}[u(x + h) - u(x)], \quad (9.3.4)$$

showing that  $T$  is the difference between the approximate and exact increment per unit step.

**Definition 9.3.2.** *The method  $\Phi$  is called consistent if*

$$T(x, y; h) \rightarrow 0 \text{ as } h \rightarrow 0, \quad (9.3.5)$$

*uniformly for*  $(x, y) \in [a, b] \times \mathbb{R}^d$ .

By (9.3.4) and (9.3.3)) we have consistency if and only if

$$\Phi(x, y; 0) = f(x, y), \quad x \in [a, b], \quad y \in \mathbb{R}^d. \quad (9.3.6)$$

A finer description of local accuracy is provided by the next definition based on the notion of a local truncation error.

**Definition 9.3.3.** *The method  $\Phi$  is said to have order  $p$  if for some vector norm  $\|\cdot\|$ ,*

$$\|T(x, y; h)\| \leq Ch^p, \quad (9.3.7)$$

*uniformly on  $[a, b] \times \mathbb{R}^d$ , with a constant  $C$  not depending on  $x, y$  and  $h$ .*

We express briefly this property as

$$T(x, y; h) = O(h^p), \quad h \rightarrow 0. \quad (9.3.8)$$

Note that  $p > 0$  implies consistency. Usually,  $p \in \mathbb{N}^*$ . It is called the *exact order*, if (9.3.7) does not hold for any larger  $p$ .

**Definition 9.3.4.** *A function  $\tau : [a, b] \times \mathbb{R}^d \rightarrow \mathbb{R}^d$  that satisfies  $\tau(x, y) \not\equiv 0$  and*

$$T(x, y; h) = \tau(x, y)h^p + O(h^{p+1}), \quad h \rightarrow 0 \quad (9.3.9)$$

*is called the principal error function .*

The principal error function determines the leading term in the truncation error. The number  $p$  in (9.3.9) is the exact order of the method since  $\tau \not\equiv 0$ .

All the preceding definitions are made with the idea in mind that  $h > 0$  is a small number. Then the larger is  $p$ , the more accurate is the method.

## 9.4 Examples of One-Step Methods

Some of the oldest methods are motivated by simple geometric considerations based on the slope field defined by the right-hand side of the differential equation. This include the Euler and modified Euler methods. More accurate and sophisticated methods are based on Taylor expansion.

### 9.4.1 Euler's method

Euler proposed his method in 1768, in the early days of calculus. It consist of simply following the slope at the generic point  $(x, y)$  over an interval of length  $h$

$$y_{next} = y + hf(x, y). \quad (9.4.1)$$

(See Figure 9.1).

Thus,  $\Phi(x, y; h) = f(x, y)$  does not depend on  $h$  and by (9.3.6) the method is consistent. For the truncation error we have by (9.3.3)

$$T(x, y; h) = f(x, y) - \frac{1}{h}[u(x+h) - u(x)], \quad (9.4.2)$$

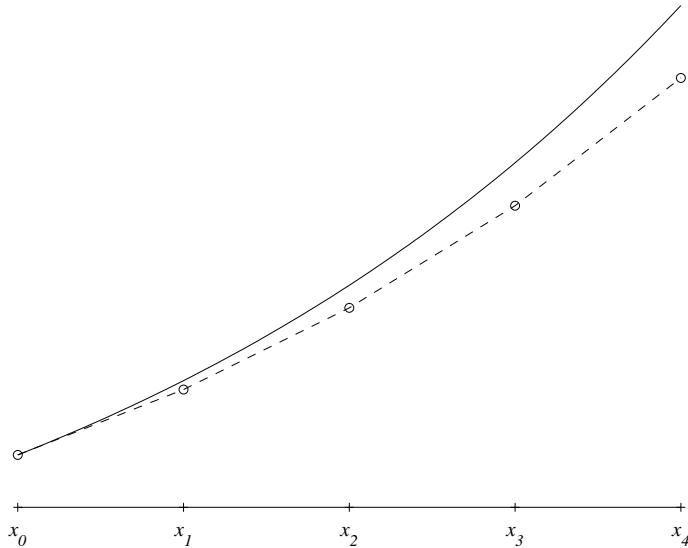


Figure 9.1: Euler's method – the exact solution (continuous line) and the approximate solution (dashed line)

where  $u(t)$  is the reference solution defined in (9.3.2). Since  $u' = f(x, u(x)) = f(x, y)$ , we can write, using Taylor's theorem,

$$\begin{aligned} T(x, y; h) &= u'(x) - \frac{1}{h}[u(x + h) - u(x)] \\ &= u'(x) - \frac{1}{h}[u(x) + hu'(x) + \frac{1}{2}h^2u''(\xi) - u(x)] \\ &= -\frac{1}{2}hu''(\xi), \quad \xi \in (x, x + h), \end{aligned} \tag{9.4.3}$$

assuming  $u \in C^r[x, x + h]$ . This is certainly true if  $f \in C^1([a, b] \times \mathbb{R}^d)$ . Now differentiating (9.3.2) totally with respect to  $t$  and then setting  $t = \xi$ , yields

$$T(x, y; h) = -\frac{1}{2}h[f_x + f_y f](\xi, u(\xi)), \tag{9.4.4}$$

where  $f_x$  is the partial derivative of  $f$  with respect to  $x$  and  $f_y$  the Jacobian of  $f$  with respect to the  $y$ -variables. If, in the spirit of Theorem 9.1.2, we assume that  $f$  and all its first partial derivatives are uniformly bounded in  $[a, b] \times \mathbb{R}^d$ , there exists a constant  $C$  independent of  $x, y$  and  $h$  such that

$$\|T(x, y; h)\| \leq Ch. \tag{9.4.5}$$

Thus, Euler's method has the order  $p = 1$ . If we make the same assumption about all second-order partial derivatives of  $f$  we have  $u''(\xi) = u''(x) + O(h)$  and, therefore from (9.4.3),

$$T(x, y; h) = -\frac{1}{2}h[f_x + f_y f](x, y) + O(h^2), \quad h \rightarrow 0, \tag{9.4.6}$$

showing that the principal error function is given by

$$\tau(x, y) = -\frac{1}{2}[f_x + f_y f](x, y). \quad (9.4.7)$$

Unless  $f_x + f_y f \equiv 0$ , the order of Euler's method is exactly  $p = 1$ .

### 9.4.2 Method of Taylor expansion

We have seen that Euler's method basically amounts to truncating the Taylor expansion of the reference solution after its second term. It is a natural idea, already proposed by Euler, to use more terms of the Taylor expansion. This requires the computation of successive "total derivatives" of  $f$ ,

$$\begin{aligned} f^{[0]}(x, y) &= f(x, y) \\ f^{[k+1]}(x, y) &= f_x^{[k]}(x, y) + f_y^{[k]}(x, y)f(x, y), \quad k = 0, 1, 2, \dots \end{aligned} \quad (9.4.8)$$

which determine the successive derivatives of the reference solution  $u(t)$  of (9.3.2) by virtue of

$$u^{(k+1)}(t) = f^{[k]}(t, u(t)), \quad k = 0, 1, 2, \dots \quad (9.4.9)$$

These, for  $t = x$ , become

$$u^{(k+1)}(x) = f^{[k]}(x, y), \quad k = 0, 1, 2, \dots \quad (9.4.10)$$

and are used to form the Taylor series approximation according to

$$y_{next} = y + h \left[ f^{[0]}(x, y) + \frac{1}{2}hf^{[1]}(x, y) + \dots + \frac{1}{p!}h^{p-1}f^{[p-1]}(x, y) \right], \quad (9.4.11)$$

that is,

$$\Phi(x, y; h) = f^{[0]}(x, y) + \frac{1}{2}hf^{[1]}(x, y) + \dots + \frac{1}{p!}h^{p-1}f^{[p-1]}(x, y). \quad (9.4.12)$$

For the truncation error, using (9.4.10) and (9.4.12) and assuming  $f \in C^p([a, b] \times \mathbb{R}^d)$ , we obtain from Taylor's theorem

$$\begin{aligned} T(x, y; h) &= \Phi(x, y; h) - \frac{1}{h}[u(x+h) - u(x)] = \\ &= \Phi(x, y; h) - \sum_{k=0}^{p-1} u^{(k+1)}(x) \frac{h^k}{(k+1)!} - u^{(p+1)}(\xi) \frac{h^p}{(p+1)!} = \\ &= -u^{(p+1)}(\xi) \frac{h^p}{(p+1)!}, \quad \xi \in (x, x+h), \end{aligned}$$

so that

$$\|T(x, y; h)\| \leq \frac{C_p}{(p+1)!} h^p,$$

where  $C_p$  is a bound on the  $p$ th total derivative of  $f$ . Thus, the method has the exact order  $p$  (unless  $f^{[p]}(x, y) \equiv 0$ ), and the principal error function is

$$\tau(x, y) = -\frac{1}{(p+1)!} f^{[p]}(x, y). \quad (9.4.13)$$

The necessity of computing many partial derivatives in (9.4.8) was a discouraging factor in the past, when this had to be done by hand. But nowadays, this task can be delegated to the computer, so that the method has become again a viable option.

### 9.4.3 Improved Euler methods

There is too much inertia in Euler's method: one should not follow the same initial slope over the whole interval of length  $h$ , since along this line segment the slope defined by the slope field of the differential equation changes. This suggests several alternatives. For example, we may wish to reevaluate the slope halfway through the line segment — retake the pulse of the differential equation, as it were — and then follow this revised slope over the whole interval (cf. Figure 9.2). In formula,

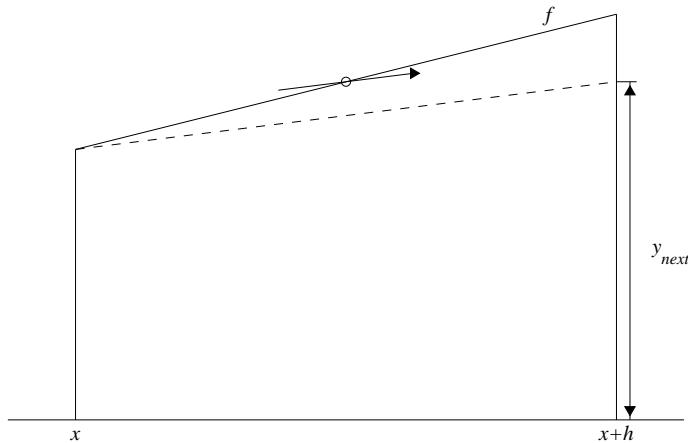


Figure 9.2: Modified Euler method

$$y_{\text{next}} = y + h f \left( x + \frac{1}{2}h, y + \frac{1}{2}h f(x, y) \right) \quad (9.4.14)$$

or

$$\Phi(x, y; h) = f \left( x + \frac{1}{2}h, y + \frac{1}{2}h f(x, y) \right) \quad (9.4.15)$$

Note the characteristic “nesting” of  $f$  that is required here. For programming purpose it may be desirable to undo the nesting and write

$$\begin{aligned} K_1(x, y) &= f(x, y) \\ K_2(x, y; h) &= f \left( x + \frac{1}{2}h, y + \frac{1}{2}h K_1 \right) \\ y_{\text{next}} &= y + h K_2 \end{aligned} \quad (9.4.16)$$

In other words, we are taking two trial slopes,  $K_1$  and  $K_2$ , one at the initial point and the other nearby, and then taking the latter as the final slope. The method is called *modified Euler method*.

We could equally well take the second trial slope at  $(x + h, y + h f(x, y))$ , but then, having waiting too long before reevaluating the slope, take now as the final slope the average of two slopes:

$$\begin{aligned} K_1(x, y) &= f(x, y) \\ K_2(x, y; h) &= f(x + h, y + h K_1) \\ y_{\text{next}} &= y + \frac{1}{2}h(K_1 + K_2). \end{aligned} \quad (9.4.17)$$

This is sometimes referred to as *Heun method*. The effect of both modifications is to raise the order by 1, as shown in the sequel.

## 9.5 Runge-Kutta Methods

We look for  $\Phi$  of the form:

$$\begin{aligned}\Phi(x, y; h) &= \sum_{s=1}^r \alpha_s K_s \\ K_1(x, y) &= f(x, y) \\ K_s(x, y) &= f\left(x + \mu_s h, y + h \sum_{j=1}^{s-1} \lambda_{sj} K_j\right), \quad s = 2, 3, \dots, r\end{aligned}\tag{9.5.1}$$

It is natural in (9.5.1) to impose the conditions

$$\mu_s = \sum_{j=1}^{s-1} \lambda_{sj}, \quad s = 2, 3, \dots, r, \quad \sum_{s=1}^r \alpha_s = 1,\tag{9.5.2}$$

where the first set is equivalent to

$$K_s(x, y; h) = u'(x + \mu_s h) + O(h^2), \quad s \geq 2,$$

and the second is nothing but the consistency condition (cf. (9.3.6)) (i.e.  $\Phi(x, y; h) = f(x, y)$ ).

We call (9.5.1) an *explicit r-stage Runge-Kutta method*; it requires  $r$  evaluation of the right-hand side  $f$  of the differential equation. Conditions (9.5.2) lead to a nonlinear system. Let  $p^*(r)$  the maximum attainable order (for arbitrary sufficient smooth  $f$ ) of an explicit  $r$ -stage Runge-Kutta method. Kutta<sup>2</sup> has shown in 1901 that

$$p^*(r) = r, \quad r = \overline{1, 4}.$$

We can consider *implicit r-stage Runge-Kutta methods*

$$\begin{aligned}\Phi(x, y; h) &= \sum_{s=1}^r \alpha_s K_s(x, y; h), \\ K_s &= f\left(x + \mu_s h, y + \sum_{j=1}^r \lambda_{sj} K_j\right), \quad s = \overline{1, r},\end{aligned}\tag{9.5.3}$$

in which the last  $r$  equations form a system of (in general nonlinear) equations in the unknowns  $K_1, K_2, \dots, K_r$ . Since each of these is a vector in  $\mathbb{R}^d$ , before we can form the approximate increment  $\Phi$

<sup>2</sup>



Wilhelm Martin Kutta (1867-1944) was a German applied mathematician. It is well-known for his work on the numerical solution of ODE. He had important contributions on application of conformal mapping to hydro- and aerodynamical problems (Kutta-Joukowski formula for the lift exerted on airfoil).

we must solve a system of  $rd$  equations in  $rd$  unknowns. *Semi-implicit Runge-Kutta methods*, where the summation in the formula for  $K_s$  extends from  $j = 1$  to  $j = s$ , require less work. This yields  $r$  systems of equations, each having only  $d$  unknowns, the components of  $K_s$ .

Already in the case of *explicit* Runge-Kutta methods, and even more so in implicit methods, we have at our disposal a large number of parameters which we can choose to achieve the maximum possible order for all sufficiently smooth  $f$ . The considerable computational expenses involved in implicit and semi-implicit methods can only be justified in special circumstances, for example, stiff problems. The reason is that implicit methods can be made not only to have higher order than explicit methods, but to have also better stability properties.

**Example 9.5.1.** Let

$$\Phi(x, y; h) = \alpha_1 K_1 + \alpha_2 K_2, \quad \diamond$$

where

$$\begin{aligned} K_1(x, y) &= f(x, y), \\ K_2(x, y; h) &= f(x + \mu_2 h, y + \lambda_{21} h K_1), \\ \lambda_{21} &= \mu_2. \end{aligned}$$

We have now three parameters,  $\alpha_1$ ,  $\alpha_2$ , and  $\mu$ . A systematic way of determining the maximum order  $p$  is to expand both  $\Phi(x, y; h)$  and  $h^{-1}[u(x + h) - u(x)]$  in powers of  $h$  and to match as many terms as we can, without imposing constraints on  $f$ .

To expand  $\Phi$ , we need Taylor's expansion for (vector-valued) functions of several variables

$$\begin{aligned} f(x + \Delta x, y + \Delta y) &= f + f_x \Delta x + f_y \Delta y + \\ &+ \frac{1}{2} [f_{xx}(\Delta x)^2 + 2f_{xy}\Delta x \Delta y + (\Delta y)^T f_{yy}(\Delta y)] + \dots, \end{aligned} \quad (9.5.4)$$

where  $f_y$  denotes the Jacobian of  $f$ , and  $f_{yy} = [f_{yy}^i]$  is the vector of Hessian matrices of  $f$ . In (9.5.4), all functions and partial derivatives are understood to be evaluated at  $(x, y)$ . Letting  $\Delta x = \mu h$ ,  $\Delta y = \mu h f$  then gives

$$\begin{aligned} K_2(x, y; h) &= f + \mu h(f_x + f_y f) \\ &+ \frac{1}{2} \mu^2 h^2 (f_{xx} + 2f_{xy} f + f^T f_{yy} f) + O(h^3), \end{aligned} \quad (9.5.5)$$

$$\frac{1}{h} [u(x + h) - u(x)] = u'(x) + \frac{1}{2} h u''(x) + \frac{1}{6} h u'''(x) + O(h^3), \quad (9.5.6)$$

where

$$\begin{aligned} u'(x) &= f \\ u''(x) &= f^{[1]} = f_x + f_y f \\ u'''(x) &= f^{[2]} = f_x^{[1]} + f_y^{[1]} f = f_{xx} + f_x f_y f + f_y f_x + (f_{xy} + (f_y f)_y) f = \\ &= f_{xx} + 2f_{xy} f + f^T f_{yy} f + f_y (f_x + f_y) f, \end{aligned}$$

and where in the last equation we have used

$$(f_y f)_y f = f^T f_{yy} f + f_y^2 f$$

Now,

$$T(x, y; h) = \alpha_1 K_1 + \alpha_2 K_2 - \frac{1}{h} [u(x + h) - u(x)]$$

wherein we substitute the expansions (9.5.5) and (9.5.6). We find

$$\begin{aligned} T(x, y; h) &= (\alpha_1 + \alpha_2 - 1)f + \left(\alpha_2\mu - \frac{1}{2}\right)h(f_x + f_yf) + \\ &+ \frac{1}{2}h^2 \left[ \left(\alpha_2\mu^2 - \frac{1}{3}\right)(f_{xx} + 2f_{xy}f + f^T f_{yy}f) - \frac{1}{3}f_y(f_x + f_yf) \right] + O(h^3) \quad (9.5.7) \end{aligned}$$

We cannot enforce the condition that the  $h^2$  coefficient be zero without imposing severe restriction on  $f$ . Thus, the maximum order is 2 and we obtain it for

$$\begin{cases} \alpha_1 + \alpha_2 = 1 \\ \alpha_2\mu = \frac{1}{2} \end{cases}$$

The solution

$$\begin{aligned} \alpha_1 &= 1 - \alpha_2 \\ \mu &= \frac{1}{2\alpha_2} \end{aligned}$$

depends upon an arbitrary parameter,  $\alpha_2 \neq 0$ .

For  $\alpha_2 = 1$  we obtain modified Euler method, and for  $\alpha_2 = \frac{1}{2}$  the Heun's method.  
We shall mention the classical Runge-Kutta formula of order  $p = 4$ .

$$\begin{aligned} \Phi(x, y; h) &= \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1(x, y; h) &= f(x, y) \\ K_2(x, y; h) &= f\left(x + \frac{1}{2}h, y + \frac{1}{2}hK_1\right) \\ K_3(x, y; h) &= f\left(x + \frac{1}{2}h, y + \frac{1}{2}hK_2\right) \\ K_4(x, y; h) &= f(x + h, y + hK_3) \end{aligned} \quad (9.5.8)$$

When  $f$  does not depend on  $y$ , then (9.5.8) becomes the Simpson's formula. Runge's <sup>3</sup>idea was to generalize Simpson's quadrature formula to ordinary differential equations. He succeeded only partially; his formula had  $r = 4$  and  $p = 3$ . The method (9.5.8) was discovered by Kutta in 1901 through a systematic search.

The classical 4th order Runge-Kutta method for a grid of  $N + 1$  equally spaced points is given by MATLAB Source 9.1.

**Example 9.5.2.** Using 4th order Runge-Kutta method for the initial value problem

$$\begin{aligned} y' &= -y + t + 1, \quad t \in [0, 1] \\ y(0) &= 1, \end{aligned}$$

3



Carle David Tolm  Runge (1856-1927) was active in the famous G ttingen school of mathematics and is one of the pioneer of numerical mathematics. He is best known for the Runge-Kutta formula in ordinary differential equation, for which he provided the basic idea. He made also notable contributions to approximation theory in the complex plane.

**MATLAB Source 9.1** Classical 4th order Runge-Kutta method

---

```

function [t,w]=RK4(f,tspan,alpha,N)
%RK4 - classical Runge-Kutta method for equispaced nodes
%call [t,w]=RK4(f,tspan,alpha,N)
%f - right hand side function
%tspan - interval
%alpha - starting value(s)
%N - number of subintervals
%t - abscissas of solution
%w - ordinates of solution

tc=tspan(1); wc=alpha(:);
h=(tspan(end)-tspan(1))/N;
t=tc; w=wc';
for k=1:N
    K1=f(tc,wc);
    K2=f(tc+1/2*h,wc+1/2*h*K1);
    K3=f(tc+1/2*h,wc+1/2*h*K2);
    K4=f(tc+h, wc+h*K3);
    wc=wc+h/6*(K1+2*K2+2*K3+K4);
    tc=tc+h;
    t=[t;tc]; w=[w;wc'];
end

```

---

with  $h = 0.1$ ,  $N = 10$ , and  $t_i = 0.1i$  we obtain the results given in Table 9.1. The exact solution is  $y(t) = e^{-t} + t$ , and the calling sequence is

`[t,w]=rk4(@edex1,[0,1],1,10);`

The MATLAB function

```

function df=edex1(t,y)
df=-y+t+1;

```

defines the right-hand side. ◊

It is usual to associate to each  $r$ -stages Runge-Kutta method (9.5.3) the tableau

$$\begin{array}{c|cccc}
 \mu_1 & \lambda_{11} & \lambda_{12} & \dots & \lambda_{1r} \\
 \mu_2 & \lambda_{21} & \lambda_{22} & \dots & \lambda_{2r} \\
 \vdots & \vdots & \vdots & \dots & \vdots \\
 \mu_r & \lambda_{r1} & \lambda_{r2} & \dots & \lambda_{rr} \\
 \hline
 & \alpha_1 & \alpha_2 & \dots & \alpha_r
 \end{array}
 \quad \left( \text{in matrix form } \begin{array}{c|c}
 \mu & \Lambda \\
 \hline
 \alpha^T &
 \end{array} \right)$$

called *Butcher table*. For an explicit method  $\mu_1 = 0$  and  $\Lambda$  is upper triangular having a null main diagonal. We can associate to the first  $r$ -lines of a Butcher table a quadrature formula  $\int_0^{\mu_s} u(t) dt \approx \sum_{j=1}^r \lambda_{sj} u(\mu_j)$ ,  $s = \overline{1, r}$  and to the last line the quadrature formula  $\int_0^1 u(t) dt \approx \sum_{s=1}^r \alpha_s u(\mu_j)$ . If corresponding degrees of exactness are  $d_s = q_s - 1$ ,  $1 \leq s \leq r + 1$  ( $d_s = \infty$  if  $\mu_s = 0$  and all

$t_i$	Approximations	Exact values	Error
0.0	1	1	0
0.1	1.00483750000	1.00483741804	8.19640e-008
0.2	1.01873090141	1.01873075308	1.48328e-007
0.3	1.04081842200	1.04081822068	2.01319e-007
0.4	1.07032028892	1.07032004604	2.42882e-007
0.5	1.10653093442	1.10653065971	2.74711e-007
0.6	1.14881193438	1.14881163609	2.98282e-007
0.7	1.19658561867	1.19658530379	3.14880e-007
0.8	1.24932928973	1.24932896412	3.25617e-007
0.9	1.30656999120	1.30656965974	3.31459e-007
1.0	1.36787977441	1.36787944117	3.33241e-007

Table 9.1: Numerical results for Example 9.5.2

$\lambda_{sj} = 0$ ), then Peano's Theorem implies that in the representation of the remainder occurs the  $q_s$ th derivatives of  $u$  and setting  $u(t) = y'(x + th)$  one obtains

$$\frac{y(x + \mu_s h) - y(x)}{h} - \sum_{j=1}^r \lambda_{sj} y'(x + \mu_j h) = O(h^{q_s}), \quad s = \overline{1, r}$$

and

$$\frac{y(x + h) - y(x)}{h} - \sum_{s=1}^r \alpha_s y'(x + \mu_s h) = O(h^{q_r+1}).$$

For classical 4th order Runge-Kutta method (9.5.8) the Butcher table is:

0	0			
$\frac{1}{2}$	$\frac{1}{2}$	0		
$\frac{1}{2}$	0	$\frac{1}{2}$	0	
1	0	0	1	0
	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

MATLAB Source 9.2 is an implementation example for a Runge-Kutta method with constant step given the Butcher table. The last parameter of `Runge_Kutta` function is a function handle, and the corresponding function returns the entries  $\mu$ ,  $\lambda$  and  $\alpha$  of the table and the number  $r$  of method stages. For the initialization of classical forth-order Runge-Kutta method Butcher table see MATLAB Source 9.3.

The MATLAB code below solves the problem in Example 9.5.2.

```
>> [t2, w2] = Runge_Kutta(@edex1, [0, 1], 1, 10, @RK4tab);
```

We emphasize that functions `RK4` and `Runge_Kutta` work for both scalar ordinary differential equations and for systems.

## 9.6 Global Description of One-Step Methods

Global description of one-step methods is best done in terms of grid and grid functions.

---

**MATLAB Source 9.2** Implementation of a Runge-Kutta method with constant step given Butcher table
 

---

```

function [x,y,nfev]=Runge_Kutta(f,tspan,y0,N,BT)
%RUNGE_KUTTA - Runge-Kutta method with constant step
%call [t,y,nfev]=Runge_Kutta(f,tspan,y0,N,BT)
%f -right-hand side function
%tspan - interval [a,b]
%y0 - starting value(s)
%N - number of steps
%BT - function that provides Butcher table, call syntax
%      [lambda,alfa,mu,s] - s number of stages
%t -abscissas of solution
%y - ordinates of solution components
%nfev - number of function evaluation

[lambda,alfa,mu,r]=BT(); %initialize Butcher table
h=(tspan(end)-tspan(1))/N; %step length
xc=tspan(1); yc=y0(:);
x=xc; y=yc';
K=zeros(length(y0),r);
for k=1:N %RK iteration
  K(:,1)=f(xc,yc);
  for i=2:r
    K(:,i)=f(xc+mu(i)*h,yc+h*(K(:,1:i-1)*lambda(i,1:i-1)'));
  end
  yc=yc+h*(K*alfa);
  xc=xc+h; %prepare next iteration
  x=[x;xc]; y=[y;yc'];
end
if nargout==3
  nfev=r*N;
end
end

```

---

**MATLAB Source 9.3** Initialize Butcher table for RK4
 

---

```

function [a,b,c,s]=RK4tab
%RK4TAB - Butcher table for classical RK4
s=4;
a=zeros(s,s-1);
a(2:s,1:s-1)=[1/2,0,0; 0, 1/2,0; 0,0,1];
b=[1,2,2,1]'/6;
c=sum(a');

```

---

A *grid* on interval  $[a, b]$  is a set of points  $\{x_n\}_{n=0}^N$  such that

$$a = x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N = b, \quad (9.6.1)$$

with *grid lengths*  $h_n$  defined by

$$h_n = x_{n+1} - x_n, \quad n = 0, 1, \dots, N-1. \quad (9.6.2)$$

The *fineness* of grid is measured by

$$|h| = \max_{0 \leq n \leq N-1} h_n. \quad (9.6.3)$$

We shall use the letter  $h$  to denote the collection of lengths  $h = \{h_n\}$ . If  $h_1 = h_2 = \cdots = h_N = (b-a)/N$ , we call (9.6.1) a *uniform grid*, otherwise a *nonuniform grid*. Letter  $h$  is also used to designate the common grid length  $h = (b-a)/N$ . A vector-valued function  $v = \{v_n\}$ ,  $v_n \in \mathbb{R}^d$ , defined on the grid (9.6.1) is called a *grid function*. Thus,  $v_n$  is the value of  $v$  at the gridpoint  $x_n$ . Every function  $v(x)$  defined on  $[a, b]$  induces a grid function by restriction. We denote the set of grid functions on  $[a, b]$  by  $\Gamma_h[a, b]$ , and for each grid function  $v = \{v_n\}$  define its norm by

$$\|v\|_\infty = \max_{0 \leq n \leq N} \|v_n\|, \quad v \in \Gamma_h[a, b]. \quad (9.6.4)$$

A one-step method – indeed, any discrete-variable method – is a method producing a grid function  $u = \{u_n\}$  such that  $u \approx y$ , where  $y = \{y_n\}$  is the grid function induced by the exact solution  $y(x)$  of the initial value problem (9.1.1)

Let the method

$$\begin{aligned} x_{n+1} &= x_n + h_n \\ u_{n+1} &= u_n + h_n \Phi(x_n, u_n; h_n), \end{aligned} \quad (9.6.5)$$

where  $x_0 = a$ ,  $u_0 = y_0$ .

To bring up the analogy between (9.1.1) and (9.6.5), we introduce operators  $R$  and  $R_h$  acting on  $C^1[a, b]$  and  $\Gamma_h[a, b]$ , respectively. These are the *residual operators*

$$(Rv)(x) := v'(x) - f(x, v(x)), \quad v \in C^1[a, b] \quad (9.6.6)$$

$$(R_h v)_n := \frac{1}{h_n} (v_{n+1} - v_n) - \Phi(x_n, v_n; h_n), \quad n = 0, 1, \dots, N-1, \quad (9.6.7)$$

where  $v = \{v_n\} \in \Gamma_h[a, b]$ . (The grid function  $\{(R_h v)_n\}$  is not defined for  $n = N$ , but we may arbitrarily set  $(R_h v)_N = (R_h v)_{N-1}$ ). Then the initial value problem (9.1.1) and its discrete analogue (9.6.5) can be written transparently as

$$Ry = 0 \text{ on } [a, b], \quad y(a) = y_0 \quad (9.6.8)$$

$$R_h u = 0 \text{ on } [a, b], \quad u_0 = y_0 \quad (9.6.9)$$

Note that the discrete residual operator (9.6.7) is closely related to the truncation error (9.3.3) when we apply the operator at a point  $(x_n, y(x_n))$  on the exact solution trajectory. Then indeed the reference solution  $u(t)$  coincides with the solution  $y(t)$  and

$$\begin{aligned} (R_h y)_n &= \frac{1}{h_n} [y(x_{n+1}) - y(x_n)] - \Phi(x_n, y(x_n); h_n) = \\ &= -T(x_n, y(x_n); h_n). \end{aligned} \quad (9.6.10)$$

### 9.6.1 Stability

Stability is a property of the numerical scheme (9.6.5) alone and has nothing to do with its approximation power. It characterizes the robustness of the scheme with respect to small perturbations. Nevertheless, stability combined with consistency yields convergence of the numerical solution to the true solution.

We define stability in terms of the discrete residual operators  $R_h$  in (9.6.7). As usual we assume  $\Phi(x, y; h)$  to be defined on  $[a, b] \times \mathbb{R}^d \times [0, h_0]$ , where  $h_0 > 0$  is some suitable positive number.

**Definition 9.6.1.** *The method (9.6.5) is called stable on  $[a, b]$  if there exists a constant  $K > 0$  not depending on  $h$  such that for an arbitrary grid  $h$  on  $[a, b]$ , and for two arbitrary grid functions  $v, w \in \Gamma_h[a, b]$ , there holds*

$$\|v - w\|_\infty \leq K (\|v_0 - w_0\|_\infty + \|R_h v - R_h w\|_\infty), \quad v, w \in \Gamma_h[a, b] \quad (9.6.11)$$

for all  $h$  with  $|h|$  sufficiently small. In (9.6.11) the norm is defined by (9.6.4).

We refer to (9.6.11) as the *stability inequality*. The motivation for it is as follows. Suppose we have two grid functions  $u, w$  satisfying

$$R_h u = 0, \quad u_0 = y_0 \quad (9.6.12)$$

$$R_h w = \varepsilon, \quad w_0 = y_0 + \eta_0, \quad (9.6.13)$$

where  $\varepsilon = \{\varepsilon_n\} \in \Gamma_h[a, b]$  is a grid function with small  $\|\varepsilon_n\|$ , and  $\|\eta_0\|$  is also small. We may interpret  $u \in \Gamma_h[a, b]$  as the result of applying the numerical scheme in (9.6.5) in infinite precision, whereas  $w \in \Gamma_h[a, b]$  could be the solution of (9.6.5) in floating-point arithmetic. Then, if stability holds, we have

$$\|u - w\|_\infty \leq K(\|\eta_0\|_\infty + \|\varepsilon\|_\infty), \quad (9.6.14)$$

that is, the global change in  $u$  is of the same order of magnitude as the local residual errors  $\{\varepsilon_n\}$  and initial error  $\eta_0$ . It should be appreciated, however that the first equations in (9.6.13) says

$$w_{n+1} - w_n - h_n \Phi(x_n, w_n, h_n) = h_n \varepsilon_n,$$

meaning that rounding errors must go to zero as  $|h| \rightarrow \infty$ .

Interestingly enough, a Lipschitz condition on  $\Phi$  is all that is required for stability.

**Theorem 9.6.2.** *If  $\Phi(x, y; h)$  satisfies a Lipschitz condition with respect to the  $y$ -variables*

$$\|\Phi(x, y; h) - \Phi(x, y^*; h)\| \leq M \|y - y^*\| \text{ on } [a, b] \times \mathbb{R}^d \times [0, h_0], \quad (9.6.15)$$

then the method (9.6.5) is stable.

For the proof we need the following lemma.

**Lemma 9.6.3.** *Let  $\{e_n\}$  be a sequence of numbers  $e_n \in \mathbb{R}$ , satisfying*

$$e_{n+1} \leq a_n e_n + b_n, \quad n = 0, 1, \dots, N-1 \quad (9.6.16)$$

where  $a_n > 0$  and  $b_n \in \mathbb{R}$ . Then

$$e_n \leq E_n, \quad E_n = \left( \prod_{k=0}^{n-1} a_k \right) e_0 + \sum_{k=0}^{n-1} \left( \prod_{l=k+1}^{n-1} a_l \right) b_k, \quad n = 0, 1, \dots, N \quad (9.6.17)$$

We adopt here the usual convention that an empty product has the value 1 and an empty sum has the value 0.

*Proof of lemma 9.6.3.* It is readily verified that

$$E_{n+1} = a_n E_n + b_n, \quad n = 0, 1, \dots, N-1, \quad E_0 = e_0.$$

Subtracting this from (9.6.16), we get

$$e_{n+1} - E_{n+1} \leq a_n(e_n - E_n), \quad n = 0, 1, \dots, N-1.$$

Now,  $e_0 - E_0 = 0$ , so that  $e_1 - E_1 \leq 0$ , since  $a_0 > 0$ . By induction, more generally,  $e_n - E_n \leq 0$ , since  $a_{n-1} > 0$ .  $\square$

*Proof of Theorem 9.6.2.* Let  $h = \{h_n\}$  be an arbitrary grid on  $[a, b]$  and  $v, w \in \Gamma_h[a, b]$  two arbitrary (vector-valued) grid functions. By definitions of  $R_h$ , we can write

$$v_{n+1} = v_n + h_n \Phi(x_n, v_n; h_n) + h_n(R_h v)_n, \quad n = 0, 1, \dots, N-1$$

and similarly for  $w_{n+1}$ . Subtracting then gives

$$\begin{aligned} v_{n+1} - w_{n+1} &= v_n - w_n + h_n[\Phi(x_n, v_n; h_n) - \Phi(x_n, w_n; h_n)] + \\ &\quad + h_n[(R_h v)_n - (R_h w)_n], \quad n = 0, 1, \dots, N-1. \end{aligned} \quad (9.6.18)$$

Define now

$$e_n = \|v_n - w_n\|, \quad d_n = \|(R_h v)_n - (R_h w)_n\|, \quad \delta = \|d_n\|_\infty. \quad (9.6.19)$$

Then, using the triangle inequality in (9.6.18) and the Lipschitz condition (9.6.19) for  $\Phi$ , we obtain

$$e_{n+1} \leq (1 + h_n M) e_n + h_n \delta, \quad n = 0, 1, \dots, N-1 \quad (9.6.20)$$

This is inequality (9.6.16) with  $a_n = 1 + h_n M$ ,  $b_n = h_n \delta$ . Since for  $k = 0, 1, \dots, n-1, n \leq N$  we have

$$\begin{aligned} \prod_{\ell=k+1}^{n-1} a_\ell &\leq \prod_{\ell=0}^{n-1} a_\ell = \prod_{\ell=0}^{N-1} (1 + h_\ell M) \leq \prod_{\ell=0}^{N-1} e^{h_\ell M} \\ &= e^{(h_0 + h_1 + \dots + h_{N-1})M} = e^{(b-a)M}, \end{aligned}$$

where the classical result  $1 + x \leq e^x$  has been used in the second inequality, we obtain from lemma 9.6.3 that

$$\begin{aligned} e_n &\leq e^{(b-a)M} e_0 + e^{(b-a)M} \sum_{k=0}^{n-1} h_k \delta \leq \\ &\leq e^{(b-a)M} (e_0 + (b-a)\delta), \quad n = 0, 1, \dots, N-1. \end{aligned}$$

Therefore

$$\|e\|_\infty = \|v - w\|_\infty \leq e^{(b-a)M} (\|v_0 - w_0\| + (b-a)\|R_h v - R_h w\|_\infty),$$

which is (9.6.11) with  $K = e^{(b-a)M} \max\{1, b-a\}$ .  $\square$

We have actually proved stability for all  $|h| \leq h_0$ , not only for  $h$  sufficiently small.

All one-step methods used in practice satisfy a Lipschitz condition if  $f$  does, and the constant  $M$  for  $\Phi$  can be expressed in terms of the Lipschitz constant  $L$  for  $f$ . This is obvious for Euler's method, and not difficult to prove for others. It is useful to note that  $\Phi$  does not need to be continuous in  $x$ ; piecewise continuity suffices, as long as (9.6.15) holds for all  $x \in [a, b]$ , taking one side limits at points of discontinuity.

The following application of Lemma 9.6.3, relative to a grid function  $v \in \Gamma_h[a, b]$  satisfying

$$v_{n+1} = v_n + h_n(A_n v_n + b_n), \quad n = 0, 1, \dots, N-1, \quad (9.6.21)$$

where  $A_n \in \mathbb{R}^{d \times d}$ ,  $b_n \in \mathbb{R}^d$ , and  $h_n$  is an arbitrary grid on  $[a, b]$  is also useful.

**Lemma 9.6.4.** *Suppose in (9.6.21) that*

$$\|A_n\| \leq M, \quad \|b_n\| \leq \delta, \quad n = 0, 1, \dots, N-1, \quad (9.6.22)$$

where the constants  $M, \delta$  do not depend on  $h$ . Then, there exists a constant  $K > 0$  independent of  $h$ , but depending on  $\|v_0\|$ , such that

$$\|v\|_\infty \leq K. \quad (9.6.23)$$

*Proof.* The lemma follows observing that

$$\|v_{n+1}\| \leq (1 + h_n M) \|v_n\| + h_n \delta, \quad n = 0, 1, \dots, N-1,$$

which is precisely the inequality (9.6.19) in the proof of Theorem 9.6.2, hence

$$\|v_n\| \leq e^{(b-a)M} \{\|v_0\| + (b-a)\delta\}. \quad (9.6.24)$$

□

## 9.6.2 Convergence

Stability is a powerful concept. It implies almost immediately convergence, and it is also instrumental in deriving asymptotic global error estimates. We begin by defining precisely what we mean by convergence.

**Definition 9.6.5.** Let  $a = x_0 < x_1 < x_2 < \dots < x_N = b$  be a grid on  $[a, b]$  with grid length  $|h| = \max_{1 \leq n \leq N} (x_n - x_{n-1})$ . Let  $u = \{u_n\}$  be the grid function defined by applying the method (9.6.5) on  $[a, b]$  and  $y = \{y_n\}$  the grid function induced by the exact solution of the initial value problem (9.1.1). The method (9.6.5) is said to converge on  $[a, b]$  if there holds

$$\|u - y\|_\infty \rightarrow 0 \text{ as } |h| \rightarrow 0 \quad (9.6.25)$$

**Theorem 9.6.6.** If the method (9.6.5) is consistent and stable on  $[a, b]$ , then it converges. Moreover, if  $\Phi$  has order  $p$ , then

$$\|u - y\|_\infty = O(|h|^p) \text{ as } |h| \rightarrow 0. \quad (9.6.26)$$

*Proof.* By the stability inequality (9.6.11) applied to the grid functions  $v = h$  and  $w = y$  of Definition 9.6.5, we have for  $|h|$  sufficiently small

$$\|u - y\|_\infty \leq K(\|u_0 - y(x_0)\| + \|R_h u - R_h y\|_\infty) = K\|R_h y\| \quad (9.6.27)$$

since  $u_0 = y(x_0)$  and  $R_h u = 0$  by (9.6.5). But, by (9.6.10),

$$\|R_h y\|_\infty = \|T(\cdot, y; h)\|_\infty \quad (9.6.28)$$

where  $T$  is the truncation error of the method  $\Phi$ . By definition of consistency

$$\|T(\cdot, y; h)\|_\infty \rightarrow 0, \text{ as } |h| \rightarrow 0,$$

which proves the first part of the theorem. The second part follows immediately from (9.6.27) and (9.6.28), since order  $p$ , means, by definition that

$$\|T(\cdot, y; h)\|_\infty = O(|h|^p), \text{ as } |h| \rightarrow 0. \quad (9.6.29)$$

□

### 9.6.3 Asymptotics of global error

Since the principal error function describes the leading contribution of the local truncation error, it is of interest to identify the leading term in the global error  $u_n - y(x_n)$ . To simplify matters, we assume a constant grid length  $h$ , although it is not difficult to deal with a variable grid length of the form  $h_n = \vartheta(x_n)h$ , where  $\vartheta(x)$  is piecewise continuous and  $0 < \vartheta(x) < \theta$  for  $a \leq x \leq b$ . Thus, we consider our one-step method to have the form

$$\begin{aligned} x_{n+1} &= x_n + h \\ u_{n+1} &= u_n + h\Phi(x_n, u_n; h); \quad n = 0, 1, \dots, N-1 \\ x_0 &= a, \quad u_0 = y_0, \end{aligned} \quad (9.6.30)$$

defining a grid function  $u = \{u_n\}$  on a uniform grid on  $[a, b]$ . We are interested in the asymptotic behavior of  $u_n - y(x_n)$  as  $h \rightarrow 0$ , where  $y(x)$  is the exact solution of the initial value problem

$$\begin{cases} \frac{dy}{dx} = f(x, y) & x \in [a, b] \\ y(a) = y_0 \end{cases} \quad (9.6.31)$$

**Theorem 9.6.7.** Assume that

- (1)  $\Phi(x, y, h) \in C^2([a, b] \times \mathbb{R}^d \times [0, h_0])$ ;
- (2)  $\Phi$  is a method of order  $p \geq 1$  admitting a principal error function  $\tau(x, y)$  continuous on  $[a, b] \times \mathbb{R}^d$ ;
- (3)  $e(x)$  is the solution of the linear initial value problem

$$\begin{cases} \frac{de}{dx} = f_y(x, y(x))e + \tau(x, y(x)), & a \leq x \leq b \\ e(a) = 0 \end{cases} \quad (9.6.32)$$

Then, for  $n = \overline{0, N}$ ,

$$u_n - y(x_n) = e(x_n)h^p + O(h^{p+1}), \quad \text{as } h \rightarrow 0. \quad (9.6.33)$$

Before we prove the theorem, we make the following remarks:

1. The precise meaning of (9.6.33) is

$$\|u - y - h^p e\|_\infty = O(h^{p+1}),$$

where  $u, y, e$  are the grid functions  $u = \{u_n\}$ ,  $y = \{y(x_n)\}$  and  $e = \{e(x_n)\}$ .

2. Since by consistency  $\Phi(x, y; 0) = f(x, y)$ , assumption (1) implies  $f$  is of class  $C^2$  on  $([a, b] \times \mathbb{R}^d)$ , which is more than enough to guarantee the existence and uniqueness of the solution  $e(x)$  of (9.6.32) on the whole interval  $[a, b]$ .
3. The fact that some, but not all, components of  $\tau(x, y)$  may vanish identically does not imply that the corresponding components of  $e(x)$  also vanish, since (9.6.32) is a *coupled* system of differential equations.

*Proof of Theorem 9.6.7.* We begin with an auxiliary computation, an estimate for

$$\Phi(x_n, u_n; h) - \Phi(x_n, y(x_n); h). \quad (9.6.34)$$

By Taylor's (for functions of several variables), applied to the  $i$ th component of (9.6.34), we have

$$\begin{aligned} \Phi^i(x_n, u_n; h) - \Phi^i(x_n, y(x_n); h) &= \sum_{j=1}^d \Phi^i y^j(x_n, y(x_n); h) [u_n^j - y^j(x_n)] \\ &\quad + \frac{1}{2} \sum_{j,k=1}^d \Phi^i y^j y^k(x_n, \bar{u}_n; h) [u_n^j - y^j(x_n)] [u_n^k - y^k(x_n)], \end{aligned} \quad (9.6.35)$$

where  $\bar{u}_n$  is on the line segment connecting  $u_n$  and  $y(x_n)$ . Using Taylor's theorem once more, in the variable  $h$ , we can write

$$\Phi_{y^j}^i(x_n, y(x_n); h) = \Phi_{y^j}^i(x_n, y(x_n); 0) + h \Phi_{y^j h}^i(x_n, y(x_n); \bar{h}),$$

where  $0 < \bar{h} < h$ . Since, by consistency,  $\Phi(x, y; 0) \equiv f(x, y)$  on  $[a, b] \times \mathbb{R}^d$ , we have

$$\Phi_{y^j}^i(x, y; 0) = f_{y^j}^i(x, y), \quad x \in [a, b], \quad y \in \mathbb{R}^d,$$

and assumption (1) allows us to write

$$\Phi_{y^j}^i(x_n, y(x_n); h) = f_{y^j}^i(x_n, y(x_n)) + O(h), \quad h \rightarrow 0. \quad (9.6.36)$$

Now observing that  $u_n - y(x_n) = O(h^p)$ , by virtue of Theorem 9.6.6 and using (9.6.36) in (9.6.35), we get, again by assumption (1),

$$\begin{aligned} \Phi^i(x_n, u_n; h) - \Phi^i(x_n, y(x_n); h) &= \sum_{j=1}^d f_{y^j}^i(x_n, y(x_n)) [u_n^j - y^j(x_n)] + \\ &\quad O(h^{p+1}) + O(h^{2p}). \end{aligned}$$

But  $O(h^{2p})$  is also of order  $O(h^{p+1})$ , since  $p \geq 1$ . Thus, in vector notation,

$$\Phi(x_n, u_n; h) - \Phi(x_n, y(x_n); h) = f_y(x_n, y(x_n)) [u_n - y(x_n)] + O(h^{p+1}). \quad (9.6.37)$$

Now, to highlight the leading term in the global error, we define the grid function  $r = \{r_n\}$  by

$$r = h^{-p}(u - y). \quad (9.6.38)$$

Then

$$\begin{aligned} \frac{1}{h}(r_{n+1} - r_n) &= \frac{1}{h} [h^{-p}(u_{n+1} - y(x_{n+1})) - h^{-p}(u_n - y(x_n))] = \\ &= h^p \left[ \frac{1}{h}(u_{n+1} - u_n) - \frac{1}{h}(y(x_{n+1}) - y(x_n)) \right] = \\ &= h^{-p} \{\Phi(x_n, u_n; h) - [\Phi(x_n, y(x_n); h) - T(x_n, y(x_n); h)]\}, \end{aligned}$$

where we have used (9.6.30) and the relation (9.6.10) for the truncation error  $T$ . Therefore, expressing  $T$  in terms of the principal error function  $\tau$ , we get

$$\begin{aligned} \frac{1}{h}(r_{n+1} - r_n) &= h^{-p} [\Phi(x_n, u_n; h) - \Phi(x_n, y(x_n); h) + \tau(x_n, y(x_n))h^p \\ &\quad + O(h^{p+1})] \end{aligned}$$

For the first two terms in brackets we use (9.6.37) and the definition of  $r$  in (9.6.38) to obtain

$$\begin{aligned} \frac{1}{h}(r_{n+1} - r_n) &= f_y(x_n, y(x_n))r_n + \tau(x_n, y(x_n)) + O(h), \quad n = \overline{0, N-1} \\ r_0 &= 0. \end{aligned} \tag{9.6.39}$$

Now letting

$$g(x, y) := f_y(x, y(x))y + \tau(x, y(x)) \tag{9.6.40}$$

we can interpret (9.6.39) by writing

$$\left( R_h^{Euler, g} r \right)_n = \varepsilon_n, \quad n = \overline{0, N-1}, \quad \varepsilon_n = O(h),$$

where  $R_h^{Euler, g}$  is the discrete residual operator (9.6.7) that goes with Euler's method applied to  $e' = g(x, e)$ ,  $e(a) = 0$ . Since Euler's method is stable on  $[a, b]$  and  $g$  being linear in  $y$  satisfies a uniform Lipschitz condition, we have by the stability inequality (9.6.11)

$$\|r - e\|_\infty = O(h),$$

and hence, by (9.6.38)

$$\|u - y - h^p e\|_\infty = O(h^{p+1}),$$

as was to be shown.  $\square$

## 9.7 Error Monitoring and Step Control

Most production codes currently available for solving ODEs monitor local truncation errors and control the step length on the basis of estimates for these errors. Here we attempt to monitor global error, at least asymptotically, by implementing the asymptotic result of Theorem 9.6.7. This necessitates the evaluation of the Jacobian matrix  $f_y(x, y)$  along or near the solution trajectory; but this is only natural, since  $f_y$ , in a first approximation, governs the effect of perturbations via the variational differential equation (9.6.32). This equation is driven by the principal error function evaluated along the trajectory, so that estimates of local truncation errors (more precisely, of the principal error function) are needed also in this approach. For simplicity we again assume constant grid length.

### 9.7.1 Estimation of global error

The idea of our estimation is to integrate the “variational equation” (9.6.32) along with the main equation (9.6.31). Since we need  $e(x)$  in (9.6.31) only to within an accuracy of  $O(h)$  (any  $O(h)$  error term in  $e(x_n)$ , multiplied by  $h^p$ , being absorbed by the  $O(h^{p-1})$  term), we can use Euler's method for that purpose, which will provide the desired approximation  $v_n \approx e(x_n)$ .

**Theorem 9.7.1.** *Assume that*

- (1)  $\Phi(x, y; h) \in C^2([a, b] \times \mathbb{R}^d \times [0, h_0])$ ;

- (2)  $\Phi$  is a method of order  $p \geq 1$  admitting a principal error function  $\tau(x, y)$  of class  $C^1([a, b] \times \mathbb{R}^d)$ ;  
(3) an estimate  $r(x, y; h)$  is available for principal error function that satisfies

$$r(x, y; h) = \tau(x, y) + O(h), \quad h \rightarrow 0, \quad (9.7.1)$$

uniformly on  $[a, b] \times \mathbb{R}^d$ ;

- (4) along with the grid function  $u = \{u_n\}$  we generate the grid function  $v = \{v_n\}$  in the following manner.

$$\begin{aligned} x_{n+1} &= x_n + h; \\ u_{n+1} &= u_n + h\Phi(x_n, u_n; h) \\ v_{n+1} &= v_n + h [f_y(x_n, v_n)v_n + r(x_n, u_n; h)] \\ x_0 &= a, \quad u_0 = y_0, \quad v_0 = 0. \end{aligned} \quad (9.7.2)$$

Then, for  $n = \overline{0, N-1}$ ,

$$u_n - y(x_n) = v_n h^p + O(h^{p+1}), \quad \text{and } h \rightarrow 0. \quad (9.7.3)$$

*Proof.* The proof begins by establishing the following estimates

$$f_y(x_n, u_n) = +O(h), \quad (9.7.4)$$

$$r(x_n, u_n; h) = \tau(x_n, y(x_n)) + O(h). \quad (9.7.5)$$

From assumption (1) we note by consistency,  $f(x, y) = \Phi(x, y; 0)$  that  $f(x, y)$  is in  $C^2([a, b] \times \mathbb{R}^d)$ . Taking into account the Theorem 9.6.6, we have  $u_n = y(x_n) + O(h^p)$ , and therefore,

$$f_y(x_n, u_n) = f_y(x_n, y_n) + O(h^p),$$

which implies (9.7.4), since  $p \geq 1$ . Next, since  $\tau(x, y) \in C^1([a, b] \times \mathbb{R}^d)$ , by assumption (2) we have

$$\begin{aligned} \tau(x_n, u_n) &= \tau(x_n, y(x_n)) + \tau_y(x_n, \bar{u}_n)(u_n - y(x_n)) \\ &= \tau(x_n, y(x_n)) + O(h^p) \end{aligned}$$

so that by assumption (3),

$$r(x_n, u_n; h) = \tau(x_n, u_n) + O(h) = \tau(x_n, y(x_n)) + O(h^p) + O(h),$$

which implies (9.7.5) immediately.

Let (cf. (9.6.40))

$$g(x, y) = f_y(x, y(x))y + \tau(x, y(x)). \quad (9.7.6)$$

The equation for  $v_{n+1}$  in (9.7.2) has the form

$$v_{n+1} = v_n + h(A_n v_n + b_n),$$

where  $A_n$  are bounded matrices and  $b_n$  bounded vectors. By Lemma 9.6.4, 9.6.4, we have boundness of  $v_n$ ,

$$v_n = O(1), \quad h \rightarrow 0. \quad (9.7.7)$$

Substituting (9.7.4) and (9.7.5) into the equation for  $v_{n+1}$  and noting (9.7.7), we obtain

$$\begin{aligned} v_{n+1} &= v_n + h [f_y(x_n, y(x_n))v_n + \tau(x_n, y(x_n)) + O(h)] \\ &= v_n + hg(x_n, v_n) + O(h^2). \end{aligned}$$

Thus, in the notation used in the proof of Theorem 9.6.7

$$\left( R_h^{Euler,g} v \right)_n = O(h), \quad v_0 = 0.$$

Since Euler's method is stable, we conclude

$$v_n - e(x_n) = O(h),$$

where  $e(x)$  is, as before, the solution of

$$\begin{aligned} e' &= g(x, e) \\ e(a) &= 0. \end{aligned}$$

Therefore, by (9.6.33)

$$u_n - y(x_n) = e(x_n)h^p + O(h^{p+1}).$$

□

## 9.7.2 Truncation error estimates

In order to apply Theorem 9.7.1 we need estimates  $r(x, y; h)$  of the principal error function  $\tau(x, y)$  which are  $O(h)$  accurate. We shall describe two of them in increasing order of efficiency.

### Local Richardson extrapolation to zero

This works for any one-step method  $\Phi$ , but is usually considered too expensive. If  $\Phi$  has the order  $p$ , the procedure is as follows

$$\begin{aligned} y_h &= y + h\Phi(x, y; h), \\ y_{h/2} &= y + \frac{1}{2}h\Phi\left(x, y; \frac{1}{2}h\right), \\ y_h^* &= y_{h/2} + \frac{1}{2}h\Phi\left(x + \frac{1}{2}h, y_{h/2}; \frac{1}{2}h\right), \\ r(x, y; h) &= \frac{1}{1 - 2^{-p}} \frac{1}{h^{p+1}} (y_h - y_h^*). \end{aligned} \tag{9.7.8}$$

Note that  $y_h^*$  is the result of applying  $\Phi$  over two consecutive steps of length  $h/2$  each, whereas  $y_h$  is the result of one application over the whole step length  $h$ .

We now verify that  $r(x, y; h)$  in (9.7.8) is an acceptable error estimator. To do this, we need to assume that  $\tau(x, y) \in C^1([a, b] \times \mathbb{R}^d)$ . In terms of the reference solution  $u(t)$  through  $(x, y)$  we have (cf. (9.3.4) and (9.3.8))

$$\Phi(x, y; h) = \frac{1}{h}[u(x+h) - u(x)] + \tau(x, y)h^p + O(h^{p+1}). \tag{9.7.9}$$

Furthermore,

$$\begin{aligned} \frac{1}{h}(y_h - y_h^*) &= \frac{1}{h}(y_h - y_{h/2}) + \Phi(x, y; h) - \frac{1}{2}h\Phi\left(x + \frac{1}{2}h, y_{h/2}; \frac{1}{2}h\right) \\ &= \Phi(x, y; h) - \frac{1}{2}\Phi\left(x, y; \frac{1}{2}h\right) - \frac{1}{2}h\Phi\left(x + \frac{1}{2}h, y_{h/2}; \frac{1}{2}h\right). \end{aligned}$$

Applying (9.7.9) to each of the three terms on the right, we find

$$\begin{aligned} \frac{1}{h}(y_h - y_h^*) &= \frac{1}{h}[u(x+h) - u(x)] + \tau(x, y)h^p + O(h^{p+1}) \\ &\quad - \frac{1}{2} \frac{1}{h/2} \left[ u\left(x + \frac{1}{2}h\right) - u(x) \right] - \frac{1}{2}\tau(x, y)\left(\frac{1}{2}h^p\right) + O(h^{p+1}) \\ &\quad - \frac{1}{2} \frac{1}{h/2} \left[ u(x+h) - u\left(x + \frac{1}{2}h\right) \right] - \frac{1}{2}\tau\left(x + \frac{1}{2}h, y + O(h)\right)\left(\frac{1}{2}h^p\right) \\ &\quad + O(h^{p+1}) = \tau(x, y)(1 - 2^{-p})h^p + O(h^{p+1}). \end{aligned}$$

Consequently

$$\frac{1}{1 - 2^{-p}} \frac{1}{h}(y_h - y_h^*) = \tau(x, y)h^p + O(h^{p+1}), \quad (9.7.10)$$

as required.

Subtracting (9.7.10) from (9.7.9) shows, incidentally that

$$\Phi^*(x, y; h) := \Phi(x, y; h) - \frac{1}{1 - 2^{-p}} \frac{1}{h}(y_h - y_h^*) \quad (9.7.11)$$

defines a one-step method of order  $p + 1$ .

Procedure (9.7.8) is rather expensive. For a fourth-order Runge-Kutta process, it requires a total of 11 evaluations of  $f$  per step, almost three times the effort for a single Runge-Kutta step. Therefore, Richardson extrapolation is normally used only after two steps of  $\Phi$ , that is one proceeds according to

$$\begin{aligned} y_h &= y + h\Phi(x, y; h), \\ y_{2h}^* &= y_h + h\Phi(x + h, y_h; h) \\ y_{2h} &= y + 2h\Phi(x, y; 2h). \end{aligned} \quad (9.7.12)$$

Then (9.7.10) gives

$$\frac{1}{2(2^p - 1)} \frac{1}{h^{p+1}} (y_{2h} - y_{2h}^*) = \tau(x, y) + O(h), \quad (9.7.13)$$

so that the expression on the left is an acceptable estimator  $r(x, y; h)$ . If the two steps in (9.7.12) yield acceptable accuracy (cf. §9.7.3), then again for a fourth-order Runge-Kutta process, the procedure requires only three additional evaluations of  $f$ , since  $y_h$  and  $y_{2h}^*$  would have to be computed anyhow. There are still more efficient schemes, as we shall seen.

## Embedded methods

The basic idea of this approach is very simple: if the given method  $\Phi$  has order  $p$ , take any other one step method  $\Phi^*$  of order  $p^* = p + 1$  and define

$$r(x, y; h) = \frac{1}{h^p} [\Phi(x, y; h) - \Phi^*(x, y; h)] \quad (9.7.14)$$

This is indeed an acceptable estimator, as follows by subtracting the two relations

$$\begin{aligned} \Phi(x, y; h) - \frac{1}{h}[u(x+h) - u(x)] &= \tau(x, y)h^p + O(h^{p+1}) \\ \Phi^*(x, y; h) - \frac{1}{h}[u(x+h) - u(x)] &= O(h^{p+1}) \end{aligned}$$

and dividing the result by  $h^p$ .

The tricky part is to make this procedure efficient. Following an idea of Fehlberg, one can try to do this by embedding one Runge-Kutta process (of order  $p$ ) into another (of order  $p + 1$ ). Specifically, let  $\Phi$  be some explicit  $r$ -stage Runge-Kutta method.

$$\begin{aligned} K_1(x, y) &= f(x, y) \\ K_s(x, y; h) &= f \left( x + \mu_s h; y + h \sum_{j=1}^{s-1} \lambda_{sj} K_j \right), \quad s = 2, 3, \dots, r \\ \Phi(x, y; h) &= \sum_{s=1}^r \alpha_s K_s \end{aligned}$$

Then for  $\Phi^*$  choose a similar  $r^*$ -stage process, with  $r^* > r$ , in such a way that

$$\mu_s^* = \mu_s, \quad \lambda_{sj}^* = \lambda_{sj}, \text{ for } s = 2, 3, \dots, r.$$

The estimate (9.7.14) then costs only  $r^* - r$  extra evaluations of  $f$ . If  $r^* = r + 1$  one might even attempt to save the additional evaluation by selecting (if possible)

$$\mu_r^* = 1, \quad \lambda_{rj}^* = \alpha_j \text{ for } j = \overline{1, r^* - 1} \quad (r^* = r + 1) \quad (9.7.15)$$

Then indeed,  $K_r^*$  will be identical with  $K_1$  for the next step.

Pairs of such embedded  $(p, p + 1)$  Runge-Kutta formulae have been developed in the late 1960's by E. Fehlberg. There is a considerable degree of freedom in choosing the parameters. Fehlberg's choices were guided by an attempt to reduce the magnitude of the coefficients of all the partial derivative aggregates that enter into the principal error function  $\tau(x, y)$  of  $\Phi$ . He succeeded in obtaining pairs with the following values of parameters  $p, r, r^*$ , given in Table 9.2.

$p$	3	4	5	6	7	8
$r$	4	5	6	8	11	15
$r^*$	5	6	8	10	13	17

Table 9.2: Embedded Runge-Kutta formulae

For the third-order process (and only for that one) one can choose the parameters for (9.7.15) to hold.

### 9.7.3 Step control

Any estimate  $r(x, y; h)$  of the principal error function  $\tau(x, y)$  implies an estimate

$$h^p r(x, y; h) = T(x, y; h) + O(h^{p+1}) \quad (9.7.16)$$

for the truncation error, which can be used to monitor the local truncation error during the integration process. However, one has to keep in mind that the local truncation error is quite different from the global error, that one really wants to control. To get more insight into the relationship between these two errors, we recall the following theorem, which quantifies the continuity of solution of an initial value problem with respect to initial values.

**Theorem 9.7.2.** Let  $f(x, y)$  be continuous in  $x \in [a, b]$  and satisfy a Lipschitz condition uniformly on  $[a, b] \times \mathbb{R}$ , with Lipschitz constant  $L$ , that is

$$\|f(x, y) - f(x, y^*)\| \leq L \|y - y^*\|.$$

Then the initial value problem

$$\begin{aligned} \frac{dy}{dx} &= f(x, y), \quad x \in [a, b], \\ y(c) &= y_c \end{aligned} \tag{9.7.17}$$

has a unique solution on  $[a, b]$  for any  $c \in [a, b]$  and for any  $y_c \in \mathbb{R}^d$ . Let  $y(x, s)$  and  $y(x; s^*)$  be the solutions of (9.7.17) corresponding to  $y_c = s$  and  $y_c = s^*$ , respectively. Then for any vector norm  $\|\cdot\|$ ,

$$\|y(x; s) - y(x; s^*)\| \leq e^{L|x-s|} \|s - s^*\|. \tag{9.7.18}$$

“Solving the given initial value problem (9.6.31) numerically by a one-step method (not necessarily with constant step) means in reality that one follows a sequence of “solution tracks”, whereby at each grid point  $x_n$  one jumps from one track to the next by an amount determined by the truncation error at  $x_n$ ” [33] (see Figure 9.3). This result from the definition of truncation error, the reference solution being one of the solution tracks. Specifically, the  $n$ th track,  $n = \overline{0, N}$ , is given by the solution of the initial value problem

$$\begin{aligned} \frac{dv_n}{dx} &= f(x, v_n), \quad x \in [x_n, b], \\ v_n(x_n) &= u_n, \end{aligned} \tag{9.7.19}$$

and

$$u_{n+1} = v(x_{n+1}) + h_n T(x_n, u_n; h_n), \quad n = \overline{0, N-1}. \tag{9.7.20}$$

Since by (9.7.19) we have  $u_{n+1} = v_{n+1}(x_{n+1})$ , we can apply Theorem 9.7.2 to the solution  $v_{n+1}$  and  $v_n$ , letting  $c = x_{n+1}$ ,  $s = u_{n+1}$ ,  $s^* = u_{n+1} - h_n T(x_n, u_n; h_n)$  (by (9.7.20)), and thus obtain

$$\|v_{n+1}(x) - v_n(x)\| \leq h_n e^{L|x-x_n|} \|T(x_n, u_n; h_n)\|, \quad n = \overline{0, N-1}. \tag{9.7.21}$$

Now

$$\sum_{n=0}^{N-1} [v_{n+1}(x) - v_n(x)] = v_N(x) - v_0(x) = v_N(x) - y(x), \tag{9.7.22}$$

and since  $v_N(x_N) = u_N$ , letting  $x = x_N$ , we get from (9.7.21) and (9.7.22) that

$$\begin{aligned} \|u_N - y(x_N)\| &\leq \sum_{n=0}^{N-1} \|v_{n+1}(x_N) - v_n(x_N)\| \\ &\leq \sum_{n=0}^{N-1} h_n e^{L|x_N-x_{n+1}|} \|T(x_n, u_n; h_n)\|. \end{aligned}$$

Therefore, if we make sure that

$$\|T(x_n, u_n; h_n)\| \leq \varepsilon_T, \quad n = \overline{0, N-1}, \tag{9.7.23}$$

then

$$\|u_N - y(x_N)\| \leq \varepsilon_T \sum_{n=0}^{N-1} (x_{n+1} - x_n) e^{L|x_N-x_{n+1}|}.$$

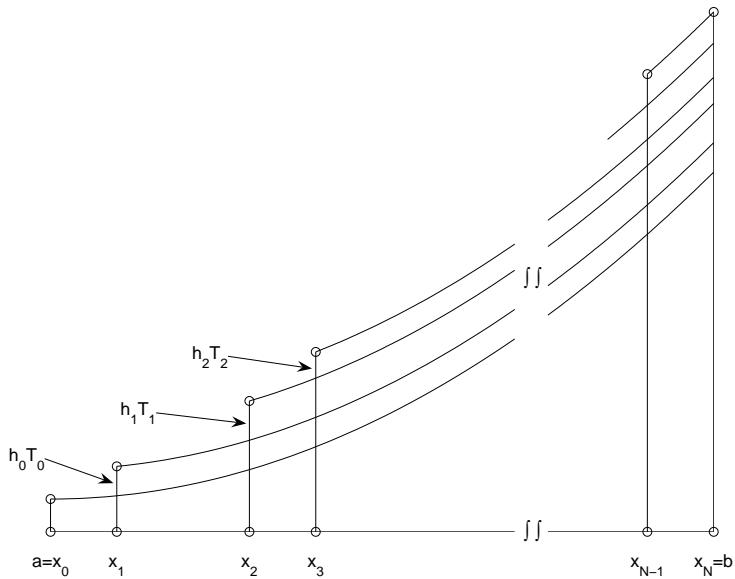


Figure 9.3: Error accumulation in a one-step method

Interpreting the sum on the right as a Riemann sum for a definite integral, we finally obtain, approximately,

$$\|u_N - y(x_N)\| \leq \varepsilon_T \int_a^b e^{L(b-x)} dx = \frac{\varepsilon_T}{L} (e^{L(b-a)} - 1).$$

Thus, knowing an estimate for  $L$  would allow us to set an appropriate  $\varepsilon_T$ , namely

$$\varepsilon_T = \frac{L}{e^{L(b-a)} - 1} \varepsilon, \quad (9.7.24)$$

to guarantee an error  $\|u_N - y(x_N)\| \leq \varepsilon$ . What holds for the whole grid on  $[a, b]$  of course, holds for any grid on a subinterval  $[a, x]$ ,  $a \leq x \leq b$ . So, in principle, given the desired accuracy  $\varepsilon$  for the solution  $y(x)$ , we can determine a “local tolerance level”  $\varepsilon_T$  (cf. (9.7.24)) and achieve the desired accuracy by keeping the local truncation error below  $\varepsilon_T$  (cf. (9.7.23)). Note that as  $L \rightarrow 0$ , we have  $\varepsilon_T \rightarrow \varepsilon/(b-a)$ . This limit value of  $\varepsilon_T$  would be appropriate for a quadrature problem but definitely not for a true differential equation problem, where  $\varepsilon_T$ , in general, has to be chosen considerably smaller than the target error tolerance  $\varepsilon$ .

Considerations such as these motivate the following *step control* mechanism: each integration step (from  $x_n$  to  $x_{n+1} = x_n + h_n$ ) consists of these parts:

1. Estimate  $h_n$ .
2. Compute  $u_{n+1} = u_n + h_n \Phi(x_n, u_n; h_n)$  and  $r(x_n, u_n; h_n)$ .
3. Test  $h_n^p \|r(x_n, u_n; h_n)\| \leq \varepsilon_T$  (cf. (9.7.16) and (9.7.23)). If the test passes, proceed with the next step; if not, repeat the step with a smaller  $h_n$ , say, half as large, until the test passes.

To estimate  $h_n$ , assume first that  $n \geq 1$ , so that the estimator from the previous step,  $r(x_{n-1}, u_{n-1}; h_{n-1})$  (or at least its norm) is available. Then, neglecting terms of  $O(h)$ ,

$$\|\tau(x_{n-1}, u_{n-1})\| \approx \|r(x_{n-1}, u_{n-1}; h_{n-1})\|,$$

and since  $\tau(x_n, u_n) \approx \tau(x_{n-1}, u_{n-1})$ , likewise

$$\|\tau(x_n, u_n)\| \approx \|r(x_{n-1}, u_{n-1}; h_{n-1})\|.$$

What we want is

$$\|\tau(x_n, u_n)\| h_n^p \approx \theta \varepsilon_T,$$

where  $\theta$  is “safety factor”, say,  $\theta = 0.8$ . Eliminating  $\tau(x_n, u_n)$ , we find

$$h_n \approx \left\{ \frac{\theta \varepsilon_T}{\|r(x_{n-1}, u_{n-1}; h_{n-1})\|} \right\}^{1/p}.$$

Note that from the previous step we have

$$h_{n-1}^p \|r(x_{n-1}, u_{n-1}; h_{n-1})\| \leq \varepsilon_T,$$

so that

$$h_n \geq \theta^{1/p} h_{n-1},$$

and the tendency is to increase the step.

If  $n = 0$ , we proceed similarly, using some initial guess  $h_0^{(0)}$  of  $h_0$  and associated  $r(x_0, y_0; h_0^{(0)})$  to obtain

$$h_0^{(1)} = \left\{ \frac{\theta \varepsilon_T}{r(x_0, y_0; h_0^{(0)})} \right\}^{1/p}.$$

The process may be repeated once or twice to get the final estimate of both quantities  $h_0$  and  $r(x_0, y_0; h_0^{(0)})$ .

For a synthetic description of variable-step Runge-Kutta methods Butcher table is completed by an supplementary line used for the computation of  $\Phi^*$  (and thus of  $r(x, y; h)$ ):

$\mu_1$	$\lambda_{11}$	$\lambda_{12}$	$\dots$	$\lambda_{1r}$	
$\mu_2$	$\lambda_{21}$	$\lambda_{22}$	$\dots$	$\lambda_{2r}$	
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	
$\mu_r$	$\lambda_{r1}$	$\lambda_{r2}$	$\dots$	$\lambda_{rr}$	
	$\alpha_1$	$\alpha_2$	$\dots$	$\alpha_r$	
	$\alpha_1^*$	$\alpha_2^*$		$\alpha_r^*$	$\alpha_{r+1}^*$

As an example, Table 9.3 is the Butcher table for a 2-3 method. For the derivation of this table see [91, pages 451–452].

Table 9.4 is the Butcher table for Bogacki-Shampine method [8]. It is the background for MATLAB `ode23` solver.

Another important example is DOPRI5 or RK5(4)7FM, a pair of order 4-5 and 7 stages (Table 9.5). This is a very efficient pair; it is the base for MATLAB `ode45` solver and for other important solvers.

We shall give, in the sequel an implementation example for a variable-step Runge-Kutta method. Following the ideas of [25], we implemented a more general MATLAB function, `oderk`, that uses a Butcher table. The error handling and general background is inspired from MATLAB functions `ode23` and `ode45`, but also from `ode23tx` function in [66].

$\mu_j$	$\lambda_{ij}$			
0	0			
$\frac{1}{4}$	$\frac{1}{4}$	0		
$\frac{27}{40}$	$-\frac{189}{800}$	$\frac{729}{800}$	0	
1	$\frac{214}{891}$	$\frac{1}{33}$	$\frac{650}{891}$	0
$\alpha_i$	$\frac{214}{891}$	$\frac{650}{891}$	0	
$\alpha_i^*$	$\frac{533}{2106}$	$\frac{800}{1053}$	$-\frac{1}{78}$	

Table 9.3: A 2-3 pair

$\mu_j$	$\lambda_{ij}$			
0	0			
$\frac{1}{2}$	$\frac{1}{2}$	0		
$\frac{3}{4}$	0	$\frac{3}{4}$	0	
1	$\frac{2}{9}$	$\frac{3}{9}$	$\frac{4}{9}$	0
$\alpha_i$	$\frac{2}{9}$	$\frac{3}{9}$	$\frac{4}{9}$	0
$\alpha_i^*$	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

Table 9.4: The Butcher table for Bogacki-Shampine method

$\mu_j$	$\lambda_{ij}$						
0	0						
$\frac{1}{5}$	$\frac{1}{5}$	0					
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0				
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$	0			
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$	0		
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$	0	
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
$\alpha_i$	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
$\alpha_i^*$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

Table 9.5: RK5(4)7FM (DOPRI5) embedded pair

The first argument of `oderk` specifies the right-hand side function,  $f(t, y)$ . It can be a function handle, a character string or an inline function. The second argument is a two-component vector, `tspan`, containing the initial and the final value, `t0` and `tfinal`, respectively. It gives the integration interval. The third argument, `y0` provides the starting values  $y_0 = y(t_0)$ . The length of `y0` provides the number of differential equations in the system. The forth argument is a function handle, that indicates a function for the initialization of Butcher table. If missing, the default method is Bogacki-Shampine method (Table 9.4, function `BS23`). The fifth argument, `opts`, contains the options of the solver. We can initialize it using the MATLAB function `odeset`. The `oderk` function takes into account only the following options: `RelTol` (relative error, default `1e-3`), `AbsTol` (absolute error, default `1e-6`), `OutputFcn` (the output function, default `odeplot`), and `Stats` (with value `on` or `off` specifying if one desires statistics). The statement

```
opts=odeset('RelTol', 1e-5, 'AbsTol', 1e-8, 'OutputFcn',...
myodeplot)
```

sets the relative error to  $10^{-5}$ , the absolute error to  $10^{-8}$  and `myodeplot` as output function.

The `oderk` output can be either numeric or graphical. Without any output argument, `oderk` produces a graph of all solution components. With two output arguments, the statements

```
[tout,yout] = oderk(F,tspan,y0)
yields a tables of abscissas and ordinates of the solution.
```

Let us examine the code of this function.

```
function [tout,yout] = oderk(F,tspan,y0,BT,opts,varargin)
function [tout,yout] = oderk(F,tspan,y0,BT,opts,varargin)
%ODERK nonstiff ODE solver
%   ODERK uses two embedded methods given by Butcher table
%
%   ODERK(F,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates
%   the system of differential equations y' = f(t,y)
%   from t=T0 to t=TFINAL. Initial condition is y(T0)=Y0.
%   F is an M-file name, an inline function or a
%   character string defining f(t,y).
%   This function must have two arguments, t and y and must
%   return a column vector of derivatives, yprime.
%
%   With two output arguments, [T,Y] = ODERK(...) return
%   a column vector T and an array Y, where Y(:,k) is the
%   solution at point T(k).
%
%   Without output arguments, ODERK plots the solution.
%
%   ODERK(F,TSPAN,Y0,RTOL) uses the relative error RTOL,
%   instead of default 1.e-3.
%
%   ODERK(F,TSPAN,Y0,BT) uses a Butcher table, BT. If BT is
%   empty or missing, one uses BS23 (Bogacki-Shampine)
%
%   ODERK(F,TSPAN,Y0,BT,OPTS) where OPTS=ODESET('reltol',...
%   'RTOL','abstol','ATOL','outputfcn','@PLOTFUN) uses the
%   relative error RTOL instead the default 1.e-3, the
```

```
% absolute error ATOL instead of the default 1.e-6 and
% call PLOTFUN instead of ODEPLOT after each
% successful step
%
% If the call has more than 5 input arguments,
% ODERK(F,TSPAN,Y0,BT,RTOL,P1,P2,...), the additional
% arguments are passed to F, F(T,Y,P1,P2,...).
%
% Stats set to 'on' provides statistics
%
% Example
tspan = [0 2*pi];
y0 = [1 0]';
F = '[0 1; -1 0]*y';
oderk(F,tspan,y0);
```

We start with variable initialization and option processing.

```
% Init variables

rtol = 1.e-3;
atol = 1.e-6;
plotfun = @odeplot;
statflag = 0;
statflag=strncmp(optsStats,'on');
if (nargin >= 4) & ~isempty(BT) %Butcher table
    [lambda, alfa, alfas, mu, s, oop, fsal]=BT();
else
    [lambda, alfa, alfas, mu, s, oop, fsal]=BS23();
end
if statflag %statistics
    stat=struct('ns',0,'nrej',0,'nfunc',0);
end

if nargin >= 5 & isnumeric(opts)
    rtol = opts;
elseif nargin >= 5 & isstruct(opts)
    statflag=strncmp(optsStats,'on');
    if ~isempty(opts.RelTol), rtol = opts.RelTol; end
    if ~isempty(opts.AbsTol), atol = opts.AbsTol; end
    if ~isempty(opts.OutputFcn),
        plotfun = opts.OutputFcn;
    end
end
if statflag %statistics
    stat=struct('ns',0,'nrej',0,'nfunc',0);
end

t0 = tspan(1);
tfinal = tspan(2);
```

```

tdir = sign(tfinal - t0);
plotit = (nargout == 0);
threshold = atol / rtol;
hmax = abs(0.1*(tfinal-t0));
t = t0;
y = y0(:);

% Make F callable

if ischar(F) & exist(F) ~= 2
    F = inline(F,'t','y');
elseif isa(F,'sym')
    F = inline(char(F),'t','y');
end

% Init outputs

if plotit
    plotfun(tspan,y,'init');
else
    tout = t;
    yout = y.';
end

```

The computation of step length is a delicate question, since it requires some knowledge about global scale of the problem.

```

% Compute initial stepsize

K=zeros(length(y0),s);
K(:,1)=F(t,y,varargin:); %first evaluation
if statflag, stat.nfunc=stat.nfunc+1; end
r = norm(K(:,1)./max(abs(y),threshold),inf) + realmin;
h = tdir*0.8*rtol^(oop)/r;

```

It follows the main loop. The integration process starts at  $t = t_0$  and increments  $t$  until it reaches  $t_{final}$ . It is possible to go backward if  $t_{final} < t_0$ .

```

% Main loop

while t ~= tfinal

    hmin = 16*eps*abs(t);
    if abs(h) > hmax, h = tdir*hmax; end
    if abs(h) < hmin, h = tdir*hmin; end

    % correct final stepsize

```

```

if 1.1*abs(h) >= abs(tfinal - t)
    h = tfinal - t;
end

```

Here is the actual computing. The first slope,  $K(:, 1)$ , is already computed. Now,  $s-1$  slope evaluations follow, where  $s$  is the number of stages.

```

% compute step attempt

for i=2:s
    K(:,i)=F(t+mu(i)*h,y+h*K(:,1:i-1)*...
        (lambda(i,1:i-1)'));
end
if statflag, stat.nfunc=stat.nfunc+s-1; end
tnew=t+h;
ynew=y+h*K*alfas;

```

Then, one estimates the error. The error vector norm is scaled by the ratio of absolute and relative error. The use of the smallest floating-point number, `realmin`, avoids `err` to be zero.

```

% Estimate error

e = h*K*(alfa-alfas);
err = norm(e./max(max(abs(y),abs(ynew)),threshold),...
    inf) + realmin;

```

One tests if the step is successful. If this is true, it displays the result or adds it to the output vector. Otherwise, if statistics are required, the unsuccessful step is counted. If the method is of type FSAL (First Same As Last), that is, the last stage of previous step is the same as the first stage of next step, then the last function value is reused.

```

% Accept solution if estimated error < tolerance

if err <= rtol %accepted step
    t = tnew;
    y = ynew;
    if plotit
        if plotfun(t,y,'');
            break
        end
    else
        tout(end+1,1) = t;
        yout(end+1,:) = y.';
        if statflag
            stat.ns=stat.ns+1;
        end
    end
end

```

```

        end
    end
    if fsal % Reuse final value if required
        K(:,1)=K(:,s);
    else
        K(:,1)=F(t,y);
        if statflag, stat.nfunc=stat.nfunc+1; end
    end
else %rejected step
    if statflag, stat.nrej=stat.nrej+1; end
end

```

We use the error estimation to compute a new step size. The ratio `rtol/err` is greater than one if the current step is successful and less than one if the current step fails. The safe factors 0.8 and 5 avoid the excessive step-length changing.

```

% Compute new step
h = h*min(5,0.8*(rtol/err)^(oop));

```

We can detect here the occurrence of a singularity.

```

% Exit if stepsize too small

if abs(h) <= hmin
    warning(sprintf('step size %e too small at ...
                    t = %e.\n',h,t));
    t = tfinal;
end
end

```

The main loop finishes here. The plot function must end its work.

```

if plotit
    plotfun([],[],'done');
end
if statflag
    fprintf('%d succesfull steps\n',stat.ns)
    fprintf('%d failed attempts\n', stat.nrej)
    fprintf('%d function evaluations\n', stat.nfunc)
end

```

For applications of numerical solution of differential equations and other numerical methods in mechanics see [52].

Solver	Problem type	Type of algorithm
ode45	Nonstiff	Explicit Runge-Kutta pair, order 4 and 5
ode23	Nonstiff	Explicit Runge-Kutta pair, order 2 and 3
ode113	Nonstiff	Explicit multistep method, variable order, orders 1 to 13
ode15s	Stiff	Implicit multistep method, variable order, orders 1 to 15
ode23s	Stiff	Modified Rosenbrock pair (one step), orders 2 and 3
ode23t	Stiff	Implicit trapezoidal rule, orders 2 and 3
ode23tb	Stiff	Implicit Runge-Kutta-type algorithm, order 2 and 3
ode15i	Fully implicit	BDF

Table 9.6: MATLAB ODE solvers

## 9.8 ODEs in MATLAB

### 9.8.1 Solvers

MATLAB has powerful facilities for solving initial value problems for ordinary differential equations:

$$\frac{d}{dt}y(t) = f(t, y(t)), \quad y(t_0) = y_0.$$

The simplest way to solve such a problem is to code a function that evaluates  $f$  and then to call one of the MATLAB's ODE solvers. The minimal information to be provided to a solver is the function name, the range of  $t$  values over which the solution is required and the initial value  $y_0$ . MATLAB's ODE solvers allow for extra (optional) input and output arguments that make it possible to specify more about the mathematical problem and how it is to be solved. Each of MATLAB's ODE solvers is designed to be efficient in specific circumstances, but all are essentially interchangeable. All solvers have the same syntax, and this fact allows us to try various methods when we do not know which is the most adequate. The simplest syntax, common to all the solver functions is

```
[t, y]=solver(@fun, tspan, y0, options)
```

where `solver` is one of the ODE solver functions given in Table 9.6.

The basic input arguments are:

`fun` Handle to a function that evaluates the system of ODEs. The function has the form

$$\text{dydt} = \text{odefun}(t, y),$$

where  $t$  is a scalar, and  $\text{dydt}$  and  $y$  are column vectors.

`tspan` Vector specifying the interval of integration. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. If it has more than two elements the solver returns the solution at those points. These elements (abscissas) must be in increasing or decreasing order. The solver does not choose the steps taking the values in `tspan`, but rather computing continuous extensions (dense outputs) having the same precision order as the solution at points generated by the solver.

`y0` Vector of initial conditions for the problem.

`options` Structure of optional parameters that change the default integration properties.

The output parameters are:

`t` Column vector of abscissas.

`y` Solution array. Each row in `y` corresponds to the solution at an abscissa returned in the corresponding row of `t`.

## 9.8.2 Nonstiff examples

Consider the scalar ODE

$$y'(t) = -y(t) + 5e^{-t} \cos 5t, \quad y(0) = 0,$$

for  $t \in [0, 3]$ . The right-hand side is given in the M file `f1scal.m`:

```
function yder=f1scal(t,y)
%F1SCAL Example of scalar ODE
yder = -y+5*exp(-t).*cos(5*t);
```

We shall use `ode45` solver. The MATLAB command sequence

```
>> tspan = [0,3]; yzero=0;
>> [t,y]=ode45(@f1scal,tspan,yzero);
>> plot(t,y,'k--*')
>> xlabel('t'), ylabel('y(t)')
```

will produce the graph in Figure 9.4. The exact solution is  $y(t) = e^{-t} \sin 5t$ . We may check the maximum error in the `ode45` approximation:

```
>> norm(y-exp(-t).*sin(5*t),inf)
ans =
3.8416e-004
```

Consider the simple pendulum equation [25, section 1.4]:

$$\frac{d^2}{dt^2}\theta(t) = -\frac{g}{L} \sin \theta(t),$$

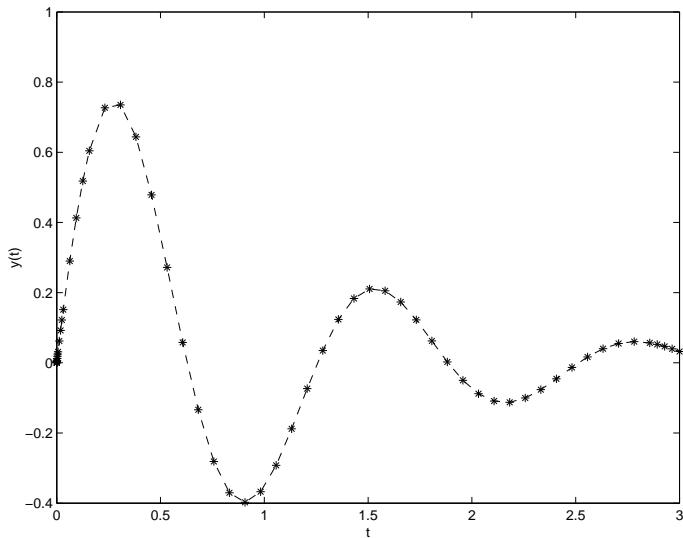


Figure 9.4: Scalar ODE example

where  $\theta$  is the angular displacement of the pendulum,  $g$  is the acceleration due to gravity and  $L$  is the length of the pendulum. Introducing the unknowns  $y_1(t) = \theta(t)$  and  $y_2(t) = d\theta(t)/dt$ , we may rewrite this equation as the two first-order equations:

$$\begin{aligned}\frac{d}{dt}y_1(t) &= y_2(t), \\ \frac{d}{dt}y_2(t) &= -\frac{g}{L} \sin y_1(t).\end{aligned}$$

These equations are coded in the file `pend.m`, given in the sequel:

```
function yp=pend(t,y,g,L)
%PEND - simple pendulum
%g - acceleration due to gravity, L - length
yp=[y(2); -g/L*sin(y(1))];
```

Here,  $g$  and  $L$  are additional parameters to be passed to `pend` by the solver. We shall compute the solution for  $t \in [0, 10]$  and three different initial conditions.

```
g=10; L=10;
tspan = [0,10];
yazero = [1; 1]; ybzero = [-5; 2];
yczero = [5; -2];
[ta,ya] = ode45(@pend,tspan,yazero,[],g,L);
[tb,yb] = ode45(@pend,tspan,ybzero,[],g,L);
[tc,yc] = ode45(@pend,tspan,yczero,[],g,L);
```

In the calls of the form

```
[ta,ya] = ode45(@pend,tspan,yazero,[],g,L);
```

[] is an empty vector of options. To produce phase plane plots, that is, plots of  $y_1(t)$  against  $y_2(t)$ , we simply plot the first column of the numerical solution against the second. In this context, it is often informative to superimpose a vector field using quiver. The arrows produced by quiver points to the direction of  $[y_2, -\sin y_1]$  and have length proportional to the 2-norm of this vector. The resulting picture is shown in Figure 9.5.

```
[y1,y2] = meshgrid(-5:0.5:5,-3:0.5:3);
Dy1Dt = y2; Dy2Dt = -sin(y1);
quiver(y1,y2,Dy1Dt,Dy2Dt)
hold on
plot(ya(:,1),ya(:,2),yb(:,1),yb(:,2),yc(:,1),yc(:,2))
axis equal, axis([-5,5,-3,3])
xlabel y_1(t), ylabel y_2(t), hold off
```

Any solution of the pendulum ODE preserves the energy: the quantity  $y_2(t)^2 - \cos y_1(t)$  is constant. We can check that this is approximately true using

```
>> Ec = 0.5*yc(:,2).^2-cos(yc(:,1));
>> max(abs(Ec(1)-Ec))
ans =
    0.0263
```

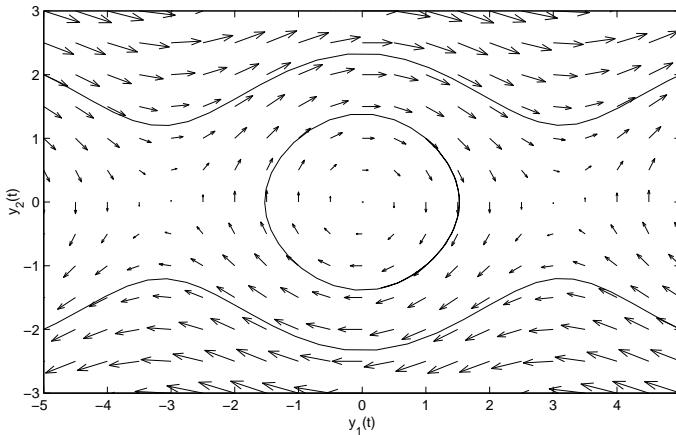


Figure 9.5: Pendulum phase plane solutions.

### 9.8.3 Options

The `odeset` function creates an options structure that you can pass as an argument to any of the ODE solver. The `odeset` arguments are pairs property name/property value. The syntax is

```
options = odeset('name1', value1, 'name2', value2, ...)
```

In the resulting structure, the named properties have the specified values. Any unspecified properties contain default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. With no input arguments, `odeset` displays all property names and their possible values; the default value are enclosed between braces:

```
>> odeset
    AbsTol: [ positive scalar or vector {1e-6} ]
    RelTol: [ positive scalar {1e-3} ]
    NormControl: [ on | {off} ]
    NonNegative: [ vector of integers ]
    OutputFcn: [ function_handle ]
    OutputSel: [ vector of integers ]
    Refine: [ positive integer ]
    Stats: [ on | {off} ]
    InitialStep: [ positive scalar ]
    MaxStep: [ positive scalar ]
        BDF: [ on | {off} ]
    MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
    Jacobian: [ matrix | function_handle ]
    JPatten: [ sparse matrix ]
    Vectorized: [ on | {off} ]
        Mass: [ matrix | function_handle ]
MStateDependence: [ none | {weak} | strong ]
    MvPattern: [ sparse matrix ]
    MassSingular: [ yes | no | {maybe} ]
    InitialSlope: [ vector ]
    Events: [ function_handle ]
```

To modify an existing structure, `oldopts`, use

```
options=odeset(oldopts,'name1', value1,...)
```

This sets `options` to the existing structure `oldopts`, overwrites any value in `oldopts` that are respecified using name/values pairs, and adds any new pairs to the structure. The command

```
options=odeset(oldopts, newopts)
```

combines the structures `oldopts` and `newopts`. In the output argument, any new options not equal to the empty matrix overwrite corresponding options in `oldopts`. An option structure created with `odeset` can be queried with

```
o=odeget(options,'name')
```

This function returns the value of the specified property, or an empty matrix [], if the property value is unspecified in the `options` structure.

Table 9.7 gives property types and property names.

Our example below solves the Rössler system [44, Section 12.2],

$$\begin{aligned}\frac{d}{dt}y_1(t) &= -y_2(t) - y_3(t), \\ \frac{d}{dt}y_2(t) &= y_1(t) + \alpha y_2(t), \\ \frac{d}{dt}y_3(t) &= b + y_3(t)(y_1(t) - c),\end{aligned}$$

where  $a$ ,  $b$  and  $c$  are real parameters. The function that defines the differential equation is:

```
function yd=Roessler(t,y,a,b,c)
%ROESSLER parametrized Roessler system

yd = [-y(2)-y(3); y(1)+a*y(2); b+y(3)*(y(1)-c)];
```

Category	Property name
Error Control	RelTol, AbsTol, NormControl
Solver output	OutputFcn, OutputSel, NonNegative Refine, Stats
Jacobian Matrix	Jacobian, JPattern, Vectorized
Step control	InitialStep, MaxStep
Mass matrix and DAE	Mass, MStateDependence, MvPattern, MassSingular, InitialSlope
Events	Events
ode15s and ode15i specific	MaxOrder, BDF

Table 9.7: ODE solver properties

We modify the absolute and relative error with

```
options = odeset('AbsTol',1e-7,'RelTol',1e-4);
```

The script Script Roessler.m (MATLAB source 9.4) solves the Rössler's system over the interval  $[0, 100]$  with initial condition  $y(0) = [1, 1, 1]^T$  and parameter sets  $(a, b, c) = (0.2, 0.2, 2.5)$  and  $(a, b, c) = (0.2, 0.2, 5)$ . Figure refroesslerfig shows the results. The 221 subplot gives the 3D phase

---

#### MATLAB Source 9.4 Rössler system

```
tspan = [0,100]; y0 = [1;1;1];
options = odeset('AbsTol',1e-7,'RelTol',1e-4);
a=0.2; b=0.2; c1=2.5; c2=5;
[t,y] = ode45(@Roessler,tspan,y0,options,a,b,c1);
[t2,y2] = ode45(@Roessler,tspan,y0,options,a,b,c2);
subplot(2,2,1), plot3(y(:,1),y(:,2),y(:,3))
title('c=2.5'), grid
xlabel('y_1(t)'), ylabel('y_2(t)'), zlabel('y_3(t)');
subplot(2,2,2), plot3(y2(:,1),y2(:,2),y2(:,3))
title('c=5'), grid
xlabel('y_1(t)'), ylabel('y_2(t)'), zlabel('y_3(t)');
subplot(2,2,3), plot(y(:,1),y(:,2))
title('c=2.5')
xlabel('y_1(t)'), ylabel('y_2(t)')
subplot(2,2,4), plot(y2(:,1),y2(:,2))
title('c=5')
xlabel('y_1(t)'), ylabel('y_2(t)')
```

---

space solution for  $c = 2.5$  and the 223 subplot gives the 2D projection onto the  $y_1 - y_2$  plane. The 222 and 224 subplots give the corresponding pictures for  $c = 5$ . We shall discuss properties and give examples in next sections. For details see help odeset or doc odeset.

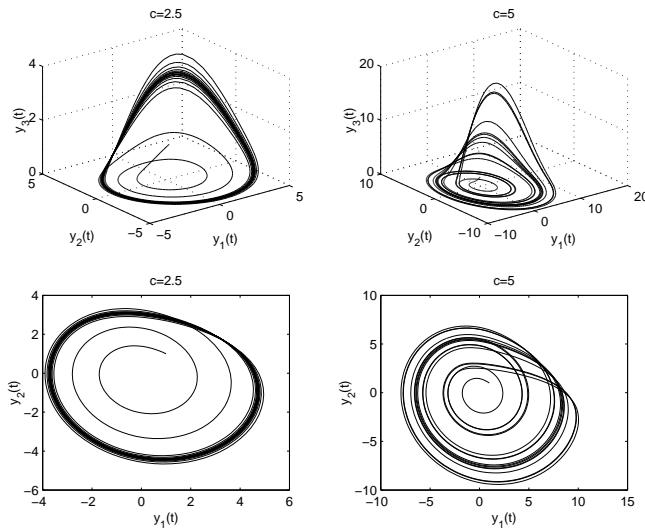


Figure 9.6: Rössler system phase space solutions

### 9.8.4 Stiff equations

Stiffness is a subtle, difficult, and important concept in the numerical solution of ordinary differential equations. It depends on the differential equation, the initial conditions, and the numerical method. Dictionary definitions of the word “stiff” involve terms like “not easily bent”, “rigid”, and “stubborn”. We are concerned with a computational version of these properties. Moler [66] characterizes this term computationally :

“A problem is stiff if the solution being sought varies slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results.”

Stiffness is an efficiency issue. Nonstiff methods can solve stiff problems; they just take a long time to do it.

The next example, due to Shampine, is from [66] and is a model of flame propagation. If you light a match, the ball of flame grows rapidly until it reaches a critical size. Then it remains at that size because the amount of oxygen being consumed by the combustion in the interior of the ball balances the amount available through the surface. A simple model is given by the initial value problem:

$$\begin{aligned} y' &= y^2 - y^3, \\ y(0) &= \delta, \quad 0 \leq t \leq 2/\delta. \end{aligned} \tag{9.8.1}$$

The real-valued function  $y(t)$  represents the radius of the ball. The  $y^2$  and  $y^3$  terms come from the surface area and the volume. The critical parameter is the initial radius,  $\delta$ , which is “small”. We seek the solution over a length of time that is inversely proportional to  $\delta$ . We shall try to solve the problem with `ode45`, for  $\delta = 0.01$  and the relative error  $10^{-4}$  (this is not a very stiff problem).

```
delta=0.01;
F = @(t,y) y^2-y^3;
```

```
opts = odeset('RelTol',1e-4);
ode45(F, [0,2/delta],delta,opts);
```

With no output arguments, `ode45` automatically plots the solution as it is computed. You should get a plot of a solution that starts at  $y = 0.01$ , grows at a modestly increasing rate until  $t$  approaches 100, which is  $1/\delta$ , then grows rapidly until it reaches a value close to 1, where it remains (it reaches a steady state). The solver uses 185 points. If we decrease  $\delta$ , say at 0.0001, the stiff character becomes

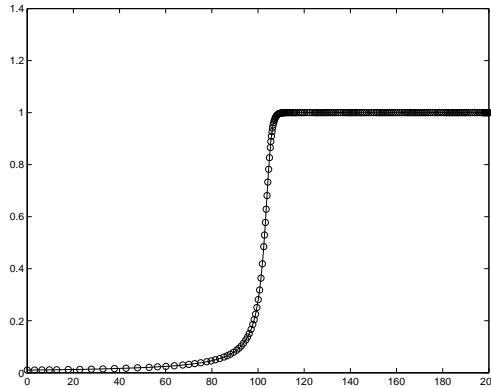


Figure 9.7: Flame propagation,  $\delta = 0.1$

stronger. The solver generates 12161 points. The graph is given in Figure 9.8, the upper part. It takes a lot of times to complete the plot. The lower part is a zoom in the neighbor of a steady state. The figure was generated with the script `flamematch2.m`, given below. The `Stats` option, set to `on`, display solver's statistics.

```
delta=1e-4; er=1e-4;
F = @(t,y) y^2-y^3;
opts = odeset('RelTol',er,'Stats','on');
[t,y]=ode45(F, [0,2/delta],delta,opts);
subplot(2,1,1)
plot(t,y,'c-'); hold on
h=plot(t,y,'bo');
set(h,'MarkerFaceColor','b','Markersize',4);
hold off
title ode45
subplot(2,1,2)
plot(t,y,'c-'); hold on
h=plot(t,y,'bo');
set(h,'MarkerFaceColor','b','Markersize',4);
axis([0.99e4,1.12e4,0.9999,1.0001])
hold off
```

Notice that the solver is keeping the solution within the required accuracy, but it must work hard for this purpose. The situation becomes more dramatic for smaller relative error, such as  $10^{-5}$  or  $10^{-6}$ . If you try for example

```
delta = 0.00001;
ode45(F, [0 2/delta], delta, opts);
```

and the plotting process is so slow you can click the stop button in the lower left corner of the window.

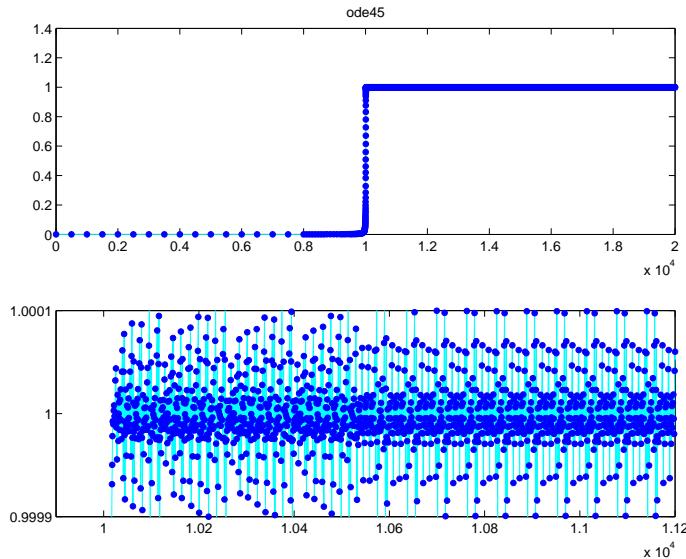


Figure 9.8: Flame propagation,  $\delta = 0.0001$ , relative error  $1e-4$  (up) and a zoom on solution (down)

The problem is not stiff initially. It only becomes stiff as the solution approaches steady state. This is because the steady state solution is so “rigid”. Any solution near  $y(t) = 1$  increases or decreases rapidly toward that solution. (We should point out that “rapidly” here is with respect to an unusually long time scale.)

We shall use a solver for stiff problems. These solvers are based on *implicit* methods. At each step they use matrix operation to solve a system of simultaneous linear equations that helps predict the evolution of the solution. For our flame example, the matrix is only 1 by 1 (this is a scalar problem), but even here, stiff methods do more work per step than nonstiff methods. Let us solve our example with a stiff solver, `ode23s`. We have only to modify the solver name: `ode23s` instead of `ode45`. Figure 9.9 shows the computed solution and the zoom detail. This time, the solver effort is much smaller, as it can be seen by examining the statistics generated by solver. In this case, the statistics for `ode23s` is:

```
99 successful steps
7 failed attempts
412 function evaluations
99 partial derivatives
106 LU decompositions
318 solutions of linear systems
```

Compare with that generated by `ode45`:

```
3040 successful steps
```

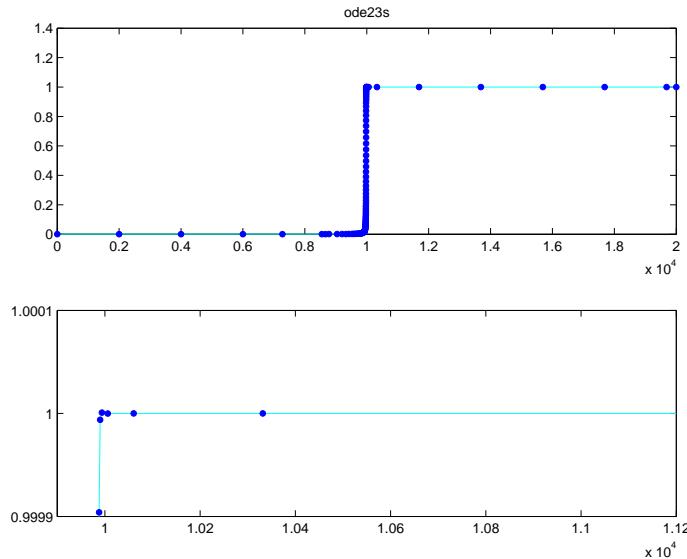


Figure 9.9: Flame propagation,  $\delta = 0.0001$ , relative error  $1e-4$  (up) and a zoom on solution (down). (Solver: `ode23s`.)

```
323 failed attempts
20179 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

It is possible to compute the exact solution of problem (9.8.1). The differential equation is separable. Integrating once gives an implicit equation for  $y$  as a function of  $t$ :

$$\frac{1}{y} + \ln\left(\frac{1}{y} - 1\right) = \frac{1}{\delta} + \ln\left(\frac{1}{\delta} - 1\right) - t.$$

The exact analytical solution to the flame model is

$$y(t) = \frac{1}{W(ae^{a-t}) + 1}$$

where  $a = 1/\delta - 1$ , and  $W$  is the Lambert's function. This is the exact solution of the functional equation.

$$W(z)e^{W(z)} = z.$$

With Matlab and the Symbolic Math Toolbox, the statements

```
y = dsolve('Dy = y^2 - y^3','y(0) = 1/100');
y = simplify(y);
pretty(y)
ezplot(y, [0, 200])
```

produces

$$\frac{1}{\text{lambertw}(99 \exp(99 - t)) + 1}$$

and the plot of the exact solution shown in Figure 9.10. If the initial value 1/100 is decreased and the time span  $0 \leq t \leq 200$  increased, the transition region becomes narrower.

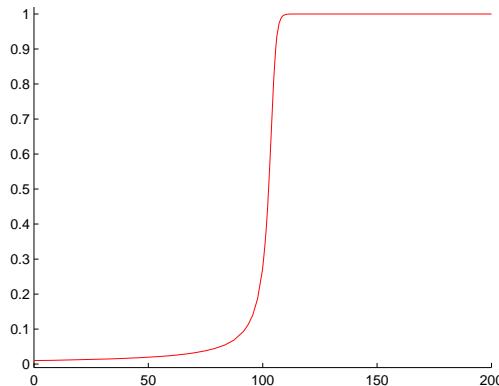


Figure 9.10: Exact solution for the flame example.

Cleve Moler [66] gives a very suggestive comparison between explicit and implicit solver.

“Imagine you are returning from a hike in the mountains. You are in a narrow canyon with steep slopes on either side. An explicit algorithm would sample the local gradient to find the descent direction. But following the gradient on either side of the trail will send you bouncing back and forth across the canyon, as with ode45. You will eventually get home, but it will be long after dark before you arrive. An implicit algorithm would have you keep your eyes on the trail and anticipate where each step is taking you. It is well worth the extra concentration.”

Consider now the an example due to Robertson and discussed in detail in [44, 25]. The Robertson ODE system

$$\begin{aligned}\frac{dy_1}{dt}(t) &= -\alpha y_1(t) + \beta y_2(t)y_3(t), \\ \frac{dy_2}{dt}(t) &= \alpha y_1(t) - \beta y_2(t)y_3(t) - \gamma y_2^2(t), \\ \frac{dy_3}{dt}(t) &= \gamma y_2^2(t)\end{aligned}$$

models a reaction between three chemicals. We set the system up as the function `chem`:

```
function yprime=chem(t,y,alpha,beta,gamma)
%CHEM - Robertson chemical reaction model
```

```
yprime = [-alpha*y(1)+beta*y(2)*y(3);
           alpha*y(1)-beta*y(2)*y(3)-gamma*y(2)^2;
           gamma*y(2)^2];
```

The script `robertson.m` solves the ODE for  $\alpha = 0.04$ ,  $\beta = 10^4$ ,  $\gamma = 3 \times 10^7$ ,  $t \in [0, 3]$  and initial condition  $y(0) = [1, 0, 0]^T$ . It uses `ode45` and the `ode15s` solvers, and generates statistics.

```
alpha = 0.04; beta = 1e4; gamma = 3e7;
tspan = [0, 3]; y0 = [1; 0; 0];
opts=odeset('Stats','on');
[ta,ya] = ode45(@chem,tspan,y0,opts,alpha,beta,gamma);
subplot(1,2,1), plot(ta,ya(:,2),'-*')
ax = axis; ax(1) = -0.2; axis(ax);
xlabel('t'), ylabel('y_2(t)')
title('ode45','FontSize',14)
[tb,yb] = ode15s(@chem,tspan,y0,opts,alpha,beta,gamma);
subplot(1,2,2), plot(tb,yb(:,2),'-*')
axis(ax)
xlabel('t'), ylabel('y_2(t)')
title('ode15s','FontSize',14)
```

Due to scale reason, only the graph of  $y_2$  was plotted in Figure 9.11. Figure 9.12 contains a zoom on solution computed by `ode45`. The solver statistics are

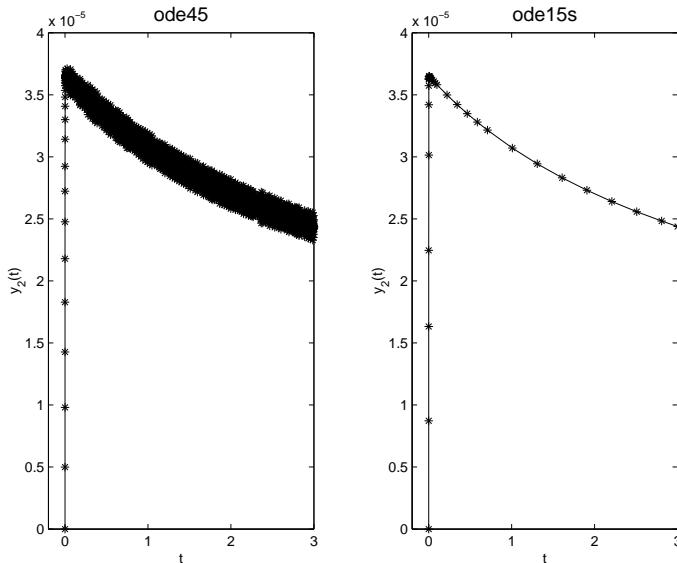


Figure 9.11: Robertson's system  $y_2$  solution. Left: `ode45`. Right: `ode15s`.

2052 successful steps

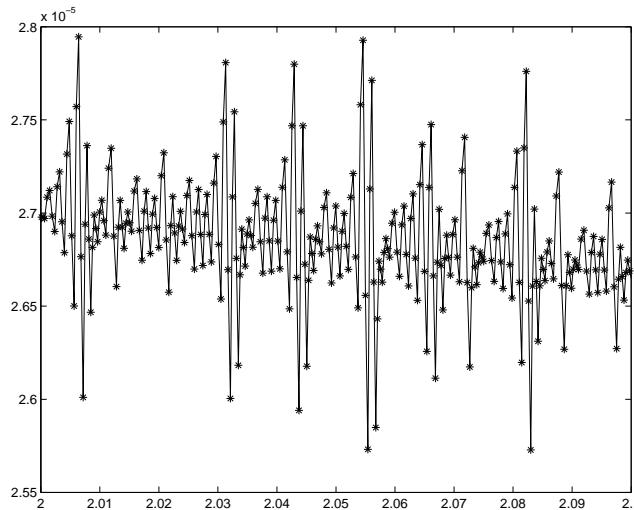


Figure 9.12: Zoom on  $y_2$  solution of Robertson's system, computed by `ode45`

```
440 failed attempts
14953 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

for `ode45` and

```
33 successful steps
5 failed attempts
73 function evaluations
2 partial derivatives
13 LU decompositions
63 solutions of linear systems
```

for `ode15s`, respectively. Note that in the computation above, we have

```
disp([length(ta), length(tb)])
8209 34
```

showing that `ode45` returned output at almost 250 times as many points as `ode15s`. However, the statistics show that `ode45` took 2051 steps, only about 62 times as many as `ode15s`. The explanation is that by default `ode45` uses interpolation to return four solution values at equally spaced points over each “natural” step. The default interpolation level can be overridden via the `Refine` property with `odeset`.

The stiff solvers use information about the Jacobian matrix,  $\partial f_i / \partial y_j$ , at various points along the solution. By default, they automatically generate approximate Jacobians using finite differences. However, the reliability and efficiency of the solvers is generally improved if a function that evaluates the

Jacobian is supplied. Options are available for specifying a function that evaluate the Jacobian or a constant matrix if the Jacobian is constant (`Jacobian`), if Jacobian is sparse and the sparsity pattern (`Jspattern`), or if it is in vector form (`Vectorized`). To illustrate how Jacobian information can be encoded, we consider system of ODEs

$$\frac{d}{dt}y(t) = Ay(t) + y(t) \cdot (1 - y(t)) + v, \quad (9.8.2)$$

where  $A$  is an  $N \times N$  matrix and  $v$  is an  $N \times 1$  vector

$$A = r_1 \begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & -1 & 0 \end{bmatrix} + r_2 \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix},$$

$v = [r_1 - r_2, 0, \dots, 0, r_1 + r_2]^T$ ,  $r_1 = -a/(2\Delta x)$  and  $r_2 = b/\Delta x^2$ . Here,  $a$ ,  $b$  and  $\Delta x$  are parameters with values  $a = 1$ ,  $b = 5 \times 10^{-2}$  and  $\Delta x = 1/(N + 1)$ . This ODE system arises when the method of lines based on central differences is used to semi-discretize the partial differential equation (PDE)

$$\frac{\partial}{\partial t}u(x, t) + a\frac{\partial}{\partial x}u(x, t) = b\frac{\partial^2}{\partial x^2}u(x, t) + u(x, t)(1 - u(x, t)), \quad 0 \leq x \leq 1,$$

with Dirichlet boundary conditions  $u(0, t) = u(1, t) = 1$ . This PDE is of reaction-convection-diffusion type (and could be solved directly with `pdepe`). The ODE solution component  $y_j(t)$  approximates  $u(j\Delta x, t)$ . We suppose that the PDE comes with the initial data  $u(x, 0) = (1 + \cos 2\pi x)/2$ , for which it can be shown that  $u(x, t)$  tends to the steady state  $u(x, t) \equiv 1$  when  $t \rightarrow \infty$ . The corresponding ODE initial condition is  $(y_0)_j = (1 + \cos(2\pi j/(N + 1))/2$ ). The Jacobian for this ODE has the form  $A + I - 2\text{diag}(y(t))$ , where  $I$  is the identity matrix. MATLAB Source 9.5 contains a function that implements and solves (9.8.2) using `ode15s`. The use of subfunctions and function handles allow the whole code to be contained in a single file, `rcd.m`. We have set  $N = 40$  and  $t \in [0, 2]$ . We specify via the Jacobian property of `odeset` the subfunction `jacobian` that evaluates the Jacobian, and the sparsity pattern of the Jacobian, encoded as a sparse matrix of 0s and 1s, is assigned to the `Jpattern` property. The  $j$ th column of the output matrix  $y$  contains the approximation to  $y_j(t)$ , and we have created  $U$  by appending an extra column `ones(size(t))` at each end of  $y$  to account for the PDE boundary conditions. The plot produced by `rcd` is shown in Figure 9.13.

The ODE solvers can be applied to problems of the form

$$M(t, y(t)) \frac{d}{dt}y(t) = f(t; y(t)), y(t_0) = y_0,$$

where the mass matrix,  $M(t, y(t))$ , is square and nonsingular. (The `ode23s` solver applies only when  $M$  is independent of  $t$  and  $y(t)$ .) Mass matrices arise naturally when semi-discretization is performed with a finite element method. A mass matrix can be specified in a similar manner to a Jacobian, via `odeset`. The `ode15s` and `ode23t` functions can solve certain problems where  $M$  is singular but does not depend on  $y(t)$ —more precisely, they can be used if the resulting differential-algebraic equation is of index 1 and  $y_0$  is close to being consistent.

## 9.8.5 Event handling

In many situations, the determination of the last value  $t_{final}$  of `tspan` is an important aspect of the problem. One example is a body falling under the force of gravity and encountering air resistance.

**MATLAB Source 9.5** Stiff problem with information about Jacobian

---

```

function rcd
%RCD Stiff ODE for reaction-convection-diffusion problem
% obtained from method of lines

N = 40; a = 1; b = 5e-2;
tspan = [0;2]; space = [1:N]/(N+1);

y0 = 0.5*(1+cos(2*pi*space));
y0 = y0(:);
options = odeset('Jacobian',@jacobian,'Jpattern',...
    jpattern(N),'RelTol',1e-3,'AbsTol',1e-3);

[t,y] = ode15s(@f,tspan,y0,options,N,a,b);
e = ones(size(t)); U = [e y e];
waterfall([0:1/(N+1):1],t,U)
xlabel('space','FontSize',16,'Interpreter','LaTeX')
ylabel('time','FontSize',16,'Interpreter','LaTeX')

% -----
% Subfunctions.
% -----
function dydt = f(t,y,N,a,b)
%F Differential equation

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
up = [y(2:N);0]; down = [0;y(1:N-1)];
e1 = [1;zeros(N-1,1)]; eN = [zeros(N-1,1);1];

dydt = r1*(up-down) + r2*(-2*y+up+down) + (r2-r1)*e1 +...
(r2+r1)*eN + y.*(1-y);

% -----
function dfdy = jacobian(t,y,N,a,b)
%JACOBIAN Jacobian matrix

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
u = (r2-r1)*ones(N,1);
v = (-2*r2+1)*ones(N,1) - 2*y;
w = (r2+r1)*ones(N,1);

dfdy = spdiags([u v w],[ -1 0 1 ],N,N);

% -----
function S = jpattern(N)
%JPATTERN Sparsity pattern of Jacobian matrix

e = ones(N,1);
S = spdiags([e e e],[ -1 0 1 ],N,N);

```

---

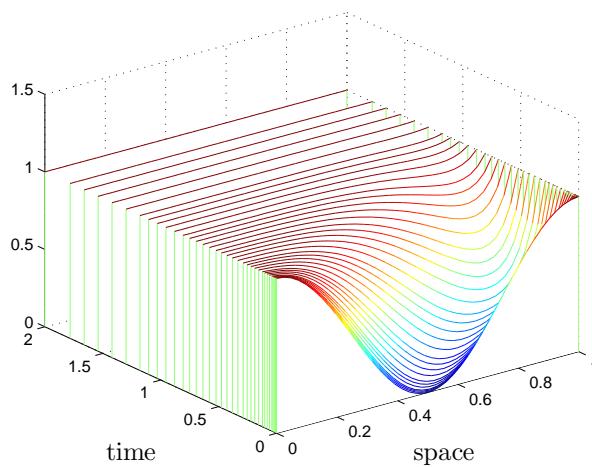


Figure 9.13: Stiff example, with information about Jacobian

When does it hit the ground? Another example is the two-body problem, the orbit of one body under the gravitational attraction of a much heavier body. What is the period of the orbit? The events feature of the MATLAB ordinary differential equation solvers provides answers to such questions.

Events detection in ordinary differential equations involves two functions,  $f(t, y)$  and  $g(t, y)$ , and an initial condition,  $(t_0, y_0)$ . The problem is to find a function  $y(t)$  and a final value  $t_*$  so that

$$\begin{aligned} y' &= f(t, y) \\ y(t_0) &= y_0 \end{aligned}$$

and

$$g(t_*, y(t_*)) = 0.$$

A simple model for the falling body is

$$y'' = -1 + y'^2,$$

with an initial conditions that provides values for  $y(0)$  and  $y'(0)$ . The question is, for what values of  $t$  we have  $y(t) = 0$ ? The source code for  $f(t, y)$  is

```
function ydot=f(t,y)
ydot = [y(2); -1+y(2)^2];
```

The equation was rewritten as a system of two first order equations, so  $g(t, y) = y_1$ . The code for  $g(t, y)$  is

```
function [gstop,isterminal,direction] = g(t,y)
gstop = y(1);
isterminal = 1;
direction = 0;
```

The first output argument, `gstop`, is the value that we want to vanish. If the second output, `isterminal`, is set to one, the solver should terminate when `gstop` is zero. If `isterminal = 0`, then the event is recorded and the solution process proceeds. The `direction` parameter may have values 0, 1 or -1, with the meaning all zeros are to be located (the default), only zeros where the event function is increasing, only zeros where the event function is decreasing, respectively. With these two functions available, the following statements compute and plot the trajectory.

```
function falling_body(y0)
opts = odeset('events', @g);
[t,y,tfinal] = ode45(@f, [0, Inf], y0, opts);
tfinal
plot(t,y(:,1),'-',[0,tfinal],[1,0],'o')
axis([-0.1, tfinal+0.1, -0.1, max(y(:,1)+0.1)]);
xlabel t
ylabel y
title('Falling body')
text(tfinal-0.8, 0, ['tfinal = ' num2str(tfinal)])
```

For the initial condition `y0=[1; 0]`, one obtains

```
>> falling_body([1;0])
tfinal =
1.65745691995813
```

and the graph in Figure 9.14.

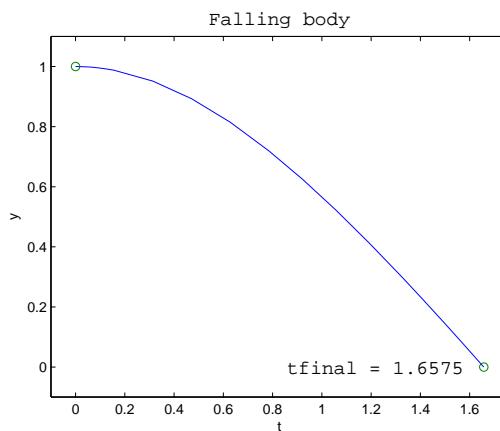


Figure 9.14: Event handling for falling object.

Events detection is particularly useful in problems involving periodic phenomena. The two body problem is a good example. It describes the orbit of a body subject of the action of gravitational force of a much heavier body. Using Cartesian coordinates,  $u(t)$  and  $v(t)$ , with the origin at position of heavier

**MATLAB Source 9.6** Two body problem

---

```

function orbit(reltol)
y0 = [1; 0; 0; 0.3];
opts = odeset('events', @gstop,'RelTol',reltol);
[t,y,te,ye] = ode45(@twobody,[0,2*pi], y0, opts, y0);
tfinal = te(end)
yfinal = ye(end,1:2)
plot(y(:,1),y(:,2),'-',0,0,'ro')
axis([-0.1 1.05 -0.35 0.35])

%-----
function ydot = twobody(t,y,y0)
r = sqrt(y(1)^2 + y(2)^2);
ydot = [y(3); y(4); -y(1)/r^3; -y(2)/r^3];

%-----
function [val,isterm,dir] = gstop(t,y,y0)
d = y(1:2)-y0(1:2);
v = y(3:4);
val = d'*v;
isterm = 1;
dir = 1;

```

---

body, our equations are

$$\begin{aligned} u''(t) &= -\frac{u(t)}{r(t)^3} \\ v''(t) &= -\frac{v(t)}{r(t)^3}, \end{aligned}$$

where  $r(t) = \sqrt{u(t)^2 + v(t)^2}$ . The complete source of the solution is contained in a single function M file, `orbit.m` (MATLAB Source 9.6). The input parameter, `reltol`, is the desired local relative tolerance. The code for the problem, `twobody`, and the event handling function, `gstop`, are given as subfunctions, but they can be kept in separate M files. The function `ode45` is used to compute the orbit. The input argument, `y0` is a 4-vector that provides the initial position and velocity. The light body starts at  $(1, 0)$ , which is a point with a distance 1 from the heavy body, and has initial velocity  $(0; 0.3)$ , which is perpendicular to the initial position vector. The input argument `opts` is an options structure created by `odeset` that overrides the default value for `reltol` and that specifies a function `gstop` that defines the events we want to locate. The last input argument of `ode45` is a copy of `y0`, passed on to both `twobody` and `gstop`. The 2-vector `d` is the difference between the current position and the starting point. The 2-vector `v` is the velocity at the current position. The quantity `val` is the inner product between these two vectors. If you specify an events function and events are detected, the solver returns three additional outputs:

- A column vector of times (abscissas) at which events occur.
- Solution values corresponding to these times (abscissas).
- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T, Y, TE, YE, IE] = solver(odefun, tspan, y0, options)
```

the solver returns these outputs as TE, YE, and IE respectively.

The expression for stopping function is

$$g(t, y) = d'(t)^T d(t),$$

where

$$d = (y_1(t) - y_1(0), y_2(t) - y_2(0))^T.$$

Points where  $g(t, y(t)) = 0$  are the local extrema of  $d(t)$ . By setting `dir = 1`, we indicate that the zeros of  $g(t, y)$  must be approached from above, so they correspond to minima. By setting `isterm = 1`, we indicate that computation of the solution should be terminated at the first minimum. If the orbit is truly periodic, then any minima of  $d$  occur when the body returns to its starting point.

Calling `orbit` with a very loose tolerance

```
orbit(2e-3)
```

produces

```
tfinal =
2.35087197761946
yfinal =
0.98107659901112 -0.00012519138558
```

and plots Figure 9.15(a). You can see from both the value of `yfinal` and the graph that the orbit does not quite return to the starting point. We need to request more accuracy.

```
orbit(1e-6)
```

yields

```
tfinal =
2.38025846171798
yfinal =
0.99998593905520 0.00000000032239
```

Now the value of `yfinal` is close enough to `y0`, and the graph looks much better (Figure 9.15(b)).

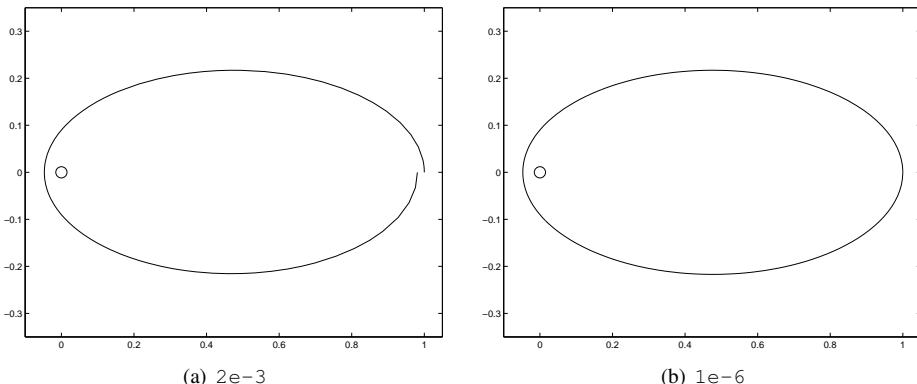
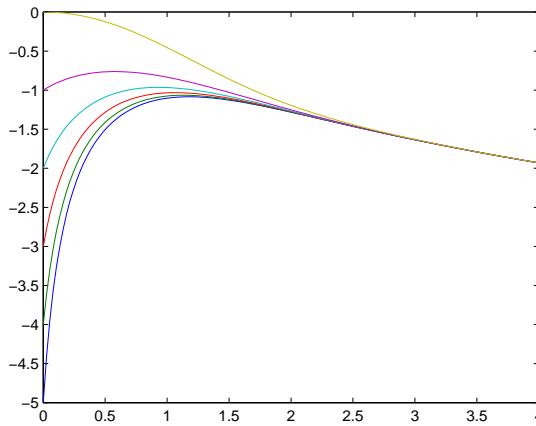


Figure 9.15: Orbit for the tolerances  $2\text{e-}3$  (left) and  $1\text{e-}6$

Figure 9.16: Solutions computed with `deval`

### 9.8.6 `deval` and `odextend`

If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns a structure `sol`, that can be used to evaluate the solution with `deval`. Consider the example: solve the ODE

$$y' = y^2 - t, \quad t \in [0, 3], \quad (9.8.3)$$

for  $y(0) = -5, -4, \dots, 0$ . We shall solve this problem using `deval`:

```
fd=@(t,y) y.^2-t;
t=linspace(0,3,150);
y=zeros(150,6);
y0=-5:0;
for k=1:length(y0)
    sol=ode45(fd,[0,8],y0(k));
    y(:,k)=deval(sol,t);
end
plot(t,y)
```

Figure 9.16 gives the graphs of the solution. For details on `deval` see `help deval` or `doc deval`.

`odextend` function extends the solution of an initial value problem for ODE. The calling syntax in its simplest form is

```
solext = odextend(sol, odefun, tfinal)
```

The next example extends the solution of (9.8.3) with initial condition  $y(0) = 0$ , from  $[0,3]$  to  $[0,8]$  and plots the solution.

```

fd=@(t,y) y^2-t;
sol=ode45(fd,[0,3],0);
sol=odextend(sol,fd,8);
t=linspace(0,8,150);
y=deval(sol,t);
plot(t,y)

```

For additional information see `help odextend` or `doc odextend`.

### 9.8.7 Implicit equations

The solver `ode15i` solves implicit ODEs of the form

$$f(t, y, y') = 0$$

using a variable order BDF method. The minimal syntax is

```
[T,Y] = ode15i(@odefun,tspan,y0,yp0)
```

but, in general it supports all the features of other solvers. The difference is the parameter `yp0` that contains the value  $y'(t_0)$ . The initial values must fulfill the consistency condition  $f(t_0, y(t_0), y'(t_0)) = 0$ . You can use the function `decic` to compute consistent initial conditions close to guessed values. To illustrate, let us solve the equation

$$y''^2 + y^2 - 1 = 0, \quad t \in [\pi/6, \pi/4],$$

with initial condition  $y(\pi/6) = 1/2$ . The exact solution is  $y(t) = \sin(t)$ , and the starting value for the derivative is  $y'(\pi/6) = \sqrt{3}/2$ . Here is the MATLAB code for the solution

```
tspan = [pi/6,pi/4];
[T,Y] = ode15i(@implic,tspan,1/2,sqrt(3)/2);
```

and for the implicit ODE

```
function z=implic(t,y,yp)
z=yp^2+y^2-1;
```

## 9.9 Applications

### 9.9.1 The restricted three-body problem

Consider the motion of a space probe in the gravitational field of two bodies (such as the earth and the moon). Both bodies impose a force on the spacecraft according to the gravitational law, but the mass of the spacecraft is too small to significantly affect the motion of the bodies. We therefore neglect the influence of the spacecraft on the two stellar bodies. In the field of celestial mechanics this problem is known as the restricted three-body problem. It is of great advantage to describe the motion of the spacecraft in a (rotating) coordinate system that has its origin in the center of gravity of the earth and the moon. In this coordinate system the earth is located at  $(-M, 0)$  and the moon at  $(E, 0)$  and the governing equations are then given as [25, 51]

$$\begin{aligned} \ddot{x} &= 2\dot{y} + x - \frac{E(x+M)}{r_1^3} - \frac{M(x-E)}{r_2^3} \\ \ddot{y} &= -2\dot{x} + y - \frac{Ey}{r_1^3} - \frac{My}{r_2^3}, \end{aligned}$$

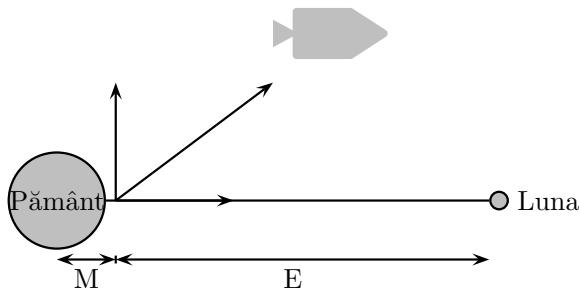


Figure 9.17: Restricted 3 body problem coordinate system

$$\text{where } r_1 = \sqrt{(x + M)^2 + y^2}, r_2 = \sqrt{(x - E)^2 + y^2}, E = 1 - M.$$

This problem has a periodic solution and the spacecraft must arrive at initial coordinates at the end of given interval. Since the system is not stable, small errors would destroy the periodicity, and for the numerical solution a larger accuracy is needed.

We convert the system of second order differential equations into a first-order system of four equations. If we introduce the unknowns

$$X_1(t) = x(t), \quad X_2(t) = \dot{x}(t), \quad Y_1(t) = y(t), \quad Y_2(t) = \dot{y}(t),$$

our system becomes

$$\frac{d}{dt} \begin{bmatrix} X_1 \\ X_2 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} X_2 \\ 2Y_2 + X_1 - \frac{E(X_1 + M)}{r_1^3} - \frac{M(X_1 - E)}{r_2^3} \\ Y_2 \\ -2X_2 + Y_1 - \frac{EY_1}{r_1^3} - \frac{MY_1}{r_2^3} \end{bmatrix}.$$

The ODE function (M-file `r3body.m`) is

```
function yp = r3body(t,y)
% R3BODY Function defining the restricted three-body ODEs

M = 0.012277471;
E = 1 - M;
%
r1 = sqrt((y(1)+M)^2 + y(3)^2);
r2 = sqrt((y(1)-E)^2 + y(3)^2);
%
yp2 = 2*y(4) + y(1) - E*(y(1)+M)/r1^3 - M*(y(1)-E)/r2^3;
yp4 = -2*y(2) + y(3) - E*y(3)/r1^3 - M*y(3)/r2^3;
yp = [y(2); yp2; y(4); yp4];
```

We shall solve the problem for two different data set. First we consider the interval  $[0, 6.192169331319640]$ ,

$M = 1/82.45$ ,  $E = 1 - M$ , and the initial conditions

$$\begin{aligned}x(0) &= 1.2, \quad \dot{x}(0) = 0, \quad y(0) = 0, \\ \dot{y}(0) &= -1.049357509830320.\end{aligned}$$

The code is listed below.

```
tspan = [0, 6.192169331319640];
M = 1/82.45; E = 1-M;
%
x0 = 1.2; xdot0 = 0; y0 = 0;
ydot0 = -1.049357509830320;
%
vec0 = [x0 xdot0 y0 ydot0];
%
options = odeset('RelTol',1e-6,'AbsTol',[1e-6 1e-6 1e-6 1e-6]);
%
[t,y] = ode45(@r3body,tspan,vec0,options);
plot(y(:,1),y(:,3))
axis([-1.5 1.5 -.8 .8]);grid on;
hold on
plot(-M,0,'o')
plot(E,0,'o');
hold off
xlabel('x');
ylabel('y')
text(0,0.1,'Earth','FontSize',16)
text(0.9,-0.1,'Moon','FontSize',16)
```

The results are shown in Figure 9.18.

For the second data set, we take  $t \in [0, 29.4602]$ ,  $M = 0.012277471$ ,  $E = 1 - M$ , and the initial conditions

$$\begin{aligned}x(0) &= 1.15, \quad \dot{x}(0) = 0, \quad y(0) = 0, \\ \dot{y}(0) &= 0.0086882909.\end{aligned}$$

Here is the code:

```
clear; close all;
tspan = [0 29.4602]; %experiment
%
M = 0.012277471; E = 1-M;
%
x0 = 1.15;
xdot0 = 0;
y0 = 0;
ydot0 = 0.0086882909;
vec0 = [x0 xdot0 y0 ydot0];
scal=1e-1; %1e0, 1e1, 1e2, 1e3,....
options = odeset('RelTol',1e-6*scal,'AbsTol',...
```

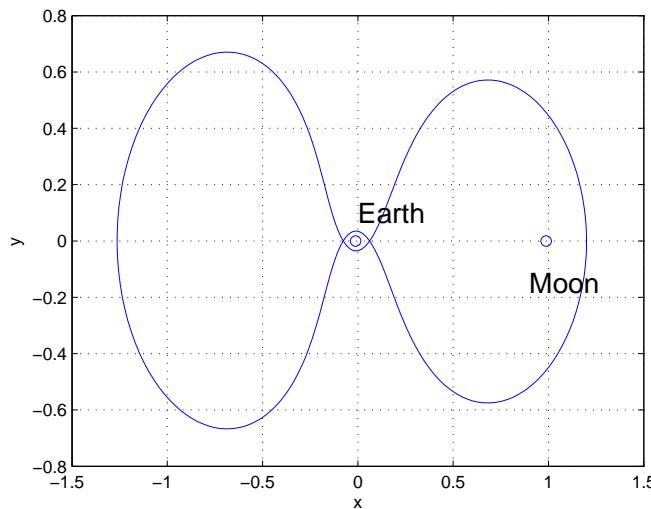


Figure 9.18: Restricted 3 body problem solution

```
[1e-6 1e-6 1e-6 1e-6]*scal);
[t,y] = ode45(@r3body,tspan,vec0,options);
figure(1);plot(y(:,1),y(:,3))
axis([- .8 1.2 -.8 .8]);grid on;
hold on
plot(-M,0,'o')
plot(E,0,'o');
hold off
xlabel('x'): ylabel('y')
text(0,0.1,'Earth','FontSize',16)
text(0.9,-0.15,'Moon','FontSize',16)
```

See Figure 9.19 for the output.

We can add to our main code the next MATLAB sequence to implement a simple animation based on `comet` function

```
figure(2);
shg
axis([- .8 1.2 -.8 .8]);grid on;
hold on
plot(-M,0,'o')
plot(E,0,'o');

xlabel('x'): ylabel('y')
text(0,0.1,'Earth','FontSize',16)
text(0.9,-0.15,'Moon','FontSize',16)
```

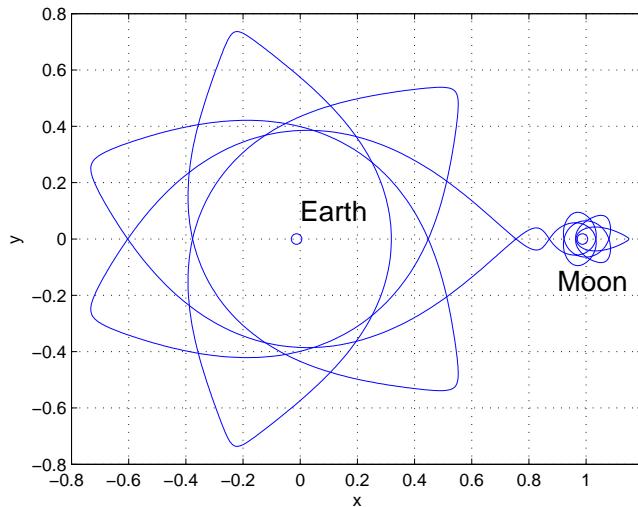


Figure 9.19: The Three-Body Problem in a rotating coordinate system

```
comet(y(:,1),y(:,3))
hold off
```

### 9.9.2 Motion of a projectile

The problem of aiming a projectile to strike a distant target involves integrating a system of differential equations governing the motion and adjusting the initial inclination angle to achieve the desired hit [105]. Assuming an atmospheric drag proportional to the square of velocity, the motion is governed by the equations

$$\dot{v}_x = -cvv_x, \quad \dot{v}_y = -g - cvv_y, \quad \dot{x} = v_x, \quad \dot{y} = v_y,$$

where  $g$  is the gravity constant and  $c$  is a ballistic coefficient depending on such physical properties as the projectile shape and air density. Because the target will be located at a distant point  $(x_f, y_f)$  relative to the initial position  $(0, 0)$  where the projectile is launched, our independent variable will be the horizontal position  $x$ , rather than the time. In order to reformulate the equations in terms of  $x$ , we use the relationship

$$dx = v_x dt \text{ or } \frac{dt}{dx} = \frac{1}{v_x}.$$

Then

$$\frac{dy}{dx} = \frac{v_y}{v_x}, \quad \frac{dv_y}{dt} = v_x \frac{dv_y}{dx}, \quad \frac{dv_x}{dt} = v_x \frac{dv_x}{dx},$$

and the equations of motion become

$$\frac{dy}{dx} = \frac{v_y}{v_x}, \quad \frac{dt}{dx} = \frac{1}{v_x}, \quad \frac{dv_x}{dx} = -cv, \quad \frac{dv_y}{dx} = \frac{-(g + cvv_y)}{v_x}.$$

Taking a vector  $z$  defined by  $z = [v_x, v_y, y, t]^T$  leads to a first order matrix differential equation

$$\frac{dz}{dx} = \frac{[-cvv_x, -(g + cvv_y), v_y, 1]^T}{v_x},$$

where

$$v = \sqrt{v_x^2 + v_y^2}.$$

The problem becomes ill-posed if the initial velocity of the projectile is not large enough so that the maximum desired value of  $x$  is reached before  $v_x$  is reduced to zero from atmospheric drag. Consequently, error checking is needed to handle such a circumstance. The function `protraject` uses `ode45` to compute the projectile trajectory. The equations of motion uses some global variables, so we chose to implement them as an embedded function (`projcteq` function). Error check is implemented via event handling facility. Graphical results for default data are given in Figure 9.20.

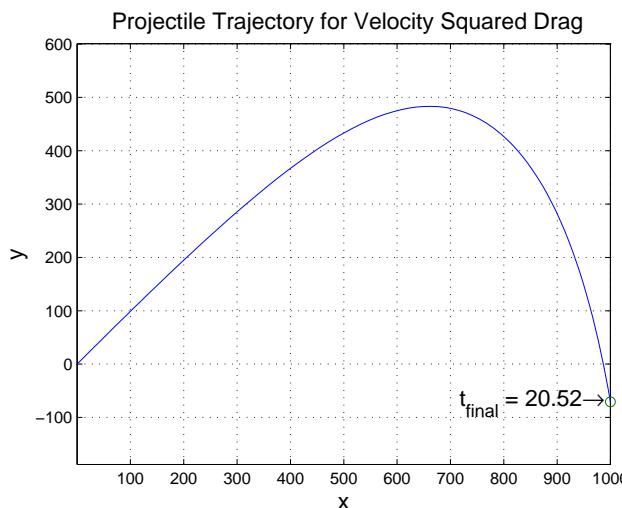


Figure 9.20: Projectile trajectory for  $v^2$  drag condition

```
function [y,x,t]=protraject(angle,vinit,grav,dragc,xfinl)
%PROTRAJECT - trajectory of a projectile
% angle - initial angle in degrees
% vinit - initial velocity of the projectile
% gravity - the gravitational constant
% cdrag - drag coefficient
% xfinl - largest x value for which the
% solution is computed. %
% y,x,t - the y, x and time vectors produced
% by integrating the equations of motion

% Default data case generated
if nargin < 5, xfinl=1000; end
```

```

if nargin < 4, dragc=0.002; end
if nargin < 3, grav=9.81; end
if nargin < 2, vinit=340; end
if nargin < 1, angle=45; end

% Evaluate initial velocity
ang=pi/180*angle;
vtol=vinit/1e6;
%initial conditions
z0=[vinit*cos(ang); vinit*sin(ang); 0; 0];

% Integrate the equations of motion defined
% in function projteq
deoptn=odeset('RelTol',1e-6,'Events',@events);
[x,z,te,ze,ie]=ode45(@projteq,[0,xfin],z0,deoptn);

% Plot the trajectory curve
y=z(:,3); t=z(:,4);
plot(x,y,'-',x(end),y(end),'o');
ss=sprintf('{t_{final}} = %5.2f \rightarrow ',t(end));
text(x(end),y(end),ss,'HorizontalAlignment','Right',...
    'FontSize',14);
xlabel('x','FontSize',14); ylabel('y','FontSize',14);
title(['Projectile Trajectory for ', ...
    'Velocity Squared Drag'],'FontSize',14);
axis('equal'); grid on;
%error
if ~isempty(te)
    error('initial velocity too small')
end

%-----
function zp=projteq(t,y)
    %PROJCTEQ - DE of projectile
    v=sqrt(y(1)^2+y(2)^2);
    zp=[-dragc*v; -(grav+dragc*v*y(2))/y(1); ...
        y(2)/y(1); 1/y(1)];
end
%-----
function [val,isterm,dir] = events(t,y)
    %EVENTSH - event handling function
    val = abs(y(1))-vtol;
    dir = -1;
    isterm =1;
end
end

```

Another interesting problem on projectile motion is Problem 9.14.

## Problems

**Problem 9.1.** Solve the problem

$$y' = 1 - y^2, \quad y(0) = 0.$$

using various methods whose Butcher tables were given in this chapter, and also `ode23` and `ode45`. Compute the global error, given that the exact solution is

$$y(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

and check it is  $O(h^p)$ .

**Problem 9.2.** Solve the equations and compare to the exact solution:

(a)

$$y' = \frac{1}{4}y\left(1 - \frac{1}{20}y\right), \quad x \in [0, 20], \quad y(0) = 1;$$

with exact solution

$$y(x) = \frac{20}{1 + 19e^{-x/4}};$$

(b)

$$y'' = 0.032 - 0.4(y')^2, \quad x \in [0, 20], \quad y(0) = 30, \quad y'(0) = 0;$$

with exact solution

$$\begin{aligned} y(x) &= \frac{5}{2} \log \left( \cosh \left( \frac{2\sqrt{2}x}{25} \right) \right) + 30, \\ y'(x) &= \frac{\sqrt{2}}{5} \tanh \left( \frac{2\sqrt{2}x}{25} \right). \end{aligned}$$

**Problem 9.3.** The equation of Lorenz attractor

$$\begin{aligned} \frac{dx}{dt} &= -ax + ay, \\ \frac{dy}{dt} &= bx - y - xz, \\ \frac{dz}{dt} &= -cz + xy \end{aligned}$$

has chaotic solutions which are very sensitive to change in initial conditions. Solve it numerically for  $a = 5$ ,  $b = 15$ ,  $c = 1$  with initial conditions

$$x(0) = 2, \quad y(0) = 6, \quad z(0) = 4, \quad t \in [0, 20],$$

and the tolerance  $T = 10^{-4}$ . Repeat for

(a)  $T = 10^{-5}$ ;

(b)  $x(0) = 2.1$ .

Compare both results. Plot them in each case.

**Problem 9.4.** The progress of an epidemic of influenza in a population of  $N$  individuals is modeled by the system of differential equation

$$\begin{aligned}\frac{dx}{dt} &= -\beta xy + \gamma, \\ \frac{dy}{dt} &= \beta xy - \alpha y, \\ \frac{dz}{dt} &= \alpha y - \gamma,\end{aligned}$$

where  $x$  is the number of people susceptible to the disease,  $y$  is the number of infected, and  $z$  is the number of immunes, which includes those recovered from the disease, at time  $t$ . The parameters  $\alpha$ ,  $\beta$ ,  $\gamma$  are the rates of recovery, transmission and replenishment (per day), respectively. It is assumed that the population is fixed so that new births are balanced by deaths.

Use `ode15` and `ode45` functions to solve the equations with initial conditions  $x(0) = 980$ ,  $y(0) = 20$ ,  $z(0) = 0$ , given the parameters  $\alpha = 0.05$ ,  $\beta = 0.0002$ ,  $\gamma = 0$ . You should terminate the simulation when  $y(t) > 0.9N$ . Determine approximately the maximum number of people infected and when it occurs.

Investigate the effect of (a) varying the initial number of infected individuals on the progress of epidemic, and (b) introducing a nonzero replenishment factor.

**Problem 9.5.** [25] Captain Kirk<sup>4</sup> and his crew, aboard the starship *Enterprise* are stranded without power in an orbit around the earth-like planet Capella III, at an altitude of 127 km. Atmospheric drag is causing the orbit to decay, and if the ship reaches denser layers of the atmosphere, excessive deceleration and frictional heating will cause irreparable damage to the life-support system. The science officer, Mr. Spock, estimates that the temporary repairs to the impulse engines will take 29 minutes provided that they can be completed before the deceleration rises to  $5g$  ( $1g = 9.81\text{ms}^{-1}$ ). Since Mr. Spock is a mathematical genius, he decides to simulate the orbital decay by solving the equations of motion numerically with the DOPRI5 embedded pair. The equation of motion of the starship, subject to atmospheric drag, are given by

$$\begin{aligned}\frac{dv}{dt} &= \frac{GM \sin \gamma}{r^2} - c\rho v^2 \\ \frac{d\gamma}{dt} &= \left( \frac{GM}{rv} - v \right) \frac{\cos \gamma}{r} \\ \frac{dz}{dt} &= -v \sin \gamma \\ \frac{d\theta}{dt} &= \frac{v \cos \gamma}{r},\end{aligned}$$

where

- $v$  is the tangential velocity (m/s);
- $\gamma$  is the re-entry angle (between velocity vector an the horizontal);
- $z$  is the altitude (m);
- $M$  is the planetary mass ( $6 \times 10^{24}$  kg);
- $G$  is the constant of gravitation ( $6,67 \times 10^{-11}$  SI);
- $c$  is a drag constant ( $c = 0.004$ );

---

<sup>4</sup>Characters from Star Trek, 1st series

$r$  is the distance to the center of the planet ( $z + 6.37 \times 10^6$  m);

$\rho$  is the atmospheric density ( $1.3 \exp(-z/7600)$ );

$\theta$  is the longitude;

$t$  is the time (s).

At time  $t = 0$ , the initial values are  $\gamma = 0$ ,  $\theta = 0$  and  $v = \sqrt{GM/r}$ . Mr. Spock solved the equations numerically to find the deceleration history and the time and the place of the impact of the Enterprise should its orbital decay not be prevented. Repeat his simulation using a variable step Runge-Kutta method, and also estimate approximately the maximum deceleration experienced during descent and the height at which this occurs. Should Captain Kirk give the order to abandon ship?

**Problem 9.6.** Halley's comet last reached perihelion (closest to the Sun) on February, 9th, 1986. Its position and velocity components at this time were

$$(x, y, z) = (0.325514, -0.459460, 0.166229)$$

$$\left( \frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt} \right) = (-9.096111, -6.916686, -1.305721).$$

Position is measured in astronomical units (the Earth's mean distance to the Sun), and the time in years. The equations of motion are

$$\begin{aligned} \frac{d^2x}{dt^2} &= -\frac{\mu x}{r^3}, \\ \frac{d^2y}{dt^2} &= -\frac{\mu y}{r^3}, \\ \frac{d^2z}{dt^2} &= -\frac{\mu z}{r^3}, \end{aligned}$$

where  $r = \sqrt{x^2 + y^2 + z^2}$ ,  $\mu = 4\pi^2$ , and the planetary perturbations have been neglected. Solve these equations numerically to determine approximately the time of the next perihelion.

**Problem 9.7.** Consider the gravitational two-body orbit problem, written here as a system of four first order equations

$$\begin{aligned} y'_1 &= y_3 \\ y'_2 &= y_4 \\ y'_3 &= -y_1/r^3 \\ y'_4 &= -y_2/r^3, \end{aligned}$$

with initial conditions

$$y(0) = \left[ 1 - e, 0, 0, \sqrt{\frac{1+e}{1-e}} \right]^T,$$

where  $r = \sqrt{y_1^2 + y_2^2}$ . The true solution represents the motion on an elliptic orbit with eccentricity  $e \in (0, 1)$ , and period  $2\pi$ .

(a) Show that the solutions could be written as

$$\begin{aligned} y_1 &= \cos E - e \\ y_2 &= \sqrt{1-e^2} \sin E \\ y_3 &= \sin E / (e \cos E - 1) \\ y_4 &= \sqrt{1-e^2} \cos E / (1 + e \cos E), \end{aligned}$$

where  $E$  is the solution of Kepler's equation

$$E - e \sin E = x.$$

- (b) Solve the problem using `ode45` and plot the solution, the variation of step length for  $x \in [0, 20]$  and a precision of  $10^{-5}$ .
- (c) Find global errors, number of function evaluations and number of step rejection for tolerances of  $10^{-4}, 10^{-5}, \dots, 10^{-11}$ .

**Problem 9.8.** [66] In the 1968 Olympic games in Mexico City, Bob Beamon established a world record with a long jump of 8.90 m. This was 0.80m longer than the previous world record. Since 1968, Beamon's jump has been exceeded only once in competition, by Mike Powell's jump of 8.95m in Tokyo in 1991. After Beamon's remarkable jump, some people suggested that the lower air resistance at Mexico City's 2250m altitude was a contributing factor. This problem examines that possibility. The fixed Cartesian coordinate system has a horizontal  $x$ -axis, a vertical  $y$ -axis, and an origin at the takeoff board. The jumper's initial velocity has magnitude  $v_0$  and makes an angle with respect to the  $x$ -axis of  $\theta_0$  radians. The only forces acting after takeoff are gravity and the aerodynamic drag,  $D$ , which is proportional to the square of the magnitude of the velocity. There is no wind. The equations describing the jumper's motion are

$$\begin{aligned} x' &= v \cos \theta, & y' &= v \sin \theta, \\ \theta' &= -\frac{g}{v} \cos \theta, & v' &= -\frac{D}{m} - g \sin \theta. \end{aligned}$$

The drag is

$$D = \frac{c\rho s}{2}(x'^2 + y'^2).$$

Constants for this exercise are the acceleration of gravity,  $g = 9.81 \text{ m/s}^2$ , the mass,  $m = 80 \text{ kg}$ , the drag coefficient,  $c = 0.72$ , the jumper's cross-sectional area,  $s = 0.50 \text{ m}^2$ , and the takeoff angle,  $\theta_0 = 22.5^\circ = \pi/8$  radians. Compute four different jumps, with different values for initial velocity,  $v_0$ , and air density,  $\rho$ . The length of each jump is  $x(t_f)$ , where the air time,  $t_f$ , is determined by the condition  $y(t_f) = 0$ .

- (a) "Nominal" jump at high altitude.  $v_0 = 10 \text{ m/s}$  and  $\rho = 0.94 \text{ kg/m}^3$ .
- (b) "Nominal" jump at sea level.  $v_0 = 10 \text{ m/s}$  and  $\rho = 1.29 \text{ kg/m}^3$ . item [(c)] Sprinter's approach at high altitude.  $\rho = 0.94 \text{ kg/m}^3$ . Determine  $v_0$  so that the length of the jump is Beamon's record, 8.90 m.
- (d) Sprinter's approach at sea level.  $\rho = 1.29 \text{ kg/m}^3$  and  $v_0$  is the value determined in (c).

Present your results by completing the following table.

$v_0$	$\theta_0$	$\rho$	distance
10	22.5	0.94	???
10	22.5	1.29	???
???	22.5	0.94	8.90
???	22.5	1.29	???

Which is more important, the air density or the jumper's initial velocity?

**Problem 9.9.** Solve the stiff problem

$$\begin{aligned} y'_1 &= \frac{1}{y_1} - x^2 - \frac{2}{x^3}, \\ y'_2 &= \frac{y_1}{y_2^2} - \frac{1}{x} - \frac{1}{2x^{3/2}}, \end{aligned}$$

$x \in [1, 10]$ , with initial conditions  $y_1(1) = 1$ ,  $y_2 = 1$ , using a nonstiff solver and then a stiff solver. The exact solutions  $y_1 = 1/x^2$ ,  $y_2 = 1/\sqrt{x}$ .

**Problem 9.10.** Van der Pol equation has the form

$$y''_1 - \mu(1 - y_1^2)y'_1 + y_1 = 0, \quad (9.9.1)$$

where  $\mu > 0$  is a scalar parameter.

1. Solve the equation for  $\mu = 1$  on  $[0, 20]$  and initial conditions  $y(0) = 2$  and  $y'(0) = 0$  (nonstiff). Plot  $y$  and  $y'$ .
2. Solve the equation for  $\mu = 1000$  (stiff), time interval  $[0, 3000]$  the vector of initial values  $[2; 0]$ . Plot  $y$ .

**Problem 9.11.** A cork of length  $L$  is on the point of being ejected from a bottle containing a fermenting liquid. The equations of motion of the cork may be written

$$\begin{aligned} \frac{dv}{dt} &= \begin{cases} g(1+q) \left[ \left(1 + \frac{x}{d}\right)^{-\gamma} + \frac{RT}{100} - 1 + \frac{qx}{L(1+q)} \right], & x < L; \\ 0, & x \geq L \end{cases} \\ \frac{dx}{dt} &= v, \end{aligned}$$

where

$g$  is the acceleration due to gravity,

$q$  is the friction-weight ratio of the cork,

$x$  is the cork displacement in the neck of the bottle,

$t$  is the time,

$d$  is the length of the bottle neck,

$R$  is the percentage rate at which the pressure is increasing,

$\gamma$  is the adiabatic constant for the gas in the bottle ( $\gamma = 1.4$ ).

The initial condition is  $x(0) = x'(0) = 0$ . While  $x < L$  the cork is still in the bottle but it leaves the bottle at  $x = L$ . Integrate the equations of motion with DOPRI5 (Table 9.5) and tolerance 0.000001 to find the time at which the cork is ejected. Also find the velocity of ejection when

$$q = 20, \quad L = 3.75\text{cm}, \quad d = 5\text{cm}, \quad R = 4.$$

**Problem 9.12.** A simple model of the human heartbeat gives

$$\begin{aligned} \varepsilon x' &= -(x^3 - Ax + c), \\ c' &= x, \end{aligned}$$

where  $x(t)$  is the displacement from the equilibrium of the muscle fiber,  $c(t)$  is the concentration of a chemical control, and  $\varepsilon$  and  $A$  are positive constants. Solutions are expected to be periodic. This can be seen by plotting the solution in the phase plane ( $x$  on horizontal axis,  $c$  on the vertical), which should produce a closed curve. Assume that  $\varepsilon = 1$  and  $A = 3$ .

- (a) Calculate  $x(t)$  and  $c(t)$ , for  $0 \leq t \leq 12$  starting with  $x(0) = 0.1$ ,  $c(0) = 0.1$ . Sketch the output in the phase plane. What does the period appear to be?  
(b) Repeat (a) with  $x(0) = 0.87$ ,  $c(0) = 2.1$ .

**Problem 9.13.** Design and implement a step control strategy for Euler method with an error estimator based on Heun method. Test on two problems in this chapter.

**Problem 9.14.** [66] Determine the trajectory of a spherical cannonball in a stationary Cartesian coordinate system that has a horizontal  $x$ -axis, a vertical  $y$ -axis, and an origin at the launch point. The initial velocity of the projectile in this coordinate system has magnitude  $v_0$  and makes an angle with respect to the  $x$ -axis of  $\mu_0$  radians. The only forces acting on the projectile are gravity and the aerodynamic drag,  $D$ , which depends on the projectile's speed relative to any wind that might be present. The equations describing the motion of the projectile are

$$\begin{aligned}x' &= v \cos \theta, & y' &= v \sin \theta, \\ \theta' &= -\frac{g}{v} \cos \theta, & v' &= -\frac{D}{m} - g \sin \theta.\end{aligned}$$

Constants for this problem are the acceleration of gravity,  $g = 9.81 \text{ m/s}^2$ , the mass,  $m = 15 \text{ kg}$ , and the initial speed,  $v_0 = 50 \text{ m/s}$ . The wind is assumed to be horizontal and its speed is a specified function of time,  $w(t)$ . The aerodynamic drag is proportional to the square of the projectile's velocity relative to the wind:

$$D(t) = \frac{c\rho s}{2} ((x' - w(t))^2 + y'^2),$$

where  $c = 0.2$  is the drag coefficient,  $\rho = 1.29 \text{ kg/m}^3$  is the density of air, and  $s = 0.25 \text{ m}^2$  is the projectile's cross-sectional area.

Consider four different wind conditions.

- No wind.  $w(t) = 0$  for each  $t$ .
- Steady headwind.  $w(t) = -10 \text{ m/s}$  for all  $t$ .
- Intermittent tailwind.  $w(t) = 10 \text{ m/s}$  if the integer part of  $t$  is even, zero otherwise.
- Gusty wind.  $w(t)$  is a Gaussian random variable with mean zero and standard deviation 10 m/s.

The integer part of a real number  $t$  is denoted by  $\lceil t \rceil$  and is computed in MATLAB by `floor(t)`. A Gaussian random variable with mean 0 and standard deviation  $\sigma$  is generated by `sigma*randn`.

For each of these four wind conditions, carry out the following computations. Find the 17 trajectories whose initial angles are multiples of 5 degrees, that is,  $\theta_0 = k\pi/36$  radians,  $k = \overline{1, 17}$ . Plot all 17 trajectories on one figure. Determine which of these trajectories has the greatest downrange distance. For that trajectory, report the initial angle in degrees, the flight time, the downrange distance, the impact velocity, and the number of steps required by the ordinary differential equation solver.

Which of the four wind conditions requires the most computation? Why?

**Problem 9.15.** [25] A parachutist jumps from an aeroplane traveling with speed  $v \text{ m/s}$  at an altitude of  $y \text{ m}$ . After a period of free-fall, the parachute is opened at height  $y_p$ . The equations of motion of the skydiver are

$$\begin{aligned}x' &= v \cos \theta, \\ y' &= v \sin \theta, \\ v' &= -D/M - g \sin \theta, & D &= \frac{1}{2} \rho C_D A v^2, \\ \theta' &= -g \cos \theta / v,\end{aligned}$$

where  $x$  is the horizontal coordinate,  $\theta$  is the angle of descent,  $D$  is the drag force, and  $A$  is the reference area for the drag force given by

$$A = \begin{cases} \sigma, & \text{if } y \geq y_p; \\ S, & \text{if } y < y_p. \end{cases}$$

The constants are

$$\begin{aligned} g &= 9.81 \text{ m/s}, & M &= 80\text{kg}, & \rho &= 1.2\text{kg/m}^3, \\ \sigma &= 0.5\text{m}^2, & S &= 30\text{m}^2, & C_D &= 1. \end{aligned}$$

Use an appropriate solve to simulate the descent of the parachutist. Use the interpolation facilities to determine the critical height  $y_p$  and the time of impact. Also estimate the minimum velocity of the descent prior to parachute deployment.

**Problem 9.16.** In [44], the authors solve the following funny pursuit problem. Suppose that a rabbit follows a predefined path  $(r_1(t), r_2(t))$  in the plane, and that a fox chases the rabbit in such a way that (a) at each moment the tangent of the fox's path points towards the rabbit and (b) the speed of the fox is some constant  $k$  times the speed of the rabbit. Then the path  $(y_1(t), y_2(t))$  of the fox is given by the system of ODEs

$$\begin{aligned} \frac{d}{dt}y_1(t) &= s(t)(r_1(t) - y_1(t)), \\ \frac{d}{dt}y_2(t) &= s(t)(r_2(t) - y_2(t)), \end{aligned}$$

where

$$s(t) = \frac{k \sqrt{\left(\frac{d}{dt}r_1(t)\right)^2 + \left(\frac{d}{dt}r_2(t)\right)^2}}{\sqrt{(r_1(t) - y_1(t))^2 + (r_2(t) - y_2(t))^2}}.$$

Note that this ODE system becomes ill-defined if the fox approaches the rabbit. We let the rabbit follow an outward spiral,

$$\begin{bmatrix} r_1(t) \\ r_2(t) \end{bmatrix} = \sqrt{1+t} \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix},$$

and start the fox at  $y_1(0) = 3$ ,  $y_2(0) = 0$ .

- (a) Integrate the ODE system for  $k < 1$  (the rabbit faster than the fox).
- (b) Integrate the ODE system for  $k > 1$  (the fox faster than the rabbit), using event handling facilities.
- (c) In each case, plot the trajectories of the animals and for the case (b) display the moment of capture.



# CHAPTER 10

---

## Multivariate Approximation

---

### 10.1 Interpolation in Higher Dimensions

The problem of interpolation in several dimensions is more difficult than the univariate analogous. It exhibits unusual aspects which do not appear in the univariate case, and these aspects are apparent even in the bivariate case. The problem of multivariate interpolation attracted the attention of the researchers, both in the past and currently. Nevertheless, few classical books on numerical analysis treat it ([50, 11]).

#### 10.1.1 Interpolation problem

Let us state the bivariate interpolation problem. Given a set of  $n$  *distinct* interpolation points (or *nodes*)

$$\mathcal{N} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\} \subset D \subseteq \mathbb{R}^2 \quad (10.1.1)$$

and  $n$  real numbers  $c_1, \dots, c_n$ , find a smooth and easily computed function  $F$  such that

$$F(x_i, y_i) = c_i, \quad i = 1, \dots, n.$$

The terms *smooth* and *easily computed* have only informal or intuitive meaning. The set  $D$  is some large domain that includes all nodes in (10.1.1).

#### 10.1.2 Cartesian product and grid

In the particular case when the set of nodes  $\mathcal{N}$  is a *Cartesian product* or a *Cartesian grid*

$$\mathcal{N} = \{x_1, x_2, \dots, x_p\} \times \{y_1, y_2, \dots, y_q\},$$

or

$$\mathcal{N} = \{(x_i, y_j) : i = 1, \dots, p, j = 1, \dots, q\}, \quad (10.1.2)$$

the interpolation problem just described can be solved by a *tensor product* of univariate interpolation methods. An example of Cartesian grid, in which  $p = 4$  and  $q = 3$ , is shown in Figure 10.1. For convenience, we have numbered the  $x$ -points from left to right, and the  $y$ -points from bottom to top, although this is not necessary.

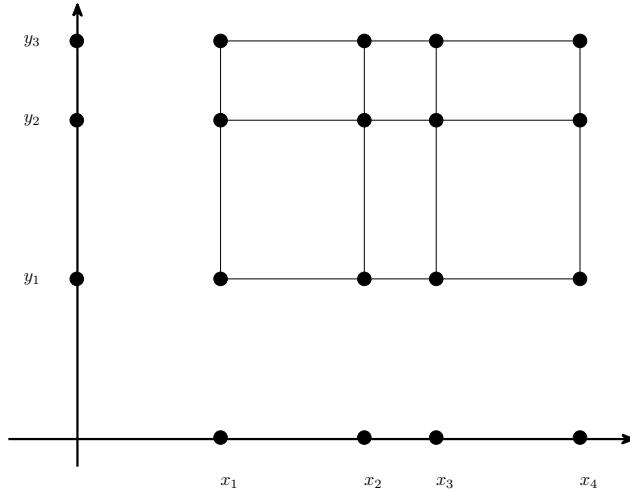


Figure 10.1: A Cartesian grid of nodes

Suppose that we possess a linear interpolation scheme for the nodes  $x_1, x_2, \dots, x_p$ . This will be a *univariate* process. We want to think of this as a linear operator  $P$  of the form

$$(Pf)(x) = \sum_{i=1}^p f(x_i)u_i(x) \quad (10.1.3)$$

in which the functions  $u_i$  have the *cardinal property*

$$u_i(x_j) = \delta_{ij}, \quad i, j = 1, \dots, p. \quad (10.1.4)$$

In the case of ordinary univariate interpolation, these functions  $u_i$  are given by basic Lagrange polynomials

$$u_i(x) = \ell_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}, \quad i = 1, \dots, p. \quad (10.1.5)$$

Notice that the operator  $P$  can be extended in a trivial manner to operate on functions of two or more variables. Thus, if  $f$  is a function of  $(x, y)$ , we can write

$$(\overline{P}f)(x, y) = \sum_{i=1}^p f(x_i, y)u_i(x). \quad (10.1.6)$$

One can see immediately that  $\overline{P}f$  is a function of two variables that interpolates  $f$  on vertical lines

$$L_i = \{(x_i, y) : -\infty < y < \infty\}, \quad i = 1, \dots, p. \quad (10.1.7)$$

Suppose that another operator is available for interpolation at the nodes  $y_1, y_2, \dots, y_q$ . We write

$$(Qf)(y) = \sum_{i=1}^q f(y_i)v_i(y), \quad (10.1.8)$$

where the functions  $v_i$  are any convenient ones having the cardinal property

$$v_i(y_j) = \delta_{i,j}, \quad 1 \leq i, j \leq q. \quad (10.1.9)$$

Again,  $Q$  can be extended to operate on bivariate functions by use of the equation

$$(\bar{Q}f)(x, y) = \sum_{i=1}^q f(x, y_i)v_i(y). \quad (10.1.10)$$

The function  $\bar{Q}f$  interpolates  $f$  on all the horizontal lines

$$L^i = \{(x, y_i) : -\infty < x < \infty\}, \quad i = 1, \dots, q. \quad (10.1.11)$$

### 10.1.3 Boolean sum and tensor product

There are two useful bivariate interpolation operators that can now be constructed from  $\bar{P}$  and  $\bar{Q}$ ; they are the *product*  $\bar{P}\bar{Q}$ , and the *Boolean sum*  $\bar{P} \oplus \bar{Q}$ , defined by

$$\bar{P} \oplus \bar{Q} = \bar{P} + \bar{Q} - \bar{P}\bar{Q}. \quad (10.1.12)$$

More detailed formulae for these operators are easily derived from the definitions of  $\bar{P}$  and  $\bar{Q}$ . Thus

$$\begin{aligned} (\bar{P}\bar{Q}f)(x, y) &= \bar{P}(\bar{Q}f)(x, y) = \sum_{i=1}^p (\bar{Q}f)(x_i, x)u_i(x) \\ &= \sum_{i=1}^p \sum_{j=1}^q f(x_i, y_j)v_j(y)u_i(x). \end{aligned} \quad (10.1.13)$$

Since  $v_j(y_k)u_i(x_\ell) = \delta_{jk}\delta_{i\ell}$ , we see without difficulty that  $\bar{P}\bar{Q}$  is a function that interpolates  $f$  at all nodes  $(x_i, y_j)$ . The *tensor product* notation  $P \otimes Q$  is also used for the operator  $\bar{P}\bar{Q}$ .

In the same way, a formula for  $\bar{P} \oplus \bar{Q}$  is

$$\begin{aligned} [(\bar{P} \oplus \bar{Q})f](x, y) &= (\bar{P}f)(x, y) + (\bar{Q}f)(x, y) - (\bar{P}\bar{Q}f)(x, y) \\ &= \sum_{i=1}^p f(x_i, y)u_i(x) + \sum_{j=1}^q f(x, y_j)v_j(y) - \sum_{i=1}^p \sum_{j=1}^q f(x_i, y_j)u_i(x)v_j(y). \end{aligned} \quad (10.1.14)$$

It is left as a problem to prove that the function  $(\bar{P} \oplus \bar{Q})f$  interpolates  $f$  on all the vertical lines  $L_i$ ,  $i = 1, \dots, p$  and all horizontal lines  $L^j$ ,  $j = 1, \dots, q$ .

If  $R_{\bar{P}}$  and  $R_{\bar{Q}}$  are the rests (errors) for the univariate interpolants, then the rests for tensor product and Boolean sum are

$$R_{\bar{P}\bar{Q}} = R_{\bar{P}} \oplus R_{\bar{Q}},$$

and

$$R_{\bar{P} \oplus \bar{Q}} = R_{\bar{P}}R_{\bar{Q}},$$

respectively (see [17, 89, 98]).

**Example 10.1.1.** Give a formula for a polynomial in two variables for the following data set

$(x, y)$	(1,1)	(2,1)	(4,1)	(5,1)	(1,3)	(2,3)
$f(x, y)$	1.7	-4.1	-3.2	4.9	6.1	-4.2
$(x, y)$	(4,3)	(5,3)	(1,4)	(2,4)	(4,4)	(5,4)
$f(x, y)$	2.3	7.5	-5.9	3.8	-1.7	2.5

Observe first that the nodes form a Cartesian grid, and the tensor product method is applicable. The functions  $u_i$  and  $v_j$  are given by Equation (10.1.5). In this example they are as follows:

$$\begin{aligned} u_1(x) &= \frac{x-2}{1-2} \cdot \frac{x-4}{1-4} \cdot \frac{x-5}{1-5} = -\frac{1}{12}(x-2)(x-4)(x-5) \\ u_2(x) &= \frac{1}{6}(x-1)(x-4)(x-5) \\ u_3(x) &= -\frac{1}{6}(x-1)(x-2)(x-5) \\ u_4(x) &= \frac{1}{12}(x-1)(x-2)(x-5) \\ v_1(y) &= \frac{y-3}{1-3} \cdot \frac{y-4}{1-4} = \frac{1}{6}(y-3)(y-4) \\ v_2(y) &= -\frac{1}{2}(y-1)(y-4) \\ v_3(y) &= \frac{1}{3}(y-1)(y-3) \end{aligned}$$

The polynomial interpolant is then

$$\begin{aligned} F(x, y) &= u_1(x) [1.7v_1(y) + 6.1v_2(y) - 5.9v_3(y)] \\ &\quad + u_2(x) [-4.1v_1(y) - 4.2v_2(y) + 3.8v_3(y)] \\ &\quad + u_3(x) [-3.2v_1(y) + 2.3v_2(y) - 1.7v_3(y)] \\ &\quad + u_4(x) [4.9v_1(y) + 7.5v_2(y) + 2.5v_3(y)] \end{aligned} \tag{10.1.15}$$

◇

If the function  $F$  in the preceding example is written as a sum of terms  $x^i y^j$ , the following 12 terms appear

$$1, x, x^2, x^3, y, xy, x^2y, x^3y, y^2, xy^2, x^3y^2. \tag{10.1.16}$$

Thus, we are interpolating by means of a 12-dimensional subspace of bivariate polynomials. The proper notation for this subspace is  $\Pi_3 \otimes \Pi_2$ . This is the *tensor product* of two linear spaces, and consists of all functions of the form

$$(x, y) \mapsto \sum_{i=1}^m a_i(x)b_i(y)$$

in which  $a_i \in \Pi_3$  and  $b_i \in \Pi_2$ . (The sum can have any number of terms). It is not difficult to prove a basis for this space consists of the functions in (10.1.16).

It is to be emphasized that the theory just outlined applies to general functions  $u_i$  and  $v_i$ , not just to polynomials. All that is needed is the cardinality property. (In an abstract theory, one works directly with the operators  $P$  and  $Q$ ; their detailed structure does not enter the analysis.)

In the tensor product method of polynomial interpolation, the general case will involve bivariate polynomials from the space  $\Pi_{p-1} \otimes \Pi_{q-1}$ , where  $p$  and  $q$  are the number of points that figure in Equation (10.1.2). A basis for this space is given by the functions

$$(x, y) \mapsto x^i y^j, \quad i = 0, \dots, p-1, \quad j = 0, \dots, q-1. \quad (10.1.17)$$

A generic element of the space is then of the form

$$(x, y) \mapsto \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} c_{ij} x^i y^j.$$

The *degree* of a term  $x^i y^j$  is defined to be  $i + j$ . Thus the space  $\Pi_{p-1} \otimes \Pi_{q-1}$  will contain one basis element of degree  $p + q - 2$ , namely  $x^{p-1} y^{q-1}$ , but it will not contain all terms of degree  $p + q - 2$ . For example a term such as  $x^p y^{q-2}$  will not be present. The degree of a polynomial in  $(x, y)$  is defined to be the largest degree of the terms present in the polynomial. The space of all bivariate polynomials of degree at most  $k$  will be denoted here by  $\Pi_k(\mathbb{R}^2)$ . A typical element of  $\Pi_k(\mathbb{R}^2)$  is a function of the form

$$(x, y) \mapsto \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} c_{ij} x^i y^j = \sum_{0 \leq i+j \leq k} c_{ij} x^i y^j. \quad (10.1.18)$$

**Theorem 10.1.2.** *A basis for  $\Pi_k(\mathbb{R}^2)$  is the set of functions*

$$(x, y) \mapsto x^i y^j, \quad 0 \leq i + j \leq k.$$

*Proof.* It is clear that this set spans  $\Pi_k(\mathbb{R}^2)$ , and it is only necessary to prove its linear independence. Suppose therefore that the function in equation (10.1.18) is 0. If  $y$  is assigned a fixed value, say  $y = y_0$ , then the equation

$$\sum_{i=0}^k \left( \sum_{j=0}^{k-i} c_{ij} y_0^j \right) x^i = 0$$

exhibits an apparent linear dependence among the functions  $x \mapsto x^i$ . Since this set of functions is linearly independent, we conclude that

$$\sum_{j=0}^{k-i} c_{ij} y_0^j = 0, \quad i = 0, \dots, k.$$

In this equation  $y_0$  can be *any* point. By the linear independence of the set of functions

$$y \mapsto y^j, \quad j = 0, \dots, k$$

we conclude that  $c_{ij} = 0$  for all  $i$  and  $j$ .  $\square$

**Corollary 10.1.3.** *The dimension of  $\Pi_k(\mathbb{R}^2)$  is  $\frac{1}{2}(k+1)(k+2)$ .*

*Proof.* The basis elements of  $\Pi_k(\mathbb{R}^2)$  given in Theorem 10.1.2 can be arrayed as follows:

$$\begin{array}{ccccccc} & x^k & & & & & \\ & x^{k-1} & x^{k-1}y & & & & \\ & x^{k-2} & x^{k-2}y & x^{k-2}y^2 & & & \\ & \vdots & \vdots & \vdots & \ddots & & \\ x^0 & x^0y & x^0y^2 & \cdots & x^0y^k & & \end{array}$$

The number of basis elements is thus

$$1 + 2 + \cdots + (k+1) = \frac{1}{2}(k+1)(k+2).$$

□

The MATLAB Sources 10.1 and 10.2 gives the implementation of the bivariate Lagrange tensor product interpolant and of the bivariate Lagrange Boolean sum interpolant, respectively.

---

### MATLAB Source 10.1 Bivariate tensor product Lagrange interpolant

---

```
function Z=tensprod(u,v,x,y,f)
%TENSPROD - tensor product Lagrange interpolant
%call [Z,X,Y]=TENSPROD(U,V,X,Y,F)
%U - evaluation abscissas
%V - evaluation ordinates
%X - node abscissas
%Y - node ordinates
%F - function
[X,Y]=meshgrid(x,y);
F=f(X,Y);
lu=pfl2b(x,u)';
lv=pfl2b(y,v)';
Z=lu*F*lv;
```

---



---

### MATLAB Source 10.2 Bivariate Boolean sum Lagrange interpolant

---

```
function Z=boolsum(u,v,x,y,f)
%BOOLSUM - Boolean sum Lagrange interpolant
%call [Z,X,Y]=BOOLSUM(U,V,X,Y,F)
%U - evaluation abscissas
%V - evaluation ordinates
%X - node abscissas
%Y - node ordinate
%F - function
[X,Y]=meshgrid(x,y);
F=f(X,Y);
[X1,V1]=meshgrid(x,v);
F1=f(X1,V1);
[U2,Y2]=meshgrid(u,y);
F2=f(U2,Y2);
lu=pfl2b(x,u);
lv=pfl2b(y,v);
Z=F1*lu+lv'*F2-lu'*F*lv;
```

---

**Example 10.1.4.** Consider the function  $f : [-2, 2] \times [-2, 2] \rightarrow \mathbb{R}$ ,  $f(x, y) = xe^{-x^2-y^2}$ . Its graph, and the graphs of tensor product, Boolean sums and their rests are given in Figure 10.2. We considered five equally spaced nodes on each axis,  $x_k, y_k = -2 + k$ ,  $k = \overline{0, 4}$ . ◇

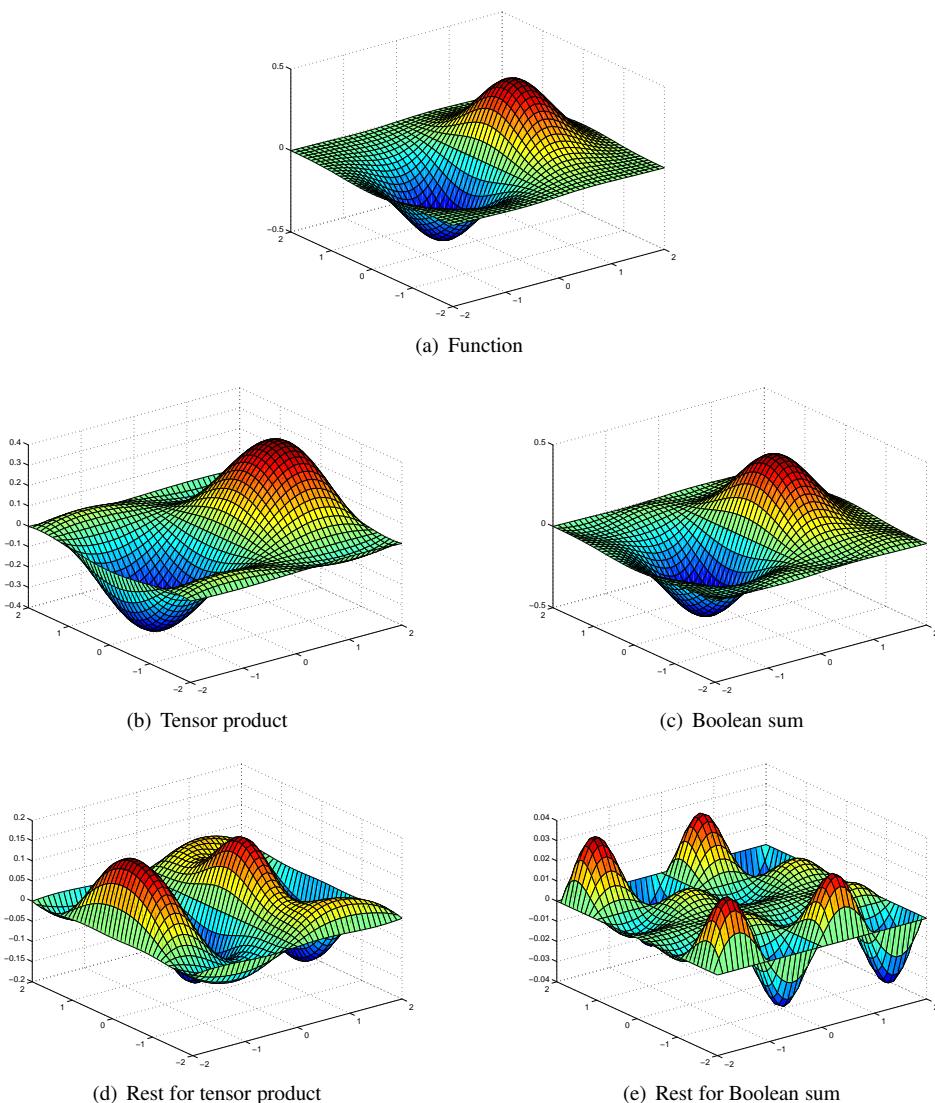


Figure 10.2: Graph of  $f : [-2, 2] \times [-2, 2] \rightarrow \mathbb{R}$ ,  $f(x, y) = xe^{-x^2-y^2}$  in Example 10.1.4 (Figure 10.2(a)), tensor product approximation (Figure 10.2(b)), Boolean sum approximation (Figure 10.2(c)) and the corresponding rest terms (Figure 10.2(d) and 10.2(e), respectively)

### 10.1.4 Geometry

Recall that in the one-variable case,  $\Pi_k$  can be used for interpolation at *any* set of  $k + 1$  nodes in  $\mathbb{R}$ . It is natural to expect that for two variables,  $\Pi_k(\mathbb{R}^2)$  can be used to interpolate at any set of  $n \equiv \frac{1}{2}(k+1)(k+2)$  nodes. This expectation is not fulfilled, however, and a simple example will show this. Suppose that  $k = 1$ , so that  $n = 3$ . A generic element of  $\Pi_1(\mathbb{R}^2)$  has the form

$$c_0 + c_1x + c_2y.$$

If we attempt to solve an interpolation problem with three nodes  $(x_i, y_i)$  we are led to a linear system whose coefficient determinant is

$$\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix}.$$

This determinant will be zero when the nodes are collinear. In that case the interpolation problem will be (in general) insoluble.

The preceding considerations indicate that the geometry of the node set  $\mathcal{N}$  will determine whether interpolation by  $\Pi_k(\mathbb{R}^2)$  is possible on  $\mathcal{N}$ . Of course, the number of nodes should be  $n = \frac{1}{2}(k+1)(k+2)$ . Some theorems concerning this question will be given to illustrate what is known. The first is due to Micchelli [64]. Its proof uses Bézout's Theorem. The Theorem of Bézout states that if  $p \in \Pi_k(\mathbb{R}^2)$ , if  $q \in \Pi_m(\mathbb{R}^2)$ , and if  $p^2 + q^2$  has more than  $km$  zeros, then  $p$  and  $q$  must have a common nonconstant factor.

**Theorem 10.1.5 (Micchelli, 1986).** *Interpolation of arbitrary data by the subspace  $\Pi_k(\mathbb{R}^2)$  is possible on a set of  $\frac{1}{2}(k+1)(k+2)$  nodes if the nodes lie on lines  $L_0, L_1, \dots, L_k$  in such a way that (for each  $i$ )  $L_i$  contains exactly  $i + 1$  nodes.*

*Proof.* Let  $\mathcal{N}$  denotes the set of nodes. Assuming the hypothesis of the theorem, we have  $\text{card}(\mathcal{N} \cap L_i) = i + 1$ . The sets  $\mathcal{N} \cap L_i$  must be pairwise disjoint, for if they were not, the following contradiction would arise:

$$\text{card}(\mathcal{N}) < \sum_{i=1}^k \text{card}(\mathcal{N} \cap L_i) = \sum_{i=1}^k (i + 1) = \frac{1}{2}(k + 1)(k + 2).$$

Since the number of nodes is equal to the dimension of space  $\Pi_k(\mathbb{R}^2)$ , it suffices to show that the homogeneous interpolation problem has only the 0 solution. Accordingly, let  $p \in \Pi_k(\mathbb{R}^2)$ , and suppose that  $p(x, y) = 0$  for each point  $(x, y)$  in  $\mathcal{N}$ . For each  $i$ , let  $\ell_i$  be a linear function describing  $L_i$ :

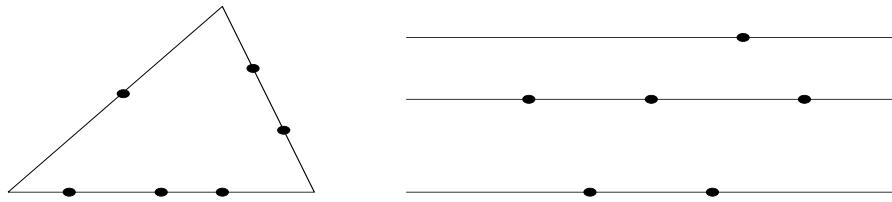
$$L_i = \{(x, y) : \ell_i(x, y) = 0\}, \quad 0 \leq i \leq k.$$

Notice that  $p^2 + \ell_k^2$  has at least  $k + 1$  zeros, namely the points in  $\mathcal{N} \cap L_k$ . Bézout's Theorem allows us to conclude that  $\ell_k$  is a divisor of  $p$ . This argument can be repeated, for  $(p/\ell_k)^2 + \ell_{k-1}^2$  has at least  $k$  zeros, and  $\ell_{k-1}$  must be a divisor of  $p/\ell_k$ . After  $k$  steps in this argument, the conclusion is drawn that  $p$  is divisible by  $\ell_1 \ell_2 \dots \ell_k$ . Thus  $p$  is a scalar multiple of  $\ell_1 \ell_2 \dots \ell_k$  because  $p$  is of degree at most  $k$ . Since  $p$  vanishes on  $\mathcal{N} \cap L_0$  while  $\ell_1 \ell_2 \dots \ell_k$  does not,  $p$  must be 0.  $\square$

Because Bézout's Theorem is limited to  $\mathbb{R}^2$ , the same is true for Theorem 10.1.5. An algorithmic proof of Theorem 10.1.5, not requiring Bézout's Theorem, is given in Section 10.1.5.

**Example 10.1.6.** The sets of nodes in Figure 10.3 are suitable for interpolation by the space  $\Pi_2(\mathbb{R}^2)$ .  $\diamond$

A theorem closely related to Theorem 10.1.5 can be given in higher dimensional spaces,  $\mathbb{R}^d$ . The dimension of  $\Pi_k(\mathbb{R}^d)$  is  $\binom{d+k}{k}$ . The following result is from Chung and Yao [13].

Figure 10.3: Node sets for interpolation by  $\Pi_2(\mathbb{R}^2)$ 

**Theorem 10.1.7.** Let  $k$  and  $d$  be given and set  $n = \binom{d+k}{k}$ . Let a set of  $n$  nodes  $z_1, z_2, \dots, z_n$  be given in  $\mathbb{R}^d$ . If there exist hyperplanes  $H_{ij}$  in  $\mathbb{R}^d$ , with  $1 \leq i \leq n$  and  $1 \leq j \leq k$ , such that

$$z_j \in \bigcup_{\nu=1}^k H_{i\nu} \iff j \neq i, \quad (1 \leq i, j \leq n) \quad (10.1.19)$$

then arbitrary data on the node set can be interpolated by polynomials in  $\Pi_k(\mathbb{R}^d)$ .

*Proof.* Each hyperplane is the kernel of a nonzero linear function, and we write

$$H_{ij} = \left\{ z \in \mathbb{R}^d : \ell_{ij}(z) = 0 \right\},$$

where  $\ell_{ij} \in \Pi_1(\mathbb{R}^d)$ . Define the function

$$q_i(z) = \prod_{j=1}^k \ell_{ij}(z), \quad i = 1, \dots, n.$$

Now  $z_i$  does not belong to any of the hyperplanes  $H_{i1}, H_{i2}, \dots, H_{ik}$ , by Condition (10.1.19), and therefore  $\ell_{ij}(z_i) \neq 0$ , for  $j = 1, \dots, k$ . This proves that  $q_i(z_i) \neq 0$ .

Again, by Condition (10.1.19), if  $j \neq i$ , then  $z_j \in H_{i\nu}$  for some  $\nu$ , and consequently  $\ell_{i\nu}(z_j) = 0$  and  $q_i(z_j) = 0$ . We define  $p_i(z) = q_i(z)/q_i(z_i)$ ; it holds  $p_i(z_j) = \delta_{ij}$ . Since  $p_i \in \Pi_k(\mathbb{R}^d)$ , we have a Lagrangian formula for interpolating a function  $f$  at the nodes by a polynomial of degree  $k$ :

$$P(z) = \sum_{i=1}^n f(z_i) p_i(z).$$

□

A node configuration satisfying the hypotheses of Theorem 10.1.7 is shown in Figure 10.4. Here the dimension is  $d = 2$ , the degree is  $k = 2$ , and the number of nodes is  $n = 6$ .

A very weak result concerning polynomial interpolation at arbitrary set of nodes is the following.

**Theorem 10.1.8.** The space  $\Pi_k(\mathbb{R}^2)$  is capable of interpolating arbitrary data on any set of  $k + 1$  distinct nodes in  $\mathbb{R}^2$ .

*Proof.* If the nodes are  $(x_i, y_i)$ , with  $i = 1, \dots, k$ , we select a linear function

$$\ell(x, y) = ax + by + c$$

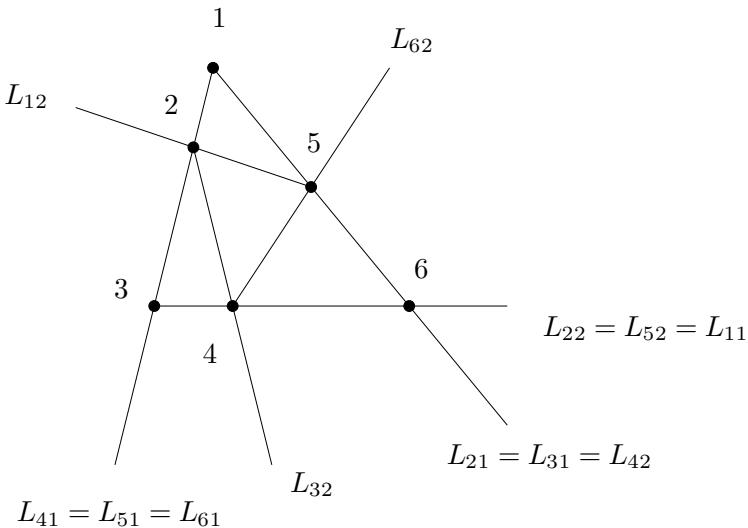


Figure 10.4: Illustrating Theorem 10.1.7

such that the  $k + 1$  numbers  $t_i = \ell(x_i, y_i)$  are all different (show that this is possible). If  $f$  is the function to be interpolated, we find  $p \in \Pi_k(\mathbb{R})$  such that  $p(t_i) = f(x_i, y_i)$ . Then  $p \circ \ell \in \Pi_k(\mathbb{R}^2)$  and

$$(p \circ \ell)(x_i, y_i) = p(\ell(x_i, y_i)) = p(t_i) = f(x_i, y_i).$$

□

A function of the form  $f \circ \ell$ , where  $\ell \in \Pi(\mathbb{R}^2)$  is called a *ridge* function. Its graph is a ruled surface, since  $f \circ \ell$  remains constant on each line  $\ell(x, y) = \lambda$ . Theorem 10.1.8 can be easily extended to  $\mathbb{R}^d$ .

### 10.1.5 A Newtonian scheme

For the practical implementation of any interpolation method, it is advantageous to have an algorithm like Newton's procedure in univariate polynomial interpolation. Recall that one feature of the Newton scheme is that from a polynomial  $p$  interpolating  $f$  at nodes  $x_1, x_2, \dots, x_n$ , we can easily obtain a polynomial  $p^*$  interpolating  $f$  at nodes  $x_1, x_2, \dots, x_n, x_{n+1}$  by adding one term to  $p$ . Indeed, we put

$$\begin{aligned} q(x) &= (x - x_1)(x - x_2) \cdots (x - x_n) \\ p^*(x) &= p(x) + cq(x) \\ c &= [f(x_{n+1}) - p(x_{n+1})]/q(x_{n+1}). \end{aligned}$$

The advantage of this algorithm is that an interpolating polynomial can be constructed step by step, adding one new interpolation node and one term to  $p$  in each stage.

The abstract form of this procedure is as follows. Let  $X$  be a set and  $f$  a real-valued function defined on  $X$ . Let  $\mathcal{N}$  be a set of nodes. If  $p$  is any function that interpolates  $f$  on  $\mathcal{N}$ , and if  $q$  is any function that vanishes on  $\mathcal{N}$ , then a function  $p^*$  interpolating  $f$  on  $\mathcal{N} \cup \{\xi\}$  can be obtained in form  $p^* = p + cq$ , provided that  $q(\xi) \neq 0$ .

A more general version of this strategy deals with set of nodes. Let  $q$  be a function from  $X$  to  $\mathbb{R}$  and let  $Z$  be its zero set. If  $p$  interpolates  $f$  on  $\mathcal{N} \cap Z$  and  $r$  interpolates  $(f - p)/q$  on  $\mathcal{N} \setminus Z$ , then  $p + qr$  interpolates  $f$  on  $\mathcal{N}$ .

The procedure just outlined can be used to give an algorithmic proof of Theorem 10.1.5. To begin, select  $p_k \in \Pi_k(\mathbb{R}^2)$  that interpolates  $f$  on  $\mathcal{N} \cap L_k$ . (Use Theorem 10.1.8). Proceeding inductively downward, suppose that  $p_i$  has been found in  $\Pi_i(\mathbb{R}^2)$  and interpolates  $f$  on all the nodes in  $L_k \cup L_{k-1} \cup \dots \cup L_i$ . We shall attempt to construct  $p_{i-1}$  in *Newton form*

$$p_{i-1} = p_i + r\ell_k\ell_{k-1}\dots\ell_i.$$

It is clear that  $p_{i-1}$  will still interpolate  $f$  at the nodes in  $L_k \cup L_{k-1} \cup \dots \cup L_i$ , since the term added to  $p_i$  vanishes on this set. In order to make  $p_{i-1}$  interpolate  $f$  on the nodes in  $L_{i-1}$ , we write

$$f(x) = p_i(x) + r(x)(\ell_k\ell_{k-1}\dots\ell_i)(x), \quad x \in \mathcal{N} \cap L_{i-1}$$

from which we infer that  $r$  should interpolate  $(f - p_i)/(\ell_k\ell_{k-1}\dots\ell_i)$  on  $\mathcal{N} \cap L_{i-1}$ . By Theorem 10.1.8, there is an  $r \in \Pi_{i-1}(\mathbb{R}^2)$  that does so. Finally, observe that  $p_{i-1} \in \Pi(R^2)$  because  $r$  is of degree  $k - i + 1$ . This algorithm was given by Micchelli [64].

It is an interesting fact that no  $n$ -dimensional subspace in  $C(\mathbb{R}^2)$  can serve for interpolation at arbitrary sets of  $n$  nodes (excepts in the trivial case of  $n = 1$ ). This was probably first noticed by Haar in 1918, and his argument goes like this. Suppose that  $n$  functions  $u_1, u_2, \dots, u_n$  are given in  $C(\mathbb{R}^2)$ . Let  $n$  nodes in  $\mathbb{R}^2$  be given, say  $p_i = (x_i, y_i)$ . If we wish to interpolate at these nodes using the base functions  $u_i$ , we shall have to solve a linear system whose determinant is

$$D = \begin{vmatrix} u_1(p_1) & u_2(p_1) & \dots & u_n(p_1) \\ u_1(p_2) & u_2(p_2) & \dots & u_n(p_2) \\ \vdots & \vdots & \ddots & \vdots \\ u_1(p_n) & u_2(p_n) & \dots & u_n(p_n) \end{vmatrix}.$$

This determinant may be nonzero for the given set of nodes. However, let the first two of the nodes undergo a continuous motion in  $\mathbb{R}^2$  in such a way that during the motion these two points never coincide, nor do they coincide with any of the other nodes, yet at the end of the motion they have exchanged their original positions. By the rule of determinants, the determinant  $D$  will have changed sign (because rows 1 and 2 are interchanged). By continuity,  $D$  assumed the value zero during the continuous motion described. Hence,  $D$  will *sometimes* be zero, even for distinct nodes. The fact that two nodes can move in  $\mathbb{R}^2$  and exchange places without ever be coincident is characteristic of  $\mathbb{R}^2, \mathbb{R}^3, \dots$  but not of  $\mathbb{R}$ . This explains why interpolation in  $\mathbb{R}^2, \mathbb{R}^3, \dots$  must be approached somewhat differently from  $\mathbb{R}^1$ . What is usually done is to fix the nodes *first* and *then* to ask what subspaces of interpolating functions are suitable.

## 10.1.6 Shepard interpolation

A very general method of this type (in which the subspace depend on the nodes) is known as Shepard interpolation, after its originator, Shepard [85]. Let the (distinct) nodes be listed as

$$p_i = (x_i, y_i), \quad i = 1, \dots, n. \quad (10.1.20)$$

We shall use  $p$  and  $q$  to denote generic elements in  $\mathbb{R}^2$ , as this will make extensions to  $\mathbb{R}^2, \mathbb{R}^3, \dots$  conceptually transparent. Next, we select a real-valued function  $\phi$  on  $\mathbb{R}^2 \times \mathbb{R}^2$  subject to the sole condition that

$$\phi(p, q) = 0 \iff p = q. \quad (10.1.21)$$

Examples that come to mind are  $\phi(p, q) = \|p - q\|$  and  $\phi(p, q) = \|p - q\|^2$ . Next, we set up some cardinal functions in exact analogy with the Lagrange formulae in univariate approximation. This is done as follows:

$$u_i(p) = \prod_{\substack{j=1 \\ j \neq i}}^n \frac{\phi(p, p_j)}{\phi(p_i, p_j)}, \quad i = 1, \dots, n. \quad (10.1.22)$$

It is easy to see that these functions have the cardinality property

$$u_i(p_j) = \delta_{ij}, \quad i, j = 1, \dots, n.$$

This is a consequence of the hypothesis in (10.1.21). It follows that an interpolant to  $f$  at the given nodes is provided by the function

$$F = \sum_{i=1}^n f(p_i) u_i. \quad (10.1.23)$$

**Example 10.1.9.** Find the formulae for Shepard interpolation when  $\|p - q\|^2$  is used for  $\phi(p, q)$ .

Let  $p_i = (x_i, y_i)$ ,  $p = (x, y)$  and

$$\phi(p, p_j) = \|p - p_j\|^2 = (x - x_j)^2 + (y - y_j)^2.$$

Then

$$F(x, y) = \sum_{i=1}^n f(x_i, y_i) \prod_{\substack{j=1 \\ j \neq i}}^n \frac{(x - x_j)^2 + (y - y_j)^2}{(x_i - x_j)^2 + (y_i - y_j)^2}. \quad \diamond$$

Another version of Shepard's method starts with the additional assumption on  $\phi$  that is a *nonnegative* function. Next, let

$$v_i(p) = \prod_{\substack{j=1 \\ j \neq i}}^n \phi(p, p_j) \quad v(p) = \sum_{i=1}^n v_i(p) \quad w_i(p) = v_i(p)/v(p). \quad (10.1.24)$$

By our assumptions on  $\phi$ , we have  $v_i(p_j) = 0$  if  $i \neq j$  and  $v_i(p) > 0$  for all points except  $p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n$ . It follows that  $v(p) > 0$  and that  $w_i$  is well defined. By the construction,  $w_i(p_j) = \delta_{ij}$  and  $0 \leq w_i(p_j) \leq 1$ . Furthermore,  $\sum_{i=1}^n w_i(p) = 1$ . The interpolation process is given by the equation

$$F = \sum_{i=1}^n f(p_i) w_i = \sum_{i=1}^n f(p_i) v_i/v(p). \quad (10.1.25)$$

This process has two favorable properties not possessed by the previous version; namely if the data are nonnegative, then the interpolant  $F$  will be a nonnegative function, and if  $f$  is a constant function, then  $F = f$  ( $F$  reproduces the constants). This two properties give evidence that the interpolant  $F$  inherits certain characteristics of the function being interpolated. On the other hand, if  $\phi$  is differentiable, then  $F$  will exhibit a flat spot at each node. This is because  $0 \leq w_i \leq 1$  and  $w_i(p_j) = \delta_{ij}$ , so that the nodes are extrema (maximum points or minimum points) of each  $w_i$ . Thus the partial derivatives of  $w_i$  are zero at each node, and consequently the same is true of  $F$ .

An important case of Shepard interpolation arises when the function  $\phi$  is a power of the Euclidian distance:

$$\phi(x, y) = \|x - y\|^\mu \quad (\mu > 0).$$

Here  $x$  and  $y$  can be points in  $\mathbb{R}^s$ . It suffices to examine the simpler function  $g(x) = \|x\|^\mu$  at the questionable point  $x = 0$ . The directional derivative of  $g$  at zero is obtained by differentiating the

function  $G(t) = \|tu\|^\mu$ , where  $u$  is a unit vector defining the direction. Since  $G(t) = |t|^\mu$ , the derivative at  $t = 0$  does not exist when  $0 < \mu \leq 1$ , but for  $\mu > 1$ ,  $G'(0) = 0$ .

The formula for  $w_i$  can be given in two ways:

$$w_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^n \|x - x_j\|^\mu / \sum_{k=1}^n \prod_{\substack{j=1 \\ j \neq i}}^n \|x - x_j\|^\mu, \quad (10.1.26)$$

$$w_i(x) = \|x - x_i\|^{-\mu} / \sum_{j=1}^n \|x - x_j\|^{-\mu}. \quad (10.1.27)$$

The second equation must be used with care since the right hand sides assumes the indeterminate form  $\infty/\infty$  at  $x_i$ .

The MATLAB source 10.3 gives an implementation for Shepard interpolation, based on formula (10.1.26).

---

### MATLAB Source 10.3 Shepard interpolation

---

```
function z=Shepgrid(xp,yp,x,y,f,mu)
%SHEPGRID - computes Shepard interpolant values on a grid
% call Z=SHEPGRID(XP,YP,X,Y,F,MU)
% XP,YP - points
% X,Y - node coordinates
% F function value on nodes
% MU - exponent
if size(xp) ~= size(yp)
    error('xp and yp have not the same size')
end
[m,n]=size(xp);
for i=1:m
    for j=1:n
        z(i,j)=Sheplpt(xp(i,j),yp(i,j),x,y,f,mu);
    end
end
function z=Sheplpt(xp,yp,x,y,f,mu)
%SHEP1PT - computes Shepard interpolant value in one point
%call Z=SHEP1PT(XP,YP,X,Y,F,MU)
% XP,YP - the point
% X,Y - node coordinates
% F function value on nodes
% MU - exponent
d=(sqrt((xp-x).^2+(yp-y).^2)).^mu;
n=length(x);
A=zeros(size(f));
for i=1:n
    A(i)=prod(d([1:i-1,i+1:n]));
end
z=sum(A.*f)/sum(A);
```

---

**Example 10.1.10.** Plot the Shepard interpolant for the function in Example 10.1.4,  $f : [-2, 2] \times [-2, 2] \rightarrow \mathbb{R}$ ,  $f(x, y) = xe^{-x^2-y^2}$ . The graph of Shepard interpolant, for  $\mu = 2$  and  $\mu = 3$  and 100 random nodes, is given in figure 10.5. The MATLAB sequence for the left figure is

```
>> P=4*rand(2,100)-2;
>> f=P(1,:).*exp(-P(1,:).^2-P(2,:).^2);
>> [X,Y]=meshgrid(linspace(-2,2,50));
>> z2=Shepgrid(X,Y,P(1,:),P(2,:),f,2);
>> surf(X,Y,z2)
```

and analogously for the right figure.  $\diamond$

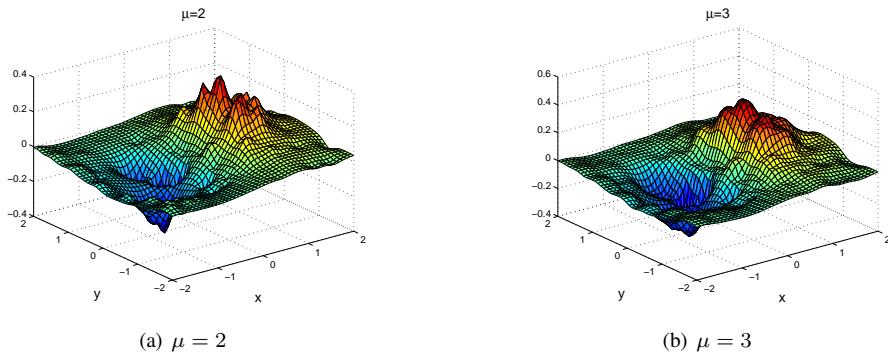


Figure 10.5: Graph of Shepard interpolant for function in Example 10.5 for  $\mu = 2$  (left) and  $\mu = 3$

A local multivariate interpolation method of Franke and Little is designed so that the datum at one node will have a very small influence on the interpolation function at points far from that node. Given nodes  $(x_i, y_i)$ ,  $i = 1, \dots, n$ , we introduce functions

$$g_i(x, y) = \left(1 - r_i^{-1} \sqrt{(x - x_i)^2 + (y - y_i)^2}\right)_+^\mu$$

The subscript ‘+’ indicates that when the quantity inside the parentheses is negative, it is replaced by 0. This occurs if  $(x, y)$  is far from the node  $(x_i, y_i)$ .

If  $r_i$  is chosen to be the distance from  $(x_i, y_i)$  to the nearest neighboring node, then  $g_i(x_j, y_j) = \delta_{ij}$ . In this case, we interpolate an arbitrary function  $f$  by means of the function

$$\sum_{i=1}^n f(x_i, y_i) g_i(x, y).$$

In the sequel we consider a slightly modified variant of Franke-Little weights. The basic functions are given by

$$\bar{w}_k(x) = \frac{\frac{(R - \|x - x_k\|)_+^\mu}{R^\mu \|x - x_k\|^\mu}}{\sum_{i=0}^n \frac{(R - \|x - x_i\|)_+^\mu}{R^\mu \|x - x_i\|^\mu}}, \quad (10.1.28)$$

where  $R > 0$  is a given constant, and the interpolant has the form

$$S(f, x) = \sum_{k=0}^n \bar{w}_k(x) f(x_k). \quad (10.1.29)$$

The function `Shepgridbloc`, given in MATLAB Source 10.4, computes the interpolant given by (10.1.29) and (10.1.28).

---

#### MATLAB Source 10.4 Local Shepard interpolation

---

```
function z=Shepgridbloc(xp,yp,x,y,f,mu,R)
%computes local Shepard interpolant values on a grid
% using the barycentric form and Franke-Little weights
% call z=Shepgridbloc(xp,yp,x,y,f,mu,R)
% xp,yp - the points
% x,y - node coordinates
% f function value on nodes
% mu - exponent
% R - Radius
if size(xp) ~=size(yp)
    error('xp and yp have not the same size')
end
[m,n]=size(xp);
for i=1:m
    for j=1:n
        z(i, j)=Sheplbarloc(xp(i, j),yp(i, j),x,y,f,mu,R);
    end
end
function z=Sheplbarloc(xp,yp,x,y,f,mu,R)
%computes local Shepard interpolant value in one point
d=(xp-x).^2+(yp-y).^2;
n=length(x);
w=zeros(1,n);
ix=(d<R);
w(ix)=((R-d(ix))./(R*d(ix))).^mu;
z=sum(w*f)/sum(w);
```

---

**Example 10.1.11.** We test `Shepgridbloc` for the function in Example 10.1.4,  $f : [-2, 2] \times [-2, 2] \rightarrow \mathbb{R}$ ,  $f(x, y) = xe^{-x^2-y^2}$ , with 201 random generated nodes,  $\mu = 2, 3$  and  $R = 0.4$ :

```
ftest=@(x,y) x.*exp(-x.^2-y.^2);
nX=[4*rand(201,1)-2];
nY=[4*rand(201,1)-2];
nX=nX(:); nY=nY(:); f=ftest(nX,nY);
[X,Y]=meshgrid(linspace(-2,2,113));
Z2a=Shepgridbloc(X,Y,nX,nY,f,2,0.4);
Z3a=Shepgridbloc(X,Y,nX,nY,f,3,0.4);
surf(X,Y,Z2a)
```

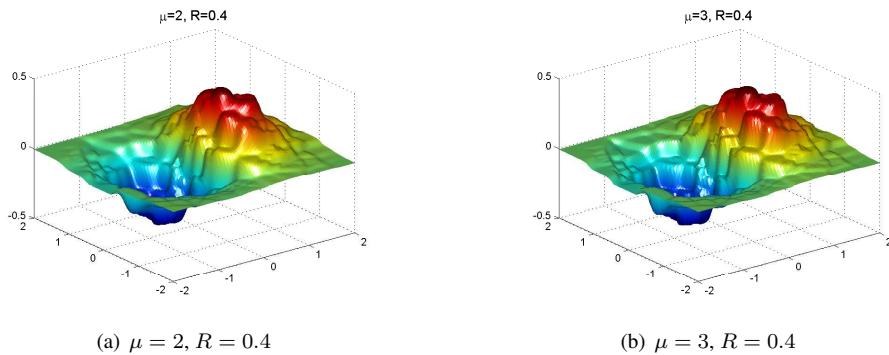


Figure 10.6: Local Shepard interpolants in Example 10.1.11

```

title('\mu=2, R=0.4','FontSize',14)
shading interp; camlight headlight
figure(2)
surf(X,Y,Z3a)
title('\mu=3, R=0.4','FontSize',14)
shading interp; camlight headlight

```

See Figure 10.6 for output.  $\diamond$

For details on local Shepard interpolation, including MATLAB implementation, see [97].

### 10.1.7 Triangulation

Another general strategy for interpolating functions given on  $\mathbb{R}^2$  begins by creating a triangulation. Informally, this means that triangles are drawn by joining nodes. In the end, we shall have a family of triangles,  $T_1, T_2, \dots, T_m$ . We consider this collection of triangles to be the triangulation. The following rules must be satisfied:

1. Each interpolation node must be the vertex of some triangle  $T_i$ .
2. Each vertex of a triangle in the collection must be a node.
3. If a node belongs to a triangle, it must be a vertex of that triangle.

The effect of Rule 3 is to disallow the construction shown in Figure 10.7.

The simplest type of interpolation on a triangulation is the piecewise linear function that interpolates a function  $f$  at all the vertices of all triangles. In any triangle,  $T_i$ , a linear function will be prescribed:

$$\ell_i(x, y) = a_i x + b_i y + c_i, \quad (x, y) \in T_i.$$

The coefficients in  $\ell_i$  are uniquely determined by the prescribed function values at the vertices of  $T_i$ . This can be seen as an application of Theorem 10.1.5, for  $L_1$  in that theorem can be taken to be one side of the triangle, and  $L_0$  can be a line parallel to  $L_1$  containing the vertex not on  $L_1$ . Let us consider the situation shown in Figure 10.8.

The line segment joining  $(x_2, y_2)$  to  $(x_3, y_3)$  is common to both triangles. This line segment can be represented as

$$\{t(x_2, y_2) + (1-t)(x_3, y_3) : 0 \leq t \leq 1\}.$$

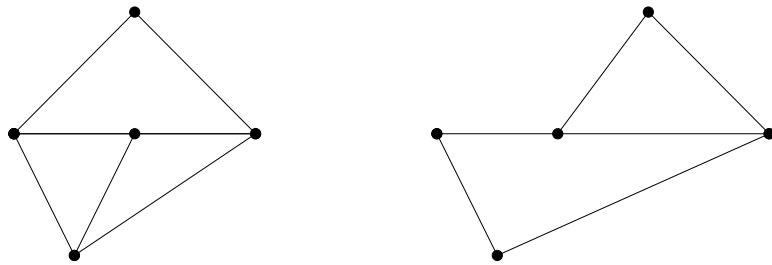


Figure 10.7: Illegal triangulations

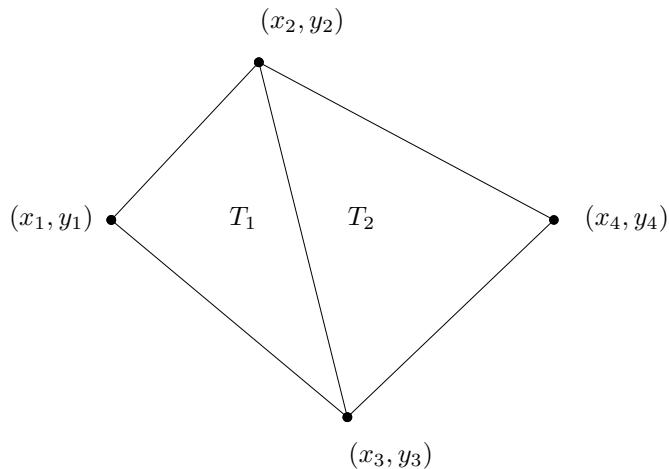


Figure 10.8: Continuity in a triangulation

The variable  $t$  can be considered to be the coordinate for the points on the line segment. The linear function  $\ell_1$ , when restricted to this line segment, will be a linear function of the single variable  $t$ , namely

$$a_1(tx_2 + (1-t)x_3) + b_1(ty_2 + (1-t)y_3) + c_1$$

or

$$(a_1x_2 - a_1x_3 + b_1y_2 - b_1y_3)t + (a_1x_3 + b_1y_3 + c_1).$$

This linear function of  $t$  is completely determined by the interpolation conditions at  $(x_2, y_2)$  and  $(x_3, y_3)$ . The same remarks pertain to the linear function  $\ell_2$ . Thus  $\ell_1$  and  $\ell_2$  agree on this line segment, and the piecewise linear function defined on  $T_1 \cup T_2$  is continuous. This proves the following result.

**Theorem 10.1.12.** *Let  $\{T_1, T_2, \dots, T_m\}$  be a triangulation in the plane. The piecewise linear function taking prescribed values at all the vertices of all the triangles  $T_i$  is continuous.*

Consider next the use of piecewise quadratic functions on triangulation. In each triangle,  $T_i$ , a

quadratic polynomial will be prescribed:

$$q_i(x, y) = a_1x^2 + a_2xy + a_3y^2 + a_4x + a_5y + a_6.$$

Six conditions will be needed to determine the six coefficients. One such set of conditions consists of values at the vertices of the triangle and the midpoints of the sides. Again, an application of Theorem 10.1.5 shows that this interpolation is always uniquely possible. Indeed, in that theorem,  $L_2$  can be one side of the triangle,  $L_1$  can be the line passing through two midpoints not on  $L_2$ , and  $L_0$  can be a line containing the remaining vertex but no other node. (See Figure 10.9). Reasoning as before, we see that the global piecewise quadratic function will be continuous because the three prescribed function values on the side of a triangle determine the quadratic function of one variable on that side.

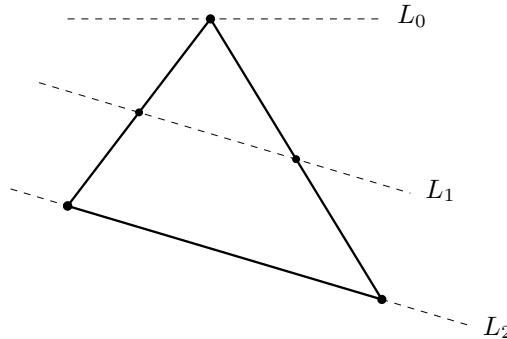


Figure 10.9: Applying Theorem 2

### 10.1.8 Moving least squares

Another versatile method of smoothing and interpolating multivariate functions is called *moving least squares*. First it is explained in a general setting, and then some specific examples will be given.

We start with a set  $X$  that is the domain of the functions involved. For example,  $X$  can be  $\mathbb{R}$ , or  $\mathbb{R}^2$ , or a subset of either. Next, a set of nodes  $\{x_1, x_2, \dots, x_n\}$  is given. These are the points at which a certain function  $f$  has been sampled. Thus the values  $f(x_i)$  are known, for  $i = 1, \dots, n$ . For purposes of approximation, we select a set of functions  $u_1, u_2, \dots, u_m$ . These are real-valued functions defined on  $X$ . The number  $m$  will usually be very small relative to  $n$ .

In the familiar least-squares method, a set of nonnegative weights  $w_i \geq 0$  is given. We try to find coefficients  $c_1, c_2, \dots, c_m$  to minimize the expression

$$\sum_{i=1}^n \left[ f(x_i) - \sum_{j=1}^m c_j u_j(x_i) \right]^2 w_i.$$

This is the sum of squares of the residuals. If we choose the discrete scalar product

$$\langle f, g \rangle = \sum_{i=1}^n w_i f(x_i) g(x_i),$$

the solution is characterized by the orthogonality condition

$$f - \sum_{j=1}^m c_j u_j \perp u_i, \quad i = 1, \dots, m.$$

This leads to the normal equations

$$\sum_{j=1}^m c_j \langle u_j, u_i \rangle = \langle f, u_i \rangle, \quad i = 1, \dots, m.$$

How does the moving least squares method differ from the previous procedure? The weights  $w_i$  are now allowed to be functions of  $x$ . The formalism of the usual least squares method can be retained, although the following notation may be better:

$$\langle f, g \rangle_x = \sum_{i=1}^n f(x_i)g(x_i)w_i(x).$$

The normal equations now should be written in the form

$$\sum_{j=1}^n c_j(x) \langle u_j, u_i \rangle_x = \langle f, u_i \rangle_x$$

and the final approximation will be

$$g(x) = \sum_{j=1}^m c_j(x)u_j(x).$$

The computation necessary to produce this function will be quite formidable if  $m$  is large, for the normal equation change with  $x$ . For this reason,  $m$  is usually no greater than 10.

The weight function can be used to achieve several desirable effects. First, if  $w_i(x)$  is “strong” at  $x_i$ , the function  $g$  will nearly interpolate  $f$  at  $x_i$ . In the limiting case,  $w_i(x_i) = +\infty$ , and  $g(x_i) = f(x_i)$ . If  $w_i(x)$  decreases rapidly to zero when  $x$  moves away from  $x_i$ , then the nodes far from  $x_i$  will have little effect on  $g(x_i)$ .

A choice for  $w_i$  that achieves these two objectives in a space  $\mathbb{R}^d$  is

$$w_i(x) = \|x - x_i\|^{-2}$$

where any norm can be employed, although the Euclidean norm is usual.

If the moving least squares procedure is used with a single function,  $u_1(x) \equiv 1$ , and with weight functions like the one just mentioned, then Shepard's method will result. To see that this is so, write the normal equation for this case, with  $c_1(x) = c(x)$ ,  $u_1(x) = u(x) = 1$ :

$$c(x) \langle u, u \rangle_x = \langle f, u \rangle_x.$$

The approximating function will be

$$\begin{aligned} g(x) &= c(x)u(x) = c(x) = \langle f, u \rangle_x \langle u, u \rangle_x \\ &= \sum_{i=1}^n f(x_i)w_i(x) / \sum_{j=1}^n w_j(x). \end{aligned}$$

If  $w_i(x) = \|x - x_i\|^{-2}$ , then after removing the singularities  $w_i / \sum_{j=1}^n w_j$  has the cardinal property: it takes the value 1 at  $x$  and the value zero at all other nodes.

### 10.1.9 Interpolation by radial basis functions

Consider a given function  $f \in C(\Omega)$ ,  $\Omega \subset \mathbb{R}^n$ , and a very large data set that consists of two parts: a finite set  $X = \{x_1, \dots, x_M\}$  of  $M$  scattered points (nodes) in  $\Omega$  and real numbers  $\{f_1, \dots, f_M\}$  (approximate values of  $f$  at given points). We want to interpolate  $f$ . We choose the following basis:

$$\phi : \mathbb{R}_+ \rightarrow \mathbb{R} \quad \Phi(x, y) = \phi(\|x - y\|_2).$$

This functions must be invariant to translation and rotation, and they are called *radial basis functions*. Reconstruction by interpolation on  $X$  will require to solve the linear system

$$\sum_{j=1}^M \alpha_j \Phi(x_k, x_j) = f_k, \quad k = 1, \dots, M, \quad (10.1.30)$$

for  $\alpha_1, \dots, \alpha_M$ , or in matrix form

$$A\alpha = f,$$

where  $A = (\Phi(x_k, x_j))_{1 \leq j, k \leq M}$ . To assure the uniqueness of (10.1.30),  $A$  must be nonsingular.

**Definition 10.1.13.** A function  $f$  is complete monotone on  $[0, \infty)$  if

- (a)  $f \in C[0, \infty)$ .
- (b)  $f \in C^\infty(0, \infty)$ .
- (c)  $(-1)^k f^{(k)}(t) \geq 0$  for  $t > 0$  and  $k = 0, 1, \dots$

**Theorem 10.1.14 (Schoenberg[11]).** If a function  $f$  is complete monotone and not constant on  $[0, \infty)$ , then for any distinct points  $x_1, \dots, x_n$  the matrix  $A = f(\|x_i - x_j\|^2)$  is positive definite (and therefore nonsingular).

Thus, Schoenberg's theorem provides a large class of functions for which interpolation is possible by expressions of the following form:

$$\sum_{j=1}^n c_j f(\|x - x_j\|).$$

Here are some examples

$$\begin{aligned} \phi(r) &= e^{-\beta r}, \quad \beta > 0 \text{ (Gaussian)} \\ \phi(r) &= (c^2 + r^2)^{\beta/2}, \quad \beta < 0 \text{ (inverse multiquadric)} \\ \phi(r) &= (1 - r)_+^4 (1 + 4r) \quad \text{(Wendland)} \end{aligned}$$

Another related result is

**Theorem 10.1.15 (Micchelli, [65]).** Suppose  $F'$  is complete monotone but not constant on  $(0, \infty)$ ,  $F$  is continuous on  $[0, \infty)$  and positive on  $(0, \infty)$ . Then for any distinct points  $x_1, \dots, x_n$  from  $\mathbb{R}^n$ , it holds

$$(-1)^{n-1} \det F(\|x_i - x_j\|^2) > 0.$$

Hence the matrix  $A_{ij} = F(\|x_i - x_j\|^2)$  is nonsingular. An example of function that fulfills the conditions of Micchelli's Theorem is

$$F(t) = (1 + t)^{1/2}. \quad (10.1.31)$$

We look for an interpolant

$$s(x) = \sum_{j=1}^n \alpha_j \phi(\|x - x_j\|),$$

by solving the linear system

$$\sum_{j=1}^n \alpha_j \phi(\|x_i - x_j\|) = f(x_i), \quad i = 1, \dots, n.$$

The function given by (10.1.31) leads us to a multivariate interpolation process called interpolation by *multiquadratics*.

A variant of this process is the one proposed by R. Hardy [42], that uses as its basic functions

$$z_i(p) = [\|p - p_i\|^2 + c^2]^{1/2}, \quad i = 1, \dots, n.$$

Here the norm is Euclidean, and  $c$  is a parameter that Hardy suggested to be set equal to 0.8 times the average distance between nodes. The nonsingularity of coefficient matrix is due to Micchelli [65].

The MATLAB Source 10.5 computes the values of the interpolant at a given set of points for a given set of nodes, a given set of function values at nodes and a given basic function  $\phi$ . By default,  $\phi$  is the function given by (10.1.31).

**Example 10.1.16.** Plot the Gaussian and multiquadratics radial basis function interpolant for the function in Example 10.1.4,  $f : [-2, 2] \times [-2, 2] \rightarrow \mathbb{R}$ ,  $f(x, y) = xe^{-x^2-y^2}$ . The graphs are given in the left column of Figure 10.10. The right column plot the errors. The MATLAB sequence for the figure is

```

ftest=@(x,y) x.*exp(-x.^2-y.^2);
phi=@(r) exp(-2*r);
nX=[4*rand(100,1)-2;-2;-2;2;2];
nY=[4*rand(100,1)-2;-2;2;-2;2];
[X,Y]=meshgrid(linspace(-2,2,40));
nX=nX(:); nY=nY(:); f=ftest(nX,nY);
Z1=RBF(X(:,1),Y(:,1),nX,nY,f,phi);
Z2=RBF(X(:,1),Y(:,1),nX,nY,f);
ZE=ftest(X,Y); T=delaunay(X(:,1),Y(:,1));
G1=dela2(Z1); G2=dela2(Z2);
figure(1); trisurf(T,X,Y,Z1,G1)
figure(2); trisurf(T,X,Y,abs(Z1-ZE(:)),G1)
figure(3); trisurf(T,X,Y,Z2,G2)
figure(4); trisurf(T,X,Y,abs(Z2-ZE(:)),G2)

```

We used `delaunay` function and `trisurf` to obtain a better representation.  $\diamond$

For other examples, see [20].

Further references on multivariate interpolation are Kincaid and Cheney [50], Chui [12], Hartley [43], Micchelli [64], Franke [30], Cătinaş [20], and Lancaster and Salkauskas [54].

References on Shepard interpolation are Shepard [85], Gordon and Wixom [40], Newman and Rivlin [69], Barnhill, Dube and Little [3], Farwig [28], and Coman and Trîmbiţaş [15, 16].

---

**MATLAB Source 10.5** Interpolation with radial basis function

---

```

function Z=RBF(X,Y,nX,nY,f,phi)
%RBF - radial basis function interpolant
%call Z=RBF(X,Y,nX,nY,f,phi)
%X,Y - points for evaluation
%nX,nY - nodes
%f - function values at nodes
%phi - radial basis function

if nargin<6
    phi=@(x) (x+1).^(1/2);
end
D=sqdist(nX,nY); %compute square of distances
%find coefficients
A=phi(D);
a=A\f;
%compute interpolant values
n=length(nX);
Z=zeros(size(X));
for j=1:n
    Z=Z+a(j)*phi((X-nX(j)).^2+(Y-nY(j)).^2);
end
function M=sqdist(X,Y)
%compute squares of distances
[rX1,rX2]=meshgrid(X);
[rY1,rY2]=meshgrid(Y);
M=(rX1-rX2).^2+(rY1-rY2).^2;

```

---

## 10.2 Multivariate Numerical Integration

Let  $D \subseteq \mathbb{R}^n$ ,  $f : D \rightarrow \mathbb{R}$  an integrable function,  $P_i$   $i = \overline{0, m}$  points in  $D$ , and  $w$  a nonnegative weight function, defined on  $D$ . A formula of the form

$$\int \cdots \int_D w(x_1, \dots, x_n) f(x_1, \dots, x_n) dx_1 \dots dx_n = \sum_{i=0}^m A_i f(P_i) + R_m f \quad (10.2.1)$$

is called a *numerical integration formula* for  $f$  or a *cubature formula*. The parameters  $A_i$  are called *weights* or *coefficients* of the formula, the points  $P_i$  se its *nodes*, and  $R_m$  the *remainder term*.

An efficient method for the construction of cubature formulas when  $D$  is a rectangular domain, consists of expressing its coefficients and its nodes based on coefficients and nodes of a univariate formula respectively. We take into account only the bivariate case.

Let  $D = [a, b] \times [c, d]$  be a rectangle,  $\Delta_x$  the grid  $a = x_0 < x_1 < \dots < x_m = b$ , and  $\Delta_y$  the grid  $c = y_0 < y_1 < \dots < y_n = d$ ,  $w(x, y) = 1, \forall (x, y) \in D$ . The formula (10.2.1) becomes, in this case,

$$\int_a^b \int_c^d f(x, y) dx dy = \sum_{i=0}^m \sum_{j=0}^n A_{ij} f(x_i, y_j) + R_{m,n}(f). \quad (10.2.2)$$

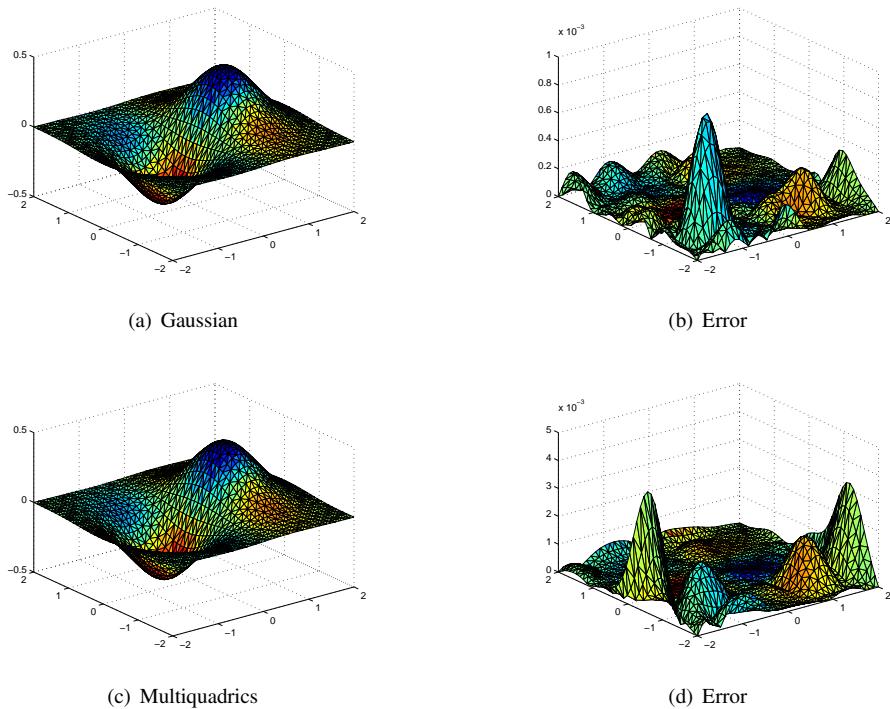


Figure 10.10: Graph of radial basis interpolants for the function in Example 10.1.16

We can generate such a formula starting from bivariate Lagrange interpolation formula

$$f = L_m^x L_n^y f + R_m^x \oplus R_n^y f.$$

If  $f \in C^{m+1, n+1}(D)$ , by integrating previous formula term by term, one obtains

$$\int_a^b \int_c^d f(x, y) dx dy = \sum_{i=0}^m \sum_{j=0}^n A_i B_j f(x_i, y_j) + R_{mn}(f) \quad (10.2.3)$$

where

$$A_i = \int_a^b \ell_i(x) dx, \quad B_j = \int_c^d \tilde{\ell}_j(y) dy,$$

and

$$\begin{aligned}
R_{m,n}(f) &= \int_a^b \int_c^d (R_m^x \oplus R_n^y) f(x, y) dx dy \\
&= \frac{1}{(m+1)!} \int_a^b \int_c^d u_m(x) f^{(m+1,0)}(\xi_x, y) dx dy \\
&\quad + \frac{1}{(n+1)!} \int_a^b \int_c^d u_n(y) f^{(0,n+1)}(x, \eta_y) dx dy \\
&\quad - \frac{1}{(m+1)!} \frac{1}{(n+1)!} \int_a^b \int_c^d u_m(x) u_n(y) f^{(m+1,n+1)}(\tilde{\xi}_x, \tilde{\eta}_y) dx dy.
\end{aligned}$$

If  $\Delta_x$  and  $\Delta_y$  are uniform grids of the intervals  $[a, b]$  and  $[c, d]$ , respectively, then cubature formula (10.2.3) is called a *Newton-Cotes* cubature formula.

**Particular cases.** For  $m = n = 1$  one obtains the trapezoidal cubature formula.

$$\begin{aligned}
\int_a^b \int_c^d f(x, y) dx dy &= \frac{(b-a)(d-c)}{4} [f(a, c) + f(a, d) + f(b, c) + f(b, d)] \\
&\quad + R_{11}(f),
\end{aligned}$$

where

$$\begin{aligned}
R_{11}(f) &= -\frac{(b-a)^3(d-c)}{12} f^{(2,0)}(\xi_1, \eta_1) - \frac{(b-a)(d-c)^3}{12} f^{(0,2)}(\xi_2, \eta_2) \\
&\quad - \frac{(b-a)^3(d-c)^3}{144} f^{(2,2)}(\xi_3, \eta_3).
\end{aligned}$$

For  $m = n = 2$  one obtains Simpson cubature formula.

$$\begin{aligned}
\int_a^b \int_c^d f(x, y) dx dy &= \frac{(b-a)(d-c)}{36} \left\{ f(a, c) + f(a, d) + f(b, c) + f(b, d) \right. \\
&\quad + 4 \left[ f\left(\frac{a+b}{2}, c\right) + f\left(\frac{a+b}{2}, d\right) + f\left(a, \frac{c+d}{2}\right) + f\left(b, \frac{c+d}{2}\right) \right] \\
&\quad \left. + 16f\left(\frac{a+b}{2}, \frac{b+c}{2}\right) \right\} + R_{22}(f),
\end{aligned}$$

where

$$\begin{aligned}
R_{22}(f) &= -\frac{(b-a)^5(d-c)}{2880} f^{(4,0)}(\xi_1, \eta_1) - \frac{(b-a)(d-c)^5}{2880} f^{(0,4)}(\xi_2, \eta_2) \\
&\quad - \frac{(b-a)^5(d-c)^5}{2880^2} f^{(4,4)}(\xi_3, \eta_3).
\end{aligned}$$

By partitioning the intervals  $[a, b]$  and  $[c, d]$  we can obtain iterated cubature formulae. We illustrate for Simpson's formula. Suppose  $[a, b]$  is divided into  $m$  equal length subintervals, and  $[c, d]$  into  $n$  equal length subintervals (we have a grid with  $mn$  rectangles). We will divide each rectangle into four smaller rectangles, as in Figure 10.11 (the vertices of rectangles are indicated by black circles, and intermediate points by lighter circles).

Let

$$h = \frac{b-a}{2m}, \quad k = \frac{d-c}{2n}.$$

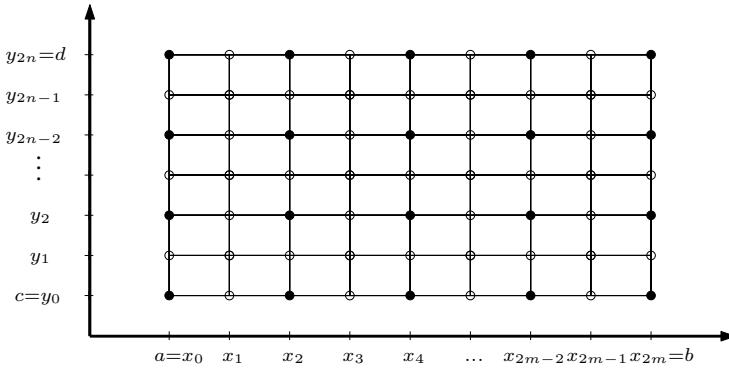


Figure 10.11: Subdivision of iterated Simpson formula

The coordinates of the subdivision points will be

$$\begin{aligned} x_i &= x_0 + ih, & x_0 &= a, & i &= \overline{0, 2m} \\ y_j &= y_0 + jk, & y_0 &= b, & j &= \overline{0, 2n}. \end{aligned}$$

We introduce the notation  $f_{ij} := f(x_i, y_j)$ . Applying elementary Simpson formula to each rectangle of the grid, we have

$$\begin{aligned} \int_a^b \int_c^d f(x, y) dx dy &= \frac{hk}{9} \sum_{i=0}^m \sum_{j=0}^n [f_{2i,2j} + f_{2i+2,j} + f_{2i+2,2j+2} \\ &\quad + f_{2i,2j+2} + 4(f_{2i+1,2j} + f_{2i+2,2j+1} + f_{2i+1,2j+2} + f_{2i,2j+1}) \\ &\quad + 16f_{2i+1,2j+1}] + R_{m,n}(f). \end{aligned}$$

After simplification, one obtains

$$\int_a^b \int_c^d f(x, y) dx dy = \frac{hk}{9} \sum_{i=0}^m \sum_{j=0}^n \lambda_{ij} f_{ij} + R_{m,n}(f),$$

where  $\lambda_{ij}$  are entries of the matrix

$$\Lambda = \begin{bmatrix} 1 & 4 & 2 & 4 & 2 & \dots & 4 & 2 & 4 & 1 \\ 4 & 16 & 8 & 16 & 8 & \dots & 16 & 8 & 16 & 4 \\ 2 & 8 & 4 & 8 & 4 & \dots & 8 & 4 & 8 & 2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 2 & 8 & 4 & 8 & 4 & \dots & 8 & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & 8 & \dots & 16 & 8 & 16 & 4 \\ 1 & 4 & 2 & 4 & 2 & \dots & 4 & 2 & 4 & 1 \end{bmatrix}.$$

[10, 88] give, when  $f \in C^{4,4}(D)$ , the following expression for the rest

$$R_{mn}(f) = -\frac{(b-a)(d-c)}{180} \left[ h^4 f^{(4,0)}(\xi_1, \eta_1) + k^4 f^{(0,4)}(\xi_2, \eta_2) \right],$$

for some  $\xi_1, \eta_1, \xi_2, \eta_2 \in D$ .

It is possible to construct bivariate Gauss formulae. For example, if  $x_i, i = \overline{0, m}$  and  $y_j, j = \overline{0, n}$  are the roots Legendre polynomial w.r.t. interval  $[a, b]$  and  $[c, d]$ , respectively, one obtains the Gauss-Legendre cubature formula, with coefficients

$$\begin{aligned} A_i &= \frac{[(m+1)!]^4(b-a)^{2m+3}}{[(2m+2)!]^2(x_i-a)(b-x_i)[u'(x_i)]^2}, & i &= \overline{0, m} \\ B_j &= \frac{[(n+1)!]^4(b-a)^{2n+3}}{[(2n+2)!]^2(y_j-c)(d-y_j)[u'(y_j)]^2}, & j &= \overline{0, n}, \end{aligned}$$

and, if  $f \in C^{2m+2, 2n+2}(D)$

$$\begin{aligned} R_{mn}(f) &= (d-c)\lambda_m f^{(2m+2,0)}(\xi_1, \eta_1) + (b-a)\tilde{\lambda}_n f^{(0,2n+2)}(\xi_2, \eta_2) \\ &\quad - \lambda_m \tilde{\lambda}_n f^{(2m+2, 2n+2)}(\xi_3, \eta_3), \end{aligned}$$

where

$$\lambda_m = \frac{[(m+1)!]^4(b-a)^{2m+3}}{[(2m+2)!]^3(2m+3)}, \quad \tilde{\lambda}_n = \frac{[(n+1)!]^4(b-a)^{2n+3}}{[(2n+2)!]^3(2n+3)}.$$

For further details, see [17].

If  $m = n = 0$ , one obtains an one-node cubature formula, analogous to rectangle formula

$$\int_a^b \int_c^d f(x, y) dx dy = (b-a)(d-c)f\left(\frac{a+b}{2}, \frac{c+d}{2}\right) + R_{00}(f),$$

where

$$\begin{aligned} R_{00}(f) &= \frac{(b-a)^3(d-c)}{24} f^{(2,0)}(\xi_1, \eta_1) + \frac{(b-a)(d-c)^3}{24} f^{(0,2)}(\xi_2, \eta_2) \\ &\quad - \frac{(b-a)^3(d-c)^3}{576} f^{(2,2)}(\xi_3, \eta_3). \end{aligned}$$

The utility of the previous methods is not limited to rectangular domains. For example, we can modify Simpson's cubature formula so that it can be applied to integrals of the form

$$\int_a^b \int_{c(x)}^{d(x)} f(x, y) dx dy \text{ or } \int_c^d \int_{a(y)}^{b(y)} f(x, y) dx dy,$$

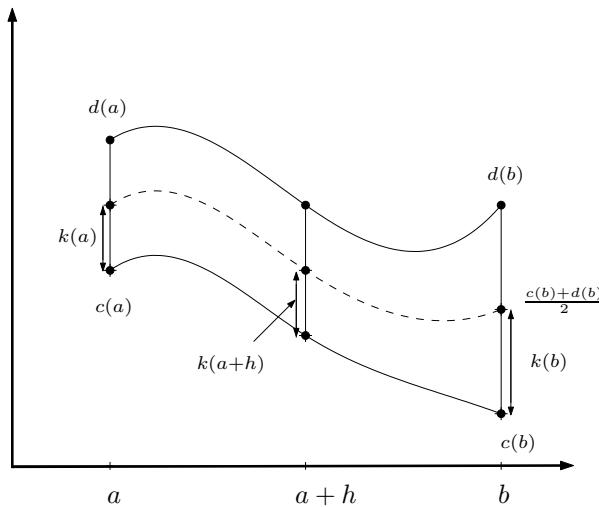
provided that, the domain is simple with respect to  $x$  or  $y$ , respectively.

For an integral of the first type the step for  $x$  will be  $h = \frac{b-a}{2}$ , but the step for  $y$  will vary as a function of  $x$  (see Figure 10.12):

$$k(x) = \frac{d(x) + c(x)}{2}.$$

One obtains

$$\begin{aligned} \int_a^b \int_{c(x)}^{d(x)} f(x, y) dx dy &\approx \int_a^b \frac{k(x)}{3} [f(x, c(x)) + 4f(x, c(x) + k(x)) + f(x, d(x))] dx \\ &\approx \frac{h}{3} \left\{ \frac{k(a)}{3} [f(a, c(a)) + 4f(a, c(a) + k(a)) + f(a, d(a))] \right. \\ &\quad + 4 \frac{k(a+h)}{3} [f(a+h, c(a+h)) + 4f(a+h, c(a+h) + k(a+h)) \\ &\quad + f(a+h, d(a+h))] \\ &\quad \left. + \frac{k(b)}{3} [f(b, c(b)) + 4f(b, c(b) + k(b)) + f(b, d(b))] \right\}. \end{aligned}$$

Figure 10.12: Simpson formula for a domain simple with respect to  $x$ 

**Implementation hints.** Consider the domain

$$D = [a, b] \times [c, d].$$

The integral to be approximated can be written as

$$\int_a^b \int_c^d f(x, y) dx dy = \int_a^b \left( \int_c^d f(x, y) dy \right) dx = \int_a^b F(x) dx,$$

where

$$F(x) = \int_c^d f(x, y) dy.$$

Suppose `adquad` is a univariate adaptive routine. Idea is to use this routine to compute values of  $F$  defined above and then to reuse the routine to integrate definite  $F$ . An implementation example is given in MATLAB source 10.6.

## 10.3 Multivariate Approximations in MATLAB

### 10.3.1 Multivariate interpolation in MATLAB

MATLAB has two functions for bivariate interpolation: `interp2` and `griddata`. The most general calling syntax of `interp2` is

```
ZI = interp2(x, y, z, XI, YI, method)
```

Here `x` and `y` contain the coordinates of interpolation nodes, `z` contains the values at nodes, and `XI` and `YI` are matrices containing the coordinates of points at which we wish to interpolate. `ZI` contains the values of interpolant at `XI`, `YI`. `method` can be one of the following values:

- '`'linear'` – Bilinear interpolation (default);
- '`'spline'` – Cubic spline interpolation;

**MATLAB Source 10.6 Double integral approximation on a rectangle**

---

```

function Q = quaddbl(F,xmin,xmax,ymin,ymax,tol, ...
    quadm,varargin)
%QUADDBL - approximates a double integral on a rectangle
%Parameters
%F - Integrand
%XMIN, XMAX, YMIN, YMAX - rectangle limits
%TOL -tolerance, default 1e-6
%QUADM - integration method, default adquad
if nargin < 5, error('Required at least 5 arguments'); end
if nargin < 6 || isempty(tol), tol = 1.e-6; end
if nargin < 7 || isempty(quadm), quadm = @adquad; end
F = fcncchk(F);

Q = quadm(@innerint, ymin, ymax, tol, [], F, ...
    xmin, xmax, tol, quadm, varargin{:});

%-----
function Q = innerint(y, F, xmin, xmax, tol, quadm, varargin)
%INNERINT - used by QUADDBL for inner integral.
%
% QUADM specifies quadrature to be used
% Evaluates inner integral for each value of outer variable

Q = zeros(size(y));
for i = 1:length(y)
    Q(i) = quadm(F, xmin, xmax, tol, [], y(i), varargin{:});
end

```

---

- 'nearest' – Nearest neighbor interpolation;
- 'cubic' – Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as 'spline'.

All interpolation methods require that  $X$  and  $Y$  be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. If you provide two monotonic vectors, `interp2` changes them to a plaid internally. Variable spacing is handled by mapping the given values in  $X$ ,  $Y$ ,  $XI$ , and  $YI$  to an equally spaced domain before interpolating. For faster interpolation when  $X$  and  $Y$  are equally spaced and monotonic, use the methods '`*linear`', '`*cubic`', '`*spline`', or '`*nearest`'.

Our example tries to interpolate MATLAB `peaks` function on a 7-by-7 grid. We generate the grid, compute the function values and plot the function with MATLAB sequence

```
[X,Y]=meshgrid(-3:1:3);
Z=peaks(X,Y);
surf(X,Y,Z)
```

The graph is given in Figure 10.13. Then we compute the interpolants on a finer grid and plot them:

```
[XI,YI]=meshgrid(-3:0.25:3);
```

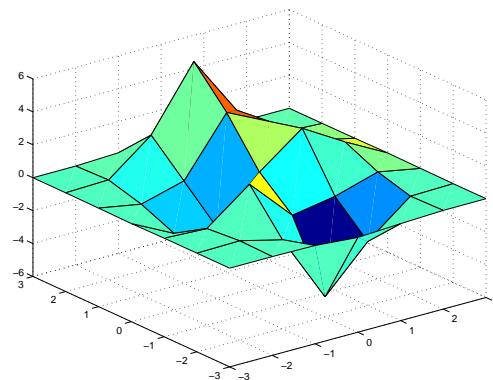


Figure 10.13: Graph of peaks on a coarse grid

```

ZI1=interp2(X,Y,Z,XI,YI,'nearest');
ZI2=interp2(X,Y,Z,XI,YI,'linear');
ZI3=interp2(X,Y,Z,XI,YI,'cubic');
ZI4=interp2(X,Y,Z,XI,YI,'spline');
subplot(2,2,1), surf(XI,YI,ZI1)
title('nearest')
subplot(2,2,2), surf(XI,YI,ZI2)
title('linear')
subplot(2,2,3), surf(XI,YI,ZI3)
title('cubic')
subplot(2,2,4), surf(XI,YI,ZI4)
title('spline')

```

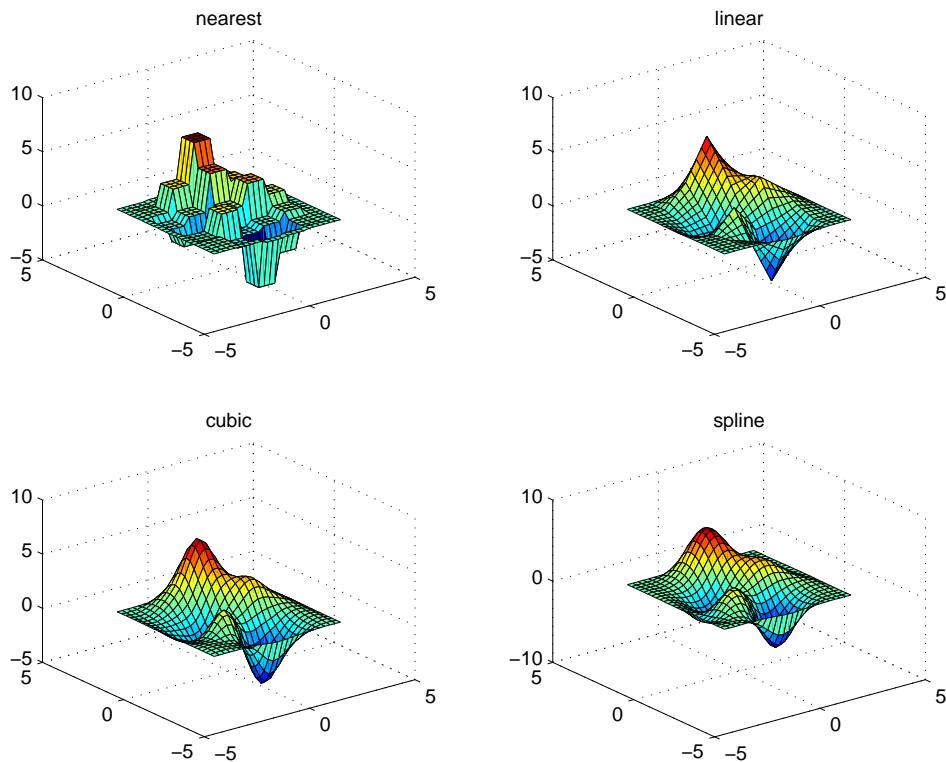
See Figure 10.14 for their graphs. If we replace everywhere `surf` by `contour` we obtain the graphs in Figure 10.15.

The `griddata` function has the same syntax as `interp2`. The input data are nodes coordinates `x` and `y`, which need not to be monotone, and the values at nodes, `z`. The function computes the values `ZI` of interpolant at nodes `XI` and `YI`. The nodes are generated via `meshgrid`. The `method` parameter may be '`linear`', '`cubic`', `nearest` and '`v4`', the latter being a method peculiar to MATLAB 4. All methods, excepting `v4` are based on Delaunay triangulation (a triangulation of a set that minimizes the maximum angle). The method is useful to interpolate values on a surface. The next example interpolates random points on the surface  $z = \frac{\sin(x^2+y^2)}{x^2+y^2}$  ("Mexican hat"). To avoid problems at origin we add `eps` to denominator.

```

x=rand(100,1)*16-8; y=rand(100,1)*16-8;
R=sqrt(x.^2+y.^2)+eps;
z=sin(R). ./ R;
xp=-8:0.5:8;
[XI,YI]=meshgrid(xp,xp);
ZI=griddata(x,y,z,XI,YI);
mesh(XI,YI,ZI); hold on
plot3(x,y,z,'ko'); hold off

```

Figure 10.14: `interp2` example

See Figure 10.16 for the result. The random points are represented as circles, and the interpolant is plotted with `mesh`.

### 10.3.2 Computing double integrals in MATLAB

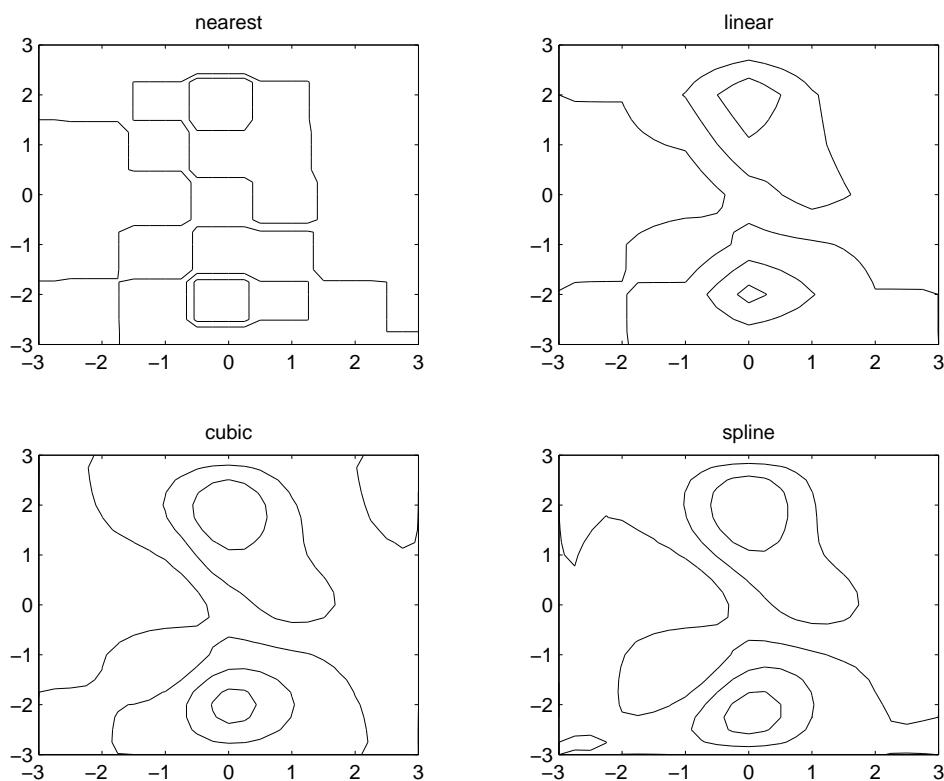
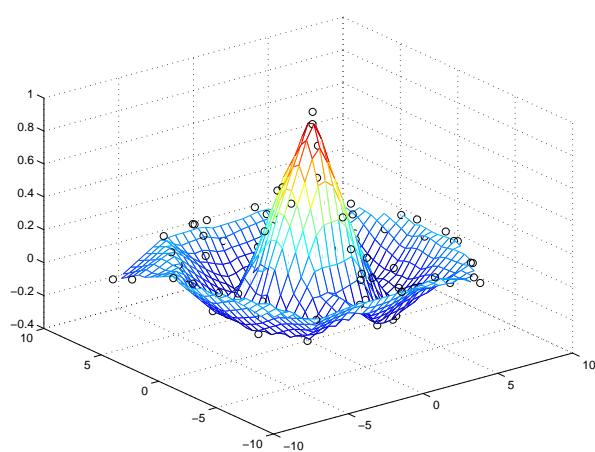
We can approximate double integrals on rectangles using `dblquad`. To illustrate, we shall approximate the integral

$$\int_0^\pi \int_\pi^{2\pi} (y \sin x + x \cos y) dx dy$$

Its exact value is  $-\pi^2$ , as it can be checked using Symbolic Math toolbox:

```
>> syms x y
>> Pi=sym(pi);
>> z=y*sin(x)+x*cos(y);
>> int(int(z,x,Pi,2*Pi),y,0,Pi)
ans =
```

$$-\frac{\pi^2}{2}$$

Figure 10.15: Contours generated by `interp2`Figure 10.16: `griddata` interpolation

We can define the integrand as an inline object, M-file, character string, anonymous function or function handle. Suppose we gave it in the M-file `integrand.m`:

```
function z = integrand(x, y)
z = y*sin(x)+x*cos(y);
```

We shall use `dblquad` and check it:

```
>> Q = dblquad(@integrand, pi, 2*pi, 0, pi)
Q =
-9.8696
>> -pi^2
ans =
-9.8696
```

Also, we compute the integral with `quaddbl` (MATLAB source 10.6):

```
>> Q2=quaddbl(@integrand,pi,2*pi,0,pi)
Q2 =
-9.8696
```

The integrand for `dblquad` and `quaddbl` must accept a vector `x` and a scalar `y` and return a vector of values of the integrand. We can pass additional arguments to specify the accuracy (tolerance) and the univariate integration method (default `quad` for `dblquad`). Suppose we want to compute

$$\int_4^6 \int_0^1 (y^2 e^x + x \cos y) dx dy$$

with a tolerance of `1e-8` and to use `quadl` instead of `quad`:

```
>> fi = @(x,y) y.^2.*exp(x)+x.*cos(y);
>> dblquad(fi,0,1,4,6,1e-8,@quadl)
ans =
87.2983
```

The exact value, provided by Maple or Symbolic Math Toolbox is

$$\frac{152}{3}(e - 1) + \frac{1}{2}(\sin 6 - \sin 4).$$

Let us check this:

```
>> syms x y
>> z=y^2*exp(x)+x*cos(y);
>> int(int(z,x,0,1),y,4,6)
ans =
- 152/3 + 152/3 exp(1) - 1/2 sin(4) + 1/2 sin(6)
>> double(ans)
ans =
87.2983
```

## Problems

**Problem 10.1.** Code a MATLAB function that plots a surface  $f(x, y)$ ,  $f : [0, 1] \times [0, 1] \rightarrow \mathbb{R}$  which fulfills the conditions

$$\begin{aligned} f(0, y) &= g_1(y) & f(1, y) &= g_2(y) \\ f(x, 0) &= g_3(x) & f(x, 1) &= g_4(y), \end{aligned}$$

where  $g_i$ ,  $i = \overline{1, 4}$  are functions defined on  $[0, 1]$ .

**Problem 10.2.** Find the bivariate tensor product and boolean sum corresponding to a univariate Hermite interpolant with double nodes 0 and 1. Plot this interpolant for the function  $f(x, y) = x \exp(x^2 + y^2)$ .

**Problem 10.3.** Adapt `quaddb1` function to approximate double integrals of the form

$$\int_a^b \int_{c(x)}^{d(x)} f(x, y) \, dy \, dx$$

or

$$\int_c^d \int_{a(y)}^{b(y)} f(x, y) \, dx \, dy,$$

when the integration domain is simple with respect to  $x$  or  $y$ .

**Problem 10.4.** Consider the double integral of the function  $f(x, y) = x^2 + y^2$  on the elliptical domain  $R$  given by  $-5 < x < 5$ ,  $y^2 < \frac{3}{5}(25 - x^2)$ .

- (a) Plot the function on  $R$ .
- (b) Find the exact value of the integral using Maple or Symbolic Math Toolbox.
- (c) Approximate the value of the integral by transforming the ellipse into a rectangle.
- (d) Approximate the integral using functions in Problem 10.3.

**Problem 10.5.** Consider the function  $f(x, y) = y \cos x^2$  and the triangular domain  $T = \{x \geq 0, y \geq 0, x + y \leq 1\}$  and

$$I = \int \int_T f(x, y) \, dx \, dy.$$

- (a) Plot the function on  $T$  using `trimesh` on `trisurf`.
- (b) Approximate the value of  $I$  by transforming the integral into an integral of a function defined on the unit square, that is null outside of  $T$ .
- (c) Approximate the integral using functions in Problem 10.3.



## Bibliography

- [1] Octavian Agratini, Ioana Chiorean, Gheorghe Coman, and Radu Trîmbițaș, *Numerical Analysis and Approximation Theory*, vol. III, Cluj University Press, 2002, D. D. Stancu, Gh. Coman (coords), (in Romanian).
- [2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Sorensen D., *LAPACK Users' Guide*, third ed., SIAM, Philadelphia, 1999, <http://www.netlib.org/lapack>.
- [3] R. Barnhill, R. P. Dube, and F. F. Little, *Properties of Shepard's surfaces*, Rocky Mtn. J. Math. **13** (1983), 365–382.
- [4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed., SIAM, Philadelphia, PA, 1994, available via [www.netlib.org/templates](http://www.netlib.org/templates).
- [5] Jean-Paul Berrut and Lloyd N. Trefethen, *Barycentric lagrange interpolation*, SIAM Review **46** (2004), no. 3, 501–517.
- [6] Å. Björk, *Numerical Methods for Least Squares Problem*, SIAM, Philadelphia, 1996.
- [7] E. Blum, *Numerical Computing: Theory and Practice*, Addison-Wesley, 1972.
- [8] P. Bogacki and L. F. Shampine, A 3(2) pair of Runge-Kutta formulas, Appl. Math. Lett. **2** (1989), no. 4, 321–325.
- [9] C. G. Broyden, *A Class of Methods for Solving Nonlinear Simultaneous Equations*, Math. Comp. **19** (1965), 577–593.
- [10] L. Burden and J. D. Faires, *Numerical Analysis*, PWS Kent, Boston, 1986.
- [11] W. Cheney and W. Light, *A Course in Approximation Theory*, Brooks/Cole, Pacific Grove, 2000.
- [12] C. K. Chui, *Multivariate splines*, SIAM Regional Conference Series in Mathematics, 1988.
- [13] K. C. Chung and T. H. Yao, *On lattices admitting unique Lagrange interpolation*, SIAMNA **14** (1977), 735–743.
- [14] P. G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimisation*, Masson, Paris, Milan, Barcelone, Mexico, 1990.
- [15] Gh. Coman and R. T. Trîmbițaș, *Bivariate shepard interpolation*, Seminar on Numerical and Statistical Calculus, preprint (1999), no. 1, 41–83.
- [16] \_\_\_\_\_, *Bivariate Shepard interpolation in MATLAB*, Seminarul itinerant "Tiberiu Popoviciu" de Ecuații funcționale, aproximare și convexitate (Cluj-Napoca, Romania), 2000, pp. 41–56.
- [17] Gheorghe Coman, *Numerical Analysis*, Libris, Cluj-Napoca, 1995, (in Romanian).
- [18] C. Cormen, T. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1994.
- [19] M. Crouzeix and A. L. Mignot, *Analyse numérique des équations différentielles*, Masson, Paris, Milan, Barcelone, Mexico, 1989.
- [20] Teodora Cătinaș, *Interpolation of Scattered Data*, Casa Cărții de Știință, 2007.
- [21] I. Cuculescu, *Numerical Analysis*, Editura Tehnică, București, 1967, (in Romanian).
- [22] P. J. Davis and P. Rabinowitz, *Numerical Integration*, Blaisdell, Waltham, Massachusetts, 1967.
- [23] James Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

- [24] J. E. Dennis and J. J. Moré, *Quasi-Newton Methods, Motivation and Theory*, SIAM Review **19** (1977), 46–89.
- [25] J. Dormand, *Numerical Methods for Differential Equations. A Computational Approach*, CRC Press, Boca Raton New York, 1996.
- [26] T. A. Driscoll, N. Hale, and L. N. Trefethen, *Chebfun guide*, Pafnuty Publications, Oxford, 2014.
- [27] Tobin A. Driscoll, *Crash course in MATLAB*, www, 2006, [math.udel.edu/~driscoll/MATLABCrash.pdf](http://math.udel.edu/~driscoll/MATLABCrash.pdf).
- [28] R. Farwig, *Rate of convergence of Shepard's global interpolation formula*, Math. of Comp. **46** (1986), 577–590.
- [29] J. G. F. Francis, *The QR transformation: A unitary analogue to the LR transformation*, Computer J. **4** (1961), 256–272, 332–345, parts I and II.
- [30] R. Franke, *Scattered data interpolation*, Math. of Comp. **38** (1982), 181–200.
- [31] W. Gander and W. Gautschi, *Adaptive quadrature - revisited*, BIT **40** (2000), 84–101.
- [32] W. Gautschi, *On the condition of algebraic equations*, Numer. Math. **21** (1973), 405–424.
- [33] ———, *Numerical Analysis, an Introduction*, Birkhäuser, Basel, 1997.
- [34] Walther Gautschi, *Orthogonal polynomials: applications and computation*, Acta Numerica **5** (1996), 45–119.
- [35] J. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation.*, SIAM J. Matrix Anal. Appl. **13** (1992), no. 1, 333–356, available in MATLAB kit.
- [36] G. Glaeser and H. Stachel, *Open Geometry: OpenGL® + Advanced Geometry*, Springer, 1999.
- [37] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, Computing Surveys **23** (1991), no. 1, 5–48.
- [38] H. H. Goldstine and J. von Neumann, *Numerical inverting of matrices of high order*, Amer. Math. Soc. Bull. **53** (1947), 1021–1099.
- [39] Gene H. Golub and Charles van Loan, *Matrix Computations*, 3rd ed., John Hopkins University Press, Baltimore and London, 1996.
- [40] W. J. Gordon and J. A. Wixom, *Shepard's method of 'metric interpolation' to bivariate and multivariate interpolation*, Math. Comp. **32** (1978), 253–264.
- [41] P. R. Halmos, *Finite-Dimensional Vector Spaces*, Springer Verlag, New York, 1958.
- [42] R. L. Hardy, *Multiquadric equations of topography and other irregular surfaces*, Journal Geophysical Research **76** (1971), 1905–1915.
- [43] P. H. Hartley, *Tensor product approximations to data defines on rectangle meshes in n-spce*, Computer Jounal **19** (1976), 348–352.
- [44] D. J. Higham and N. J. Higham, *MATLAB Guide*, second ed., SIAM, Philadelphia, 2005.
- [45] N. J. Higham and F. Tisseur, *A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra*, SIAM Journal Matrix Anal. Appl. **21** (2000), no. 4, 1185–1201.
- [46] Nicholas J. Higham, *The Test Matrix Toolbox for MATLAB*, Tech. report, Manchester Centre for Computational Mathematics, 1995, available via WWW, address <http://www.ma.man.ac.uk/MCCM/MCCM.html>.
- [47] Nicholas J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.
- [48] E. Isaacson and H. B. Keller, *Analysis of Numerical Methods*, John Wiley, New York, 1966.

- [49] C. G. J. Jacobi, *Über eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden linearen Gleichungen*, Astronomische Nachrichten **22** (1845), 9–12, Issue no. 523.
- [50] D. Kincaid and W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*, Brooks/Cole Publishing Company, Belmont, CA, 1991.
- [51] Mirela Kohr, *Special Chapter of Mechanics*, Cluj University Press, 2005, in Romanian.
- [52] Mirela Kohr and Ioan Pop, *Viscous Incompressible Flow for Low Reynolds Numbers*, WIT Press, Southampton(UK) - Boston, 2004.
- [53] V. N. Kublanovskaya, *On some algorithms for the solution of the complete eigenvalue problem*, USSR Comp. Math. Phys. **3** (1961), 637–657.
- [54] P. Lancaster and K. Salkauskas, *Curve and Surface Fitting*, Academic Press, New York, 1986.
- [55] P. Marchand and O. T. Holland, *Graphics and GUIs with MATLAB*, third ed., CHAPMAN & HALL/CRC, Boca Raton, London, New York, Washington, D.C., 2003.
- [56] The Mathworks Inc., Natick, Ma, *Using MATLAB*, 2002.
- [57] The Mathworks Inc., *Learning MATLAB 7*, 2005, Version 7.
- [58] The Mathworks Inc., *MATLAB 7. Getting Started Guide*, 2008, Minor revision for MATLAB 7.7 (Release 2008b).
- [59] The Mathworks Inc., *MATLAB 7 Graphics*, 2008, Revised for MATLAB 7.7 (Release 2008b).
- [60] The Mathworks Inc., *MATLAB 7. Mathematics*, 2008, Revised for MATLAB 7.7 (Release 2008b).
- [61] The Mathworks Inc., *MATLAB 7. Programming Fundamentals*, 2008, Revised for Version 7.7 (Release 2008b).
- [62] The Mathworks Inc., Natick, Ma, *MATLAB. Symbolic Math Toolbox 5*, 2008, Revised for Version 5.1 (Release 2008b).
- [63] J. Meier, *Parametrische Flächen*, 2000, available via www, address <http://www.3d-meier.de/tut3/Seite0.html>.
- [64] C. A. Micchelli, *Algebraic aspects of interpolation*, Approximation Theory (Providence R.I.) (C. deBoor, ed.), Proceedings of Symposia in Applied Mathematics, vol. 36, AMS, 1986, pp. 81–102.
- [65] ———, *Interpolation of scattered data: Distance matrices and conditionally positive definite functions*, Constructive Approximation **2** (1986), 11–22.
- [66] Cleve Moler, *Numerical Computing in MATLAB3*, SIAM, 2004, available via www at <http://www.mathworks.com/moler>.
- [67] J. J. Moré and M. Y. Cosnard, *Numerical Solutions of Nonlinear Equations*, ACM Trans. Math. Softw. **5** (1979), 64–85.
- [68] Shoichiro Nakamura, *Numerical Computing and Graphic Visualization in MATLAB*, Prentice Hall, Englewood Cliffs, NJ, 1996.
- [69] D. J. Newman and T. J. Rivlin, *Optimal universally stable interpolation*, Analysis **3** (1983), 355–367.
- [70] Dana Petcu, *Computer Assisted Mathematics*, Eubeea, Timișoara, 2000, (in Romanian).
- [71] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, Cambridge, New York, Port Chester, Melbourne, Sidney, 1996, available via www, <http://www.nr.com/>.

- [72] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri, *Numerical Mathematics*, Springer, New York, Berlin, Heidelberg, 2000.
- [73] I. A. Rus, *Differential Equations, Integral Equations and Dynamical Systems*, Transilvania Press, Cluj-Napoca, 1996, (in Romanian).
- [74] I. A. Rus and P. Pavel, *Differential Equations*, 2nd ed., Editura Didactică și Pedagogică, București, 1982, (in Romanian).
- [75] H. Rutishauser, *Solution of the eigenvalue problems with the LR transformation*, Nat. Bur. Stand. App. Math. Ser. **49** (1958), 47–81.
- [76] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing, Boston, 1996, available via www, <http://www-users.cs.umn.edu/~saad/books.html>.
- [77] H. E. Salzer, *Lagrangian interpolation at the chebyshev points  $\cos(\nu\pi/n)$ ,  $\nu = 0(1)n$ ; some unnoted advantages*, Computer Journal **15** (1974), 156–159.
- [78] A. Sard, *Linear Approximation*, American Mathematical Society, Providence, RI, 1963.
- [79] Thomas Sauer, *Numerische Mathematik I*, Universität Erlangen-Nürnberg, Erlangen, 2000, Vorlesungskript.
- [80] ———, *Numerische Mathematik II*, Universität Erlangen-Nürnberg, Erlangen, 2000, Vorlesungskript.
- [81] R. Schwarz, H., *Numerische Mathematik*, B. G. Teubner, Stuttgart, 1988.
- [82] L. F. Shampine, *Vectorized adaptive quadrature in MATLAB*, Journal of Computational and Applied Mathematics **211** (2008), 131–140.
- [83] L. F. Shampine, R. C. Allen, and S. Pruess, *Fundamentals of Numerical Computing*, John Wiley & Sons, Inc, 1997.
- [84] L. F. Shampine, I. Gladwell, and S Thompson, *Solving ODEs with MATLAB*, Cambridge University Press, Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, 2003.
- [85] D. Shepard, *A two-dimensional interpolation function for irregularly spaced data*, Proceedings 23rd National Conference ACM, 517–524.
- [86] D. D. Stancu, *On Hermite's interpolation formula and some of its applications*, Acad. R. P. Rom. Studii și Cercetări Matematice **8** (1957), 339–355, (in Romanian).
- [87] D. D. Stancu, *Numerical Analysis - Lecture Notes and Problem Book*, Lito UBB, Cluj-Napoca, 1977, (in Romanian).
- [88] D. D. Stancu, G. Coman, and P. Blaga, *Numerical Analysis and Approximation Theory*, vol. II, Cluj University Press, Cluj-Napoca, 2002, D. D. Stancu, Gh. Coman, (coord.) (in Romanian).
- [89] D. D. Stancu, Gh. Coman, O. Agratini, and R. Trîmbițaș, *Numerical Analysis and Approximation Theory*, vol. I, Cluj University Press, Cluj-Napoca, 2001, D. D. Stancu, Gh. Coman, (coord.) (in Romanian).
- [90] J. Stoer and R. Burlisch, *Einführung in die Numerische Mathematik*, vol. II, Springer Verlag, Berlin, Heidelberg, 1978.
- [91] ———, *Introduction to Numerical Analysis*, 2nd ed., Springer Verlag, 1992.
- [92] Volker Strassen, *Gaussian elimination is not optimal*, Numer. Math. **13** (1969), 354–356.
- [93] A. H. Stroud, *Approximate Calculation of Multiple Integrals*, Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

- [94] L. N. Trefethen, *Maxims About Numerical Mathematics, Computers, Science and Life*, SIAM News **31** (1998), no. 1, 1.
- [95] Lloyd N. Trefethen, *The Definition of Numerical Analysis*, SIAM News (1992), no. 3, 1–5.
- [96] Lloyd N. Trefethen and David Bau III, *Numerical Linear Algebra*, SIAM, Philadelphia, 1996.
- [97] R. T. Trîmbițaș, *Local bivariate Shepard interpolation*, Rendiconti del Circolo matematico di Palermo **68** (2002), 701–710, Serie II, Suppl.
- [98] ———, *Numerical Analysis. An Introduction Based on MATLAB*, Cluj University Press, Cluj-Napoca, 2005, (in Romanian).
- [99] E. E. Tyrtyshnikov, *A Brief Introduction to Numerical Analysis*, Birkhäuser, Boston, Basel, Berlin, 1997.
- [100] C. Überhuber, *Computer-Numerik*, vol. 1, 2, Springer Verlag, Berlin, Heidelberg, New-York, 1995.
- [101] C. Ueberhuber, *Numerical Computation. Methods, Software and Analysis*, vol. I, II, Springer Verlag, Berlin, Heidelberg, New York, 1997.
- [102] R. E. White, *Computational Mathematics. Models, Methods, and Analysis with MATLAB and MPI*, Chapman & Hall/CRC, Boca Raton, London, New York, Washington, D.C., 2004.
- [103] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.
- [104] J. H. Wilkinson, *The perfidious polynomial*, Studies in Numerical Analysis (Gene H. Golub, ed.), MAA Stud. Math., vol. 24, Math. Assoc. America, Washington, DC, 1984, pp. 1–28.
- [105] H. B. Wilson, L. H. Turcotte, and D. Halpern, *Advanced Mathematics and Mechanics Applications Using MATLAB*, third ed., Chapman & Hall/CRC, Boca Raton, London, New York, Washington, D.C., 2003.

---

## Index

---

*p*-norm, 101  
\ (operator), 13, 126  
: (operator), 11  
% (comment), 26  
adaptive quadratures, 251  
anonymous function, 30  
asymptotic error, 270  
axis, 57  
blkdiag, 10  
Boolean sum, 415  
box, 54  
break, 24  
Butcher table, 353, 370  
camlight, 71  
Cartesian grid, 413  
cell, 35  
cell array, 35  
chol, 130  
Cholesky factorization, 120  
class, 32  
composite Simpson formula, 238  
composite trapezoidal rule, 238  
cond, 109  
condest, 109  
condition number, 86  
continue, 25  
contour, 65  
conv, 179  
convergence  
    linear, 270  
    order of  $\sim$ , 270  
sublinear, 270  
superlinear, 270  
cubature formula, 437  
    Newton-Cotes  $\sim$ , 438  
    Simpson  $\sim$ , 439  
    trapezoidal  $\sim$ , 438  
cumprod, 16  
cumsum, 16  
dblquad, 445  
deal, 37  
decic, 397  
deconv, 179  
degree of exactness, 232, 237  
delaunay, 436  
det, 15  
deval, 396  
diag, 10  
diary, 6  
diff(Symbolic), 41  
digits, 46  
disp, 5  
divided difference, 202  
double, 32  
double, 47  
dsolve, 46  
efficiency index, 271  
eig, 326  
eigenvalue  
    condition number, 329  
eps, *see* machine epsilon  
eps, 4  
eps, 81, 84

error, 40  
eval, 34  
eyes, 8  
  
fcnchk, 31  
fill, 56  
find, 19  
fliplr, 10  
flipud, 10  
fminbnd, 299  
fminsearch, 298  
for, 23  
format, 4  
formula  
    Euler-MacLaurin ~, 257  
    Simpson ~, 238  
fplot, 54  
fsolve, 45  
full, 21  
fzero, 296  
  
gallery, 11  
Gauss-Christoffel quadrature formula, *see* Gaussian quadrature formula  
Gaussian quadrature formula, 244  
generalized eigenvalues, 330  
global, 39  
grid, 355  
grid, 57  
grid function, 355  
griddata, 442, 444  
gsvd, 332  
  
handle graphics, 68  
hess, 330  
  
if, 23  
Inf, 4  
inline, 30  
int, 32  
int, 41  
interp1, 219  
interp2, 442  
interpolation  
    cardinal property, 414  
inv, 15  
  
Lagrange interpolation  
    Aitken method, 201  
    Neville method, 200  
  
lasterr, 40  
lasterror, 40  
Lebesgue  
    constant, 196  
    function, 196  
left eigenvector, 329  
length, 7  
light, 71  
limit, 43  
linsolve, 133  
linspace, 8  
load, 6  
log2, 83  
logical, 20  
logspace, 8  
lsqnonneg, 128  
lu, 129  
  
M file, 25  
M-file, 25  
    function, 25, 27  
    script, 25  
machine epsilon, 77  
maple, 46  
matrices  
    similar, 307  
matrix  
    characteristic polynomial of a ~, 305  
    companion, 306  
    condition number of a ~, 181  
    condition number of a ~, 108  
    diagonalisable ~, 307  
    eigenvalue of a ~, 305  
    eigenvector of a ~, 305  
    hermitian, 102  
    Jordan normal form of a ~, 307  
    nonderogatory ~, 307  
    normal, 102  
    orthogonal, 102  
    real Schur decomposition of a ~, 309  
    RQ transformation of a ~, 317  
    Schur decomposition of a ~, 307  
    singular value decomposition of a ~, 331  
    symmetric, 102  
    unitary, 102  
    upper Hessenberg, 102, 309  
matrix norm, 102  
    subordinate, 103  
max, 16

- maximal pivoting, *see* total pivoting  
 mean, 16  
 median, 16  
 mesh, 63  
 meshgrid, 63  
 method  
     Broyden's ~, 293  
     Euler ~, 345  
     false position ~, 274  
     fixed point iteration ~, 285  
     Heun ~, 348, 351  
     modified Euler ~, 348, 351  
     Newton ~, 279  
     power ~, *see* vector iteration  
     QR  
         double shift, 326  
         simple, 319  
         spectral shift, 322  
     QR ~, 312  
     quasi-Newton~, 291  
     Romberg ~, 252  
     Runge-Kutta ~, 349  
     secant ~, 277  
     semi-implicit Runge-Kutta ~, 350  
     SOR, 139  
     Sturm ~, 272  
     Taylor expansion ~, 346  
 min, 16  
 moving least squares, 432  
 multiquadric, 435  
  
 NaN, 4  
 nargin, 28  
 nargout, 28  
 ndims, 7  
 nested function, 28  
 Newton-Cotes formulae, 243  
 nnz, 22  
 norm  
     Chebyshev, 101  
     Euclidian, 101  
     Frobenius, 106  
     Minkowski, 101  
 norm, 101, 107  
 normest, 107  
 notation  
      $\Omega$ , 90  
 null, 129  
 num2str, 34  
  
 numerical differentiation formula, 232  
 numerical integration formula, 237  
 numerical quadrature formula, *see* numerical integration formula  
 numerical solution of differential equations  
     one-step methods, 343  
  
 ode23tb, 377  
 ode15i, 397  
 ode113, 377  
 ode15s, 377  
 ode23s, 377  
 ode23tb, 377  
 ode23t, 377  
 ode23, 377  
 ode45, 377  
 odeset, 390  
 odextend, 396  
 one step method  
     stable ~, 356  
 one-step method  
     consistent ~, 344  
     convergent ~, 359  
     exact order, 344  
     order, 344  
     principal error function, 344  
 ones, 8  
 optimset, 296  
  
 pchip, 221, 223  
 pi, 5  
 pinv, 127, 181  
 plot, 51  
 plot3, 61  
 polar, 55  
 poly, 178  
 polyder, 178  
 polyfit, 182, 219  
 polyval, 178  
 polyvalm, 178  
 pow2, 83  
 ppval, 222  
 print, 69  
 prod, 16  
  
 qr, 131  
 quad, 258  
 quadgk, 261  
 quadl, 258

quadv, 261  
radial basis functions, 434  
rand, 8  
rand, 8  
randn, 8  
randn, 8  
rcond, 109  
realmax, 81  
realmin, 81  
repmat, 8, 36  
reshape, 10  
roots, 178  
rot90, 10  
Runge-Kutta method  
    implicit ~, 349  
  
save, 6  
schur, 330  
shading, 66  
Shepard interpolation, 425  
simple, 42  
simplify, 42  
single, 32  
single, 83  
size, 7  
solve, 44  
sort, 16  
sparse, 21  
spdiags, 22  
spline  
    complete, 215  
    Not-a-knot, 215  
spline, 221  
sprintf, 34  
spy, 22  
stability inequality, 356  
std, 16  
str2num, 34  
struct, 36  
structure, 36  
subfunction, 28  
subplot, 60  
subs, 42  
sum, 16  
surf, 64  
svd, 331  
switch, 23  
sym, 41  
syms, 41  
taylor, 43  
tensor product, 414, 415, 417  
text, 58  
theorem  
    Peano, 186  
tic, 39  
title, 54  
toc, 39  
total pivoting, 114  
transform  
    Householder, 122  
trapezes rule, *see* composite trapezoidal rule  
trapezoidal formula, *see* trapezoidal rule  
trapezoidal rule, 237  
trapz, 261  
tril, 10  
trisurf, 436  
triu, 10  
truncation error, 344  
try-catch, 40  
  
uint\*, 32  
  
var, 16  
varargin, 36  
varargin, 37  
varargout, 36  
varargout, 37  
vector iteration, 310  
vectorize, 31  
view, 64  
vpa, 46  
  
warning, 40  
while, 23, 24  
whos, 5  
  
xlabel, 54  
xlim, 57  
  
ylabel, 54  
ylim, 57  
  
zeros, 8  
zlim, 63

