Contents

Chapter 0. Introduction to these Lecture Notes 0-1

0.1. About These Notes 0-1

0.2. Mathematical Background 0-1

0.3. Computer Background 0-2

Chapter 1. Introduction 1-1

1.1. Why a New Book? 1-1

1.2. The Subroutines 1-1

1.2.1. What is available? 1-1

1.2.2. Where do I find them? 1-2

1.3. Mathematical Software: An Example 1-2

1.4. Design of Scientific Subroutines 1-2

Chapter 2. Computer Arithmetic and Computational Errors 2-1

2.1. Introduction 2-1

2.2. Representation of Numbers 2-1

2.2.1. Integers 2-1

2.2.2. Reals 2-2

2.2.3. Double Precision 2-2

2.2.4. Rounding 2-3

2.3. Machine Constants 2-3

2.4. Errors in Scientific Computing 2-3

2.4.1. The Condition Number 2-4

2.4.2. Sources of Errors 2-4

2.4.3. Arithmetic Errors 2-5

2.5. Extrapolation 2-5

2.6. Historical Perspective 2-7

2.7. Examples 2-7

2.7.1. Avoiding Ill-conditioned Algorithms 2-8

2.7.2. Interaction of Errors 2-8

Chapter 3. Linear Systems of Equations 3-1
3.1. Introduction 3-1
  3.1.1. Vectors 3-1
  3.1.2. Matrices 3-3
  3.1.3. Matrix Equations 3-7

3.2. Gaussian Elimination 3-8
  3.2.1. The Basic Algorithm 3-8
  3.2.2. Programming Technique 3-9
  3.2.3. Pivoting 3-10
  3.2.4. Singular Matrices 3-11
  3.2.5. The Complete Algorithm 3-11
  3.2.6. Operation Count 3-11
  3.2.7. LU Decomposition 3-12
  3.2.8. Calculating the Determinant 3-12
  3.2.9. Calculating $A^{-1}$ 3-13

3.3. Subroutine SGEFS 3-13

3.4. Historical Perspective 3-13

3.5. Column-Oriented Algorithms 3-13

3.6. The Condition Number 3-14

Chapter 4. Interpolation 4-1

4.1. Introduction 4-1
  4.1.1. Solving Linear Interpolation Problems 4-2

4.2. Polynomial Interpolation 4-3

4.3. Using Other Basis Functions 4-3
  4.3.1. The Matrix Approach 4-3
  4.3.2. The Basis Function Approach 4-4
  4.3.3. Newton’s Approach 4-4
  4.3.4. Comparison of Methods 4-6

4.4. How Good is Polynomial Interpolation? 4-6
  4.4.1. Fixed Interpolation Points 4-6
  4.4.2. Increasing the Number of Points 4-6

4.5. Historical Perspective 4-7

4.6. Evaluation of Polynomials 4-7

4.7. Piecewise Linear Interpolation 4-8

4.8. Piecewise Cubic Functions 4-8
  4.8.1. Hermite Cubics 4-8
  4.8.2. Cubic Splines 4-9

4.9. PCHIP 4-10

4.10. Cubic Hermite Interpolation: Details 4-10

4.11. Cubic Splines 4-10

4.12. Practical Differences Between Splines and cubic Hermite polynomials 4-10

Chapter 5. Numerical Quadrature 5-1

5.1. Introduction 5-1
CONTENTS

5.2. One-dimensional Quadrature Rules and Formulas 5-1
   5.2.1. Newton-Cotes Formulas 5-2
      5.2.1.1.  
      5.2.1.2.  
      5.2.1.3.  
   5.2.2. Gaussian Quadrature 5-7

5.3. Change of Interval 5-8

5.4. Compound Quadrature Rules and Error Estimates 5-9

5.5. Gauss-Kronrod Quadrature Rules 5-9

5.6. Automatic and Adaptive Quadrature Algorithms 5-10

5.7. Subroutines Q1DA and QK15 5-10

5.8. Data Integration 5-10

5.9. Improper Integrals 5-10
   5.9.1. Truncation 5-11
   5.9.2. Transformation 5-11
   5.9.3. Weighted Quadrature Rules 5-11

Chapter 6. Linear Least-Squares Data Fitting 6-1

6.1. Introduction 6-1
   6.1.1. Linear Least Squares Problems 6-1
   6.1.2. Source of Linear Least Squares Problems 6-2
   6.1.3. Weighted Least Squares Problems 6-2

6.2. Exploring Data 6-2

6.3. The Normal Equations 6-2

6.4. Orthogonal Factorizations 6-3

6.5. Subroutine SQRLS 6-6

6.6. Historical Perspective: Gauss 6-6

6.7. Degenerate Least-Squares Problems 6-6

6.8. The Singular Value Decomposition 6-6
   6.8.1. Summary 6-8

Chapter 7. Solution of Nonlinear Equations 7-1

7.1. Introduction 7-1
   7.1.1. Nonlinear Equations 7-1
   7.1.2. Root Finding and Optimization 7-2
   7.1.3. Iterative Methods and Errors 7-2

7.2. Methods for Computing Real Roots 7-3
   7.2.1. The Bisection Method 7-3
   7.2.2. Fixed Point Iteration 7-4
   7.2.3. Aitken’s Δ² Method 7-5
   7.2.4. Newton’s Method 7-6
   7.2.5. The Secant Method 7-7
   7.2.6. Summary of One-Dimensional Methods 7-7

7.3. Subroutine FZERO 7-8

7.4. Historical Perspective 7-8
7.5. Systems of Nonlinear Equations 7-8
   7.5.1. Review of Multivariate Calculus 7-8
   7.5.2. Fixed Point Iteration 7-9
   7.5.3. Newton’s Method 7-10
   7.5.4. Quasi-Newton methods 7-11

7.6. Subroutine SNSQE 7-11

Chapter 8. Ordinary Differential Equations 8-1
8.1. Introduction 8-1
   8.1.1. Theory of ODEs 8-1
   8.1.2. Numerical Methods 8-3
   8.1.3. Errors 8-4
8.2. Stable and Unstable Equations, Numerical Methods 8-5
8.4. Euler’s Method 8-5
8.5. Accuracy and Stability 8-6
   8.5.1. Accuracy 8-6
   8.5.2. Stability 8-7
8.6. Order of an Integration Method 8-8
8.19. Taylor Series and Runge-Kutta Methods 8-8
   8.19.1. Taylor Series Methods 8-8
   8.19.2. Runge-Kutta Methods 8-9
8.8. Implicit Methods 8-13
8.9. Multi-step Methods 8-14
   8.9.1. The Predictor-Corrector Scheme 8-14
   8.9.2. The Adams-Bashforth-Moulton Methods 8-15
   8.9.3. Stability 8-17
8.10. Order and Error of a Multi-step Method 8-19
8.11. Stability for Multi-step Methods 8-19
8.7. Subroutine SDRIV2 8-19

Chapter 9. Optimization and Nonlinear Least Squares 9-1
9.1. Introduction 9-1
   9.1.1. Accuracy 9-1
9.2. One-Dimensional Optimization 9-1
   9.2.1. Golden Section Search 9-1
   9.2.2. Newton’s Method 9-2
   9.2.3. Successive Parabolic Interpolation 9-3
   9.2.4. Summary of One-Dimensional Methods 9-3
9.3. Subroutine FMIN 9-3
9.4. Optimization in Many Dimensions 9-3
   9.4.1. Surfaces and Level Sets 9-4
   9.4.2. Coordinate Search 9-4
   9.4.3. Steepest Descent 9-5
   9.4.4. Conjugate Gradient 9-6
   9.4.5. Newton and Quasi-Newton Methods 9-8
   9.4.6. Summary of Higher Dimensional Optimization 9-9
9.5. Subroutine UNCMIN 9-9
CHAPTER 0

Introduction to these Lecture Notes

0.1. About These Notes

These are the notes I use to teach Math 473 (Introduction to Scientific Computation). They are not meant to be used by themselves, but in conjunction with the textbook Numerical Methods and Software by Kahaner, Moler and Nash, abbreviated NMS from now on.

I make detailed notes whenever I teach a class, but they are usually hand-written. During a period of temporary insanity a few years ago I decided to typeset them, so I could update them more easily. I am not sure I will ever do that again for another course, but here they are. After teaching the course a number of times, I updated the notes in Spring 1992.

There is no guarantee that everything I do in class is in these notes. Every time I teach the course I make some minor changes, and it is a major effort to update the notes. Still, they should be fairly close to the material I actually cover.

The numbering of chapters is taken directly from NMS wherever possible, so it is easier to cross-reference these notes with the book.

These notes can be used in several ways. If you have to miss a class, you can find the missing material here. You can bring the notes to class and just write remarks on them, instead of copying everything down; this will give you more time to concentrate on what is said. You can use them to study for exams.

These notes do not replace NMS, and they do not replace regular attendance at class either. From experience I can tell you that regular attendance and good grades are highly correlated. The students who are missing the most are usually the ones who can least afford it. I try to keep these notes as complete as possible, but there may still be something I do in class that is not in the notes.

These notes are available for on-screen viewing or printing on Project Vincent. See appendix A for details. You may be able to get a printed copy at one of the copying shops around campus. Watch for an announcement in class.

If you find any mistakes or unclear passages in these notes, please let me know, either in person or via e-mail to keinert@iastate.edu.

0.2. Mathematical Background

You will need to know basic calculus (differentiation, integration, Taylor’s theorem) and basic linear algebra (vectors, matrices and how to calculate with them). I will review linear algebra at the beginning of chapter 3. For chapter 9, we will need some advanced calculus. I will review the necessary material at the beginning of chapter 9.

The single most important fact we need from calculus is Taylor’s theorem, so I will state it here:

**Theorem 0.1 (Taylor’s Theorem).** If \( f \) is \((n + 1)\) times continuously differentiable on the interval \((a, b)\), if \( x \in (a, b) \), and if \( h \) is so small that \((x + h) \in (a, b)\), then

\[
\begin{align*}
f(x + h) &= f(x) + f'(x)h + \frac{h^2}{2!}f''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n,
\end{align*}
\]

0-1
where the remainder term $R_n$ is given by

$$R_n = \frac{h^{n+1}}{(n+1)!} f^{(n+1)}(\xi),$$

with $\xi$ some point between $x$ and $x + h$.

In many applications, $h$ is a stepsize. For example, the area under a curve (definite integral) is approximated by subdividing the interval into small intervals of length $h$, and approximating the area of each one by a rectangle or trapezoid. For each choice of stepsize $h$, you get a different approximate answer, so the error depends on $h$.

Many numerical methods are based on some partial Taylor series expansion, and the error is some (known) constant times the remainder term. Higher derivatives of the function $f$ involved are usually not available, and we don’t know what the point $\xi$ is, anyway, but for small $h$ the $\xi$ has to be close to $x$, so we can assume that the $f^{(n+1)}(\xi)$ is approximately equal to some (unknown) constant.

Together, this tells us that

$$\text{error} \approx C h^p,$$

where $C$ is some unknown constant. The exponent $p$ is called the order of the method. This situation is usually expressed in the notation

$$\text{error} = O(h^p) \quad \text{as} \quad h \to 0,$$

which is pronounced the error is of order $h^p$ or the error is big $O$ of $h^p$ as $h \to 0$.

In practice, if we calculate an approximate answer for a single $h$, we have no idea how large the error is, since we don’t know the $C$. However, if we calculate approximate answers for several different $h$, we can estimate the error.

If we calculate the numerical answer for stepsizes $h$ and $h/2$, and the method is first order ($p = 1$), the error in the second case will be smaller by a factor of 2. If the method is of order 2, the second error will be smaller by a factor of 4. If the method is of order 3, the second error will be smaller by a factor of 8, and so on. This can be exploited (see extrapolation in chapter 2).

We will see many examples of this type of error estimate later.

### 0.3. Computer Background

You will need a working knowledge of Fortran and of the computer system you plan to use. Look at the appendices for more information.
CHAPTER 1

Introduction

1.1. Why a New Book?

Read this section on your own.

1.2. The Subroutines

Read this section in the book. It explains what is on the disk in the textbook. Here, I want to talk about available subroutine libraries in more detail.

A subroutine library is a collection of subroutines, usually written in Fortran, that solve common mathematical problems. The subroutines are meant to be used as building blocks for larger programs.

1.2.1. What is available? Common commercial libraries include NAG, IMSL, SLATEC and PORTLIB. These libraries were developed at large corporations or government agencies for in-house use. Afterwards, they figured they might as well try to make some money with them, so they started offering them to the public. These packages are usually quite expensive, and are only offered on larger machines for that reason. NAG is available on both Vax and PV at Iowa State; I am not sure about the rest.

Public domain libraries were usually written by researchers or their slaves (graduate students), and are made available to the public as a courtesy. These libraries are usually much smaller in scope, so you need a whole bunch of them, but they are free.

Some of the more common public domain libraries are

xerror: A collection of error-reporting subroutines. It is useful to be able to redirect error messages, or turn them off altogether. Many public domain subroutines do their error reporting through xerror, so there is a central place to catch them.

machcon: Three short routines that define machine constants. Through them, other routines can find out what type of machine they run on, what the available accuracy is, and so on. Getting this information from machcon, rather than using the values of a particular machine, makes routines more portable.

blas: The Basic Linear Algebra Subroutines. Routines that perform basic vector and matrix computations, like dot products, length of a vector, matrix times vector, etc. Most linear algebra programs and some others are based on blas.

The reason here is optimization. Supercomputers are designed for fast vector and matrix calculations, because often the bulk of the computation is there. However, to achieve full speed, the programmer has to arrange the order of calculations carefully for each type of machine. If the program is based on blas, most of the possible speedup can be gained by simply optimizing the blas.

linpack: A collection of matrix equation solvers. The matrix solver sgefs in the textbook is based on linpack.

eispack: Eigenvalue and eigenvector routines.

quadpack: Numerical integration (quadrature) routines.

odepack: Ordinary Differential Equation solvers.
fnlib: Special functions of mathematics and physics, like Bessel functions, Airy functions, etc.

There are many more. Many of these routines have been collected into a super-library called cmlib (core mathematics library). Cmlib is available on PV in /home/math/lib/cmlib.

1.2.2. Where do I find them? Suppose you need a subroutine to solve a particular type of problem you have. Where do you start looking?

1. Find out what commercial packages are available on your system.
2. Look around the departmental lockers on PV, like math, physics, aero, etc. Cmlib in the math locker would be my first choice. (I may be biased, of course, since I installed it).
3. Look in netlib. Netlib is an archive containing most of the public domain scientific software. To get started, send an e-mail message with the single line send index to either netlib@ornl.gov (Oak Ridge National Lab) or to netlib@research.att.com (what is left of Bell Labs).
   You can request the programs themselves by e-mail, or you can fetch them from research.att.com via ftp:

   % ftp research.att.com
   Connected to research.att.com.
   Name (research.att.com:keinert): netlib
   Password (research.att.com:netlib): keinert@iastate.edu
   ...

   (Use your own name instead of keinert).
4. Look through back issues of ACM TOMS (Association for Computing Machinery: Transactions on Mathematical Software, or something like that). Many of the programs are available on netlib.
5. Post a request for help on one of the Usenet groups, like sci.math.numerical-analysis. Maybe somebody out there in net-land can help you.
6. If you know the name of the package you need, ask archie. Look at the manual page for archie on your system, or telnet to quiche.cs.mcgill.ca and log in as archie.

Once you have located a source, you usually get the code via anonymous ftp. Log into the site as user anonymous, and use your e-mail address as the password.

1.3. Mathematical Software: An Example

Read this on your own.

1.4. Design of Scientific Subroutines

Read the remaining sections (1.4 through 1.8) on your own. This section of my notes summarizes the contents of sections 1.4 through 1.6 in the book.

I want to address the question: What are the characteristics a good scientific subroutine should have, and why? After reading this part, it should be obvious that convenience is not the only reason for using “canned” subroutines. They are simply better than anything you could hope to write yourself with reasonable effort.

Some characteristics of a good general purpose subroutine are

**Portability:** The routine runs on a variety of machines and compilers. The programmer can insure that by following some simple rules:
- Stick with standard ANSI Fortran and stay away from local extensions. Many compilers are able to flag nonstandard usage to help the programmer.
- Don’t rely on the capabilities of any particular compiler. If one compiler sets all variables to zero initially or leaves variables in subroutines undisturbed between calls, that does not mean that all of them do.
- Use the machcon routines to find out about machine characteristics.

**Parameter Checking:** Before the routine does any work, it checks the input parameters to see if they make sense. It notices errors like arrays with negative dimensions, a requested accuracy of 8 decimals when the machine can only produce 7 decimals, and things like that.
Accuracy Requests: A good routine is efficient. If the user only needs 3 digits of accuracy, the routine should only do enough work to produce that, rather than go for full accuracy all the time. A good routine lets the user request how much accuracy is needed.

Error Estimate: The routine estimates how accurate the result is, and reports it back.

Error Messages: Error messages are printed through `xerror` or something similar, so that they can be turned off or redirected.

Error Code: In addition to printing a message, the routine returns an error code, so that the main program knows what is going on.

Scratch Storage: Many calculations need some intermediate storage. Ideally, the subroutine requests that from the operating system, and returns it after use. In reality, Fortran has no such mechanism. Many implementations of Fortran do, but they are not portable. So, any scratch space whose size is not known in general is usually provided by the user, making extra parameters necessary. Not pretty, but a necessary evil with the current Fortran. One solution would be to write a separate library that does memory management and make the other routines use that. Then, you would only have to rewrite that one library for every machine.

Optimization: The routine can easily be adapted to use special features of a particular machine, like multiple processors, vector registers, and the like. The way to do that is to identify the most computation-intensive parts of the code, and put them in a separate package that can be optimized for a particular machine. Often, these parts are vector and matrix computations, which is why the `blas` were developed.

It is not easy to achieve all these objectives. For example, in chapter 8, we will learn some ways to solve differential equations numerically. A typical program you write for this course is about 50 lines long. The source code for the ODE solver on the disk is about 2,500 lines. The algorithm at the heart of it takes 50 to 100 lines, the rest is error checking, error estimation, and the like.
CHAPTER 2
Computer Arithmetic and Computational Errors

2.1. Introduction

Read this section in the book on your own. It may make more sense if you read it after the rest of the chapter.

2.2. Representation of Numbers

It is impossible to cover all kinds of computers. I will concentrate on the ones you are likely to use: The Vax and any computer conforming to the IEEE standard. The latter includes PV and most micros and workstations (IBM PC, Mac, Sun, etc.). When I say “the computer does this”, this means “any computer you are likely to use does this”. There are exceptions to everything.

The standard numerical data types available in most programming languages are integers, reals and double precision reals. Let us take a brief look at how they are represented on the computer. In all of the following, we only look at positive numbers. Negative numbers are represented by the two’s complement of the corresponding positive number. If you don’t know what that means, don’t worry about it.

I assume you are familiar with representation of numbers in various bases. For example,

\[(123)_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0,\]
\[(1101)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (13)_{10}.\]

For fractional numbers, the same system works

\[(1.23)_{10} = 1 \cdot 10^0 + 2 \cdot 10^{-1} + 3 \cdot 10^{-2},\]
\[(1.011)_{2} = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} = (1.25)_{10}.\]

Most machines use base 2 numbers internally. The HP 41C calculator uses base 10. I will use base 10 examples in class, since that is what everybody is used to, even though your computer really uses base 2.

Most machines set aside a fixed amount of storage per number. For the machines you are likely to use, it is 32 bits = 4 bytes per integer or real, 64 bits = 8 bytes per double precision number. (Wylbur is different, I think).

No matter which system you use, a machine number is stored in a finite space, so only finitely many different numbers can be represented, and the results of computations need to be rounded to fit into the available storage.

2.2.1. Integers. An integer is represented on the computer as a straight binary number of 31 bits, plus one sign bit (0 = positive, 1 = negative). Thus, the largest integer that can be represented is

\[(0111\ldots111)_2 = 2^{31} - 1 = 2,147,483,647.\]

Since integers are usually used only as array indices and loop counters, this is not likely to cramp your style.
2.2.2. Reals. Early computers worked with fixed point arithmetic, where a certain number of places before and after the decimal point are used. This is still useful in some applications such as banking, where fractions of cents are not desired, but for scientific computation floating point numbers are used.

A number such as 123 can be written in scientific notation in many different ways:

\[ 123 = 1.23 \cdot 10^2 = 0.0123 \cdot 10^4 = 12,300 \cdot 10^{-2} = \cdots \]

The power of 10 (or 2, or whatever base you are using) is called the exponent, the number in front is called the mantissa. To standardize things, we define a normalized number as one where the exponent has been adjusted so that the mantissa has exactly one nonzero digit in front of the decimal point. In the above example, that is \(1.23 \cdot 10^2\). Each number except zero has exactly one normalized form.

In base 2, the normalized form of the number \(5/16 = (0.0101)_2\) is

\[ 5/16 = (1.01)_2 \cdot 2^{-2}. \]

Note that a normalized binary number always has the digit 1 before the binary point, since that is the only nonzero binary digit there is.

In the IEEE Standard, a real number takes up 32 bits, divided into 1 bit for the sign, 8 for the exponent and 23 for the mantissa. Instead of having a sign for the exponent part, a bias of 127 is added to the true exponent. Thus, the stored exponent is the true exponent + 127. The leading digit 1 of the mantissa is not stored, since it is always there. This is called the hidden bit.

As an example, the number \(5/16\) is represented as

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111011</td>
<td>01000000000000000000000000000000</td>
</tr>
</tbody>
</table>

The stored exponent is the number 125 in binary form (true exponent of \((-2)\) plus bias 127). The 1. in front of the mantissa is hidden.

The IEEE standard provides for \(\pm \infty\) and for NaN (not a number). The result of \(1/0\) is \(+\infty\), the result of \(\sqrt{-1}\) is NaN. (Stored) exponent 255 is reserved for these special non-numbers.

Very small numbers can be represented as denormalized numbers, which have lower accuracy. (Stored) exponent 0 is reserved for that. We won’t go into details of denormalized numbers.

Stored exponents therefore go from 1 to 254 and represent true exponents of \(-126\) to 127.

The largest number is \((1.1111 \ldots)_2 \cdot 2^{254-127}\), which is about \(3.4028 \cdot 10^{38}\). The smallest normalized number is \((1.000 \ldots)_2 \cdot 2^{1-127}\), which is about \(1.175 \cdot 10^{-38}\). The accuracy is 24 bits (23 stored and 1 hidden), which corresponds to about seven decimals, since \(2^{24} \approx 10^7\). Denormalized numbers can be smaller than \(10^{-38}\), but they also have less accuracy.

The Vax has no denormalized numbers and a slightly different range: the smallest and largest numbers are about \(0.29 \cdot 10^{-38}\) and \(1.7 \cdot 10^{38}\). The bias is 128, the accuracy is the same (24 bits \(\approx\) seven decimals).

What you should keep in mind from all this is:

**For most computers, the range of representable single precision real numbers is approximately from \(10^{-38}\) to \(10^{38}\). The accuracy is about seven decimals.**

Among other things, this means that it is useless to print out numbers to more than seven decimals. Anything past the seventh digit is garbage. Note that your pocket calculator probably has 10 digits (or even 12 or 13 internally), so it is more accurate than a big computer.

2.2.3. Double Precision. If seven decimals are not enough accuracy, you can use double precision numbers. The idea is the same as for reals, but you use 64 bits instead of 32.

Many compilers accept the notation `real*4` for single precision reals, which take 4 bytes of storage each, and `real*8` for double precision numbers, which take 8 bytes of storage each.

Here the IEEE standard and the Vax differ considerably. In the IEEE standard, the exponent increases to 11 bits, the mantissa increases to 53 bits (52 stored + 1 hidden), which gives a range of about \(10^{-308}\) to \(10^{308}\), and 15 decimals accuracy.

The Vax keeps the 8 bit exponent, which leaves 56 bits for the mantissa. Thus, the range of double precision numbers is the same as for reals (about \(10^{-38}\) to \(10^{38}\)), but the accuracy is 16 decimals.

The Vax also has a kind of double precision numbers similar to the IEEE standard with 11-bit exponent, but has to simulate them in software. This increases execution time by a factor of about 100 (no kidding!). The Vax also has quadruple precision, for those days when double precision just isn’t enough. That is even
slower. You are discouraged from using these kinds of numbers. If you really want to know more, get URM 146 from the computer center.

What is the penalty for using double precision? It takes twice as much storage and is slower. Runtime on the Vax increases by about 60 %. On a micro with a co-processor, I would assume only 10 – 20 % runtime increase, mostly for extra data moves. The co-processors work internally with more than double precision, anyway. Without a co-processor, my guess is that double precision would take at least 3 times as long as single precision. This is on top of the fact that single precision is already a factor of 20 slower than with a co-processor. (This number I determined experimentally).

For the type of small programs you will write in this course, there is no problem with using double precision. However, you should only use it if single precision fails.

The reason is this: For every numerical routine, no matter how well it is written, you can find some data that it cannot handle. It is part of the learning process to observe how far you can push a program before it fails. Making programs fail is much easier in single precision.

2.2.4. Rounding. Most numbers are represented exactly only by infinite decimal or binary expansions. In order to fit them into a 24-bit mantissa, we need to cut them off somehow. There are two ways of doing this:

- **Truncation or chopping**, where all bits past the 24rd are simply thrown away.
- **Rounding**, where we look at the 25th bit first. If it is 0, we truncate; if it is 1, we round up.

This is analogous to representing the number 2/3 in three decimal digits. Truncated you get 0.666, rounded you get 0.667.

Rounding is normally preferred, since we expect only half as much error as truncation on the average. Also, we expect errors to cancel more, since sometimes we round up and sometimes down. (Truncation is the same as always rounding down).

2.3. Machine Constants

There are three constants that describe the floating point characteristics of a computer.

**OFL:** The *overflow limit*, which is the largest floating point number that can be represented. In single precision, it is near $10^{38}$. If the result of a computation gets larger than OFL, and *overflow error* occurs. This is usually fatal to the program.

**UFL:** The *underflow limit*, which is the smallest positive normalized number that can be represented. In single precision, it is near $10^{-38}$. If the result of a computation gets smaller than UFL, an *underflow error* occurs. Usually, the number is rounded to zero, and the program continues (with or without a message).

The **machine epsilon** $\epsilon$: $\epsilon$ represents the accuracy of the machine, in the sense that the relative error from rounding a number to machine accuracy is of order $\epsilon$. One definition is that $\epsilon$ is the smallest floating point number such that

$$1 + \epsilon > 1.$$

In single precision, $\epsilon$ is approximately $10^{-7}$, since we have 7 decimals of accuracy.

The behavior of Vax and PV in response to overflow and underflow errors is explained in the Fortran write-up (appendix B).

2.4. Errors in Scientific Computing

In numerical analysis, errors are defined as follows. Assume that $x$ is some number (for example the number $\pi$), and $x^*$ is an approximation to $x$ (for example the machine representation of $\pi$). Then

- absolute error = $x - x^*$ (or $|x - x^*|$)
- relative error = $\frac{x - x^*}{x}$ (or $\frac{|x - x^*|}{|x|}$)

The first form is called *signed error*, since it can be positive or negative. Sometimes it is more convenient to use signed errors, sometimes it is more convenient to use absolute values. I will switch back and forth between the two. You may do the same, unless otherwise stated.
The signed error has the advantage that you know whether your result is too large or too small. The unsigned error has the advantage that it is often easier to handle.

Note that “absolute error” has nothing to do with absolute values.

Usually, the relative error is a better measure for accuracy than the absolute error. If your absolute error is 0.001, this may be very good (if the true answer is 10,000) or very bad (if the true answer is 0.00000001).

Sometimes the relative error can be misleading, usually when the true result is very small. For example, suppose the true result is $10^{-10}$, and your numerical answer is $10^{-8}$. The error is 100 times as large as the result, but maybe you only need to know that the value is very small. In these cases, the absolute error is a better measure of accuracy.

Most good scientific subroutines let you specify either absolute or relative error bounds, or a combination of both.

2.4.1. The Condition Number. The difficulty of a problem is measured by the condition number. It is defined as

$$\text{condition number} = \max \frac{\text{relative error in output}}{\text{relative error in input}}$$

The fraction is the factor by which (relative) input errors get magnified. The “max” means we consider the worst possible case. In any particular case, the error magnification may be less.

In practice, this means the following: suppose you know that a particular problem has a condition number of 1,000. You are computing in single precision. The input data must be assumed to carry a relative error of $10^{-7}$ (the machine $\epsilon$). Therefore, the output data may carry a relative error of $1,000 \cdot 10^{-7} = 10^{-4}$. In other words: you can only trust 4 decimals in your answer. In a particular case, your error may be less, but you don’t know that. You have to assume the worst.

The exact value of the condition number is not that important, and is usually not known, anyway. What counts is the order of magnitude. A problem with a small condition number (tens or hundreds) is well-posed, a problem with a large condition number (millions) is ill-posed. A problem with a condition number above $10^7$ cannot be solved at all in single precision.

There are really two condition numbers at work: the condition number of the problem, and the condition number of the algorithm. An algorithm is a particular sequence of steps the computer follows to get the answer.

The condition number of the problem is the error magnification you get when you do exact calculations. The condition number of a particular algorithm is the error magnification you get when you solve the problem on a computer, using this particular algorithm.

The condition number of any algorithm is always at least as large as the condition number of the problem. There are some problems that simply cannot be solved. For solvable problems, you have to watch out that your algorithm does not make things worse than they already are. Look at the examples in section 2.7 in these notes.

2.4.2. Sources of Errors. There are many sources of error in scientific computation. Some of them are listed here. Note that in practice, the distinction between the types of error is not always clearcut.

**Arithmetic Error:** This comes from rounding numbers to machine accuracy. Every input number, and every result of a calculation acquires a relative error of the order of the machine $\epsilon$.

**Discretization Error:** Computers can only do computations that take a finite number of steps. Any computation dealing with a continuum needs to be broken into discrete steps. For example, to calculate $\int_a^b f(x) \, dx$ numerically, you have to sample the function at a finite number of points. This causes an error, because you are ignoring all the other points.

**Truncation Error:** The same as discretization error, except you cut down a countably infinite operation to a finite one. For example, you cannot add up an infinite series; you have to stop after a finite number of terms. The ignored terms lead to truncation error.

**Data Error:** Your data may consist of measurements of something and already carry a measurement error.
2.4.3. Arithmetic Errors. Here we consider the effect of roundoff errors. What are the condition numbers of basic operations (addition, multiplication, calculation of exp, sin, sqrt, etc.)?

Fact: With one exception, all basic operations available in Fortran are well-behaved. This means the condition number is near 1, so that the result of a calculation has approximately the same relative error as the input. The one exception is cancellation, which can occur in addition/subtraction.

Cancellation happens when you subtract two almost equal numbers. Suppose you work in seven digit decimal arithmetic. The number 0.1234567 stands for some number between 0.12345665 and 0.123456749999 . . . . It has a relative error of order $10^{-7}$.

Likewise, the number 0.1234576 stands for anything between 0.12345755 and 0.12345764999 . . . . If I subtract these numbers, I get $0.0000009$. However, the true answer could be anywhere between $0.12345764999 . . . - 0.12345665 \approx 0.0000010$

and

$0.12345755 - 0.12345674999 . . . \approx 0.0000008$,

so the relative error in the result could be up to 10 %, depending on what “true” numbers the machine numbers stand for. The original relative error has been multiplied a millionfold.

There is no point in worrying about cancellation every time you add or subtract two numbers. You just have to recognize it when you see it. Look at section 2.7 in these notes examples on how to avoid cancellation in some cases.

2.5. Extrapolation

Many algorithms are based on Taylor series expansions. The (absolute) error can be calculated from the series remainder and is of the form

$$ error = c_p h^p + c_{p+1} h^{p+1} + c_{p+2} h^{p+2} + \cdots $$

or similar, so

$$ error = O(h^p). $$

Example: One method to approximate a derivative numerically is

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

Taylor series expansion gives

$$ f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \cdots $$

$$ f(x + h) - f(x) = f'(x) + \frac{h}{2} f''(x) + \frac{h^2}{3!} f'''(x) + \cdots $$

$$ \frac{f(x + h) - f(x)}{h} - f'(x) = \frac{h}{2} f''(x) + \frac{h^2}{3!} f'''(x) + \cdots, $$

so that

$$ error = c_1 h + c_2 h^2 + \cdots = O(h). $$

A better method is

$$ f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}. $$
Taylor series expansion gives

\[ f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \cdots \]

\[ f(x - h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \cdots \]

\[ \frac{f(x + h) - f(x - h)}{2h} = f'(x) + \frac{h^2}{3!}f'''(x) + \frac{h^4}{5!}f^{(5)}(x) + \cdots \]

\[ \text{error} = \frac{f(x + h) - f(x - h)}{2h} - f'(x) = \frac{h^2}{3!}f'''(x) + \frac{h^4}{5!}f^{(5)}(x) + \cdots , \]

so that

\[ \text{error} = c_2h^2 + c_4h^4 + \cdots = O(h^2). \]

If the order of the error is known, we can eliminate the leading error term. This is called *extrapolation*. Let \( A(h) \) be the approximate result calculated with stepsize \( h \). Assume

\[ \text{error} = O(h^p) \approx ch^p. \]

Then

\[ A(h) \approx \text{true result} + ch^p, \]

\[ A(h/2) \approx \text{true result} + c(h/2)^p, \]

so

\[ A(h) - A(h/2) \approx ch^p \left[ 1 - \left( \frac{1}{2} \right)^p \right], \]

\[ \text{true result} \approx A(h) - ch^p = A(h) - \frac{A(h) - A(h/2)}{1 - (1/2)^p}, \]

\[ \text{true result} \approx \frac{2^p A(h/2) - A(h)}{2^p - 1}. \]

After extrapolation, the next term in the error series takes over.

**Example:** Calculate \( f'(x) \) numerically for \( f(x) = e^x \), \( x = 1 \). Using the first formula, we know the error is \( O(h) \), so \( p = 1 \).

I got the following values on my calculator:

<table>
<thead>
<tr>
<th>( h )</th>
<th>( \frac{f(x + h) - f(x)}{h} )</th>
<th>extrapolated values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2.858841955</td>
<td>2.715929629</td>
</tr>
<tr>
<td>0.05</td>
<td>2.787385792</td>
<td>2.718296492</td>
</tr>
<tr>
<td>0.025</td>
<td>2.752545284</td>
<td>2.718281791</td>
</tr>
<tr>
<td>0.0125</td>
<td>2.735342100</td>
<td>2.718138916</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( h )</th>
<th>error</th>
<th>error of extrapolated values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.14056126</td>
<td>-0.002352199</td>
</tr>
<tr>
<td>0.05</td>
<td>0.069103964</td>
<td>0.000014663</td>
</tr>
<tr>
<td>0.025</td>
<td>0.034263456</td>
<td>0.000000037</td>
</tr>
<tr>
<td>0.0125</td>
<td>0.017060272</td>
<td>0.000142912</td>
</tr>
</tbody>
</table>
The first column of extrapolated values is calculated from the derivative estimates by

\[ \text{true result} \approx \frac{2A(h/2) - A(h)}{1}. \]

The more accurate answer \( A(h/2) \) is the value from the row below, the less accurate answer \( A(h) \) is the value from the row above. After the leading error term \( c_1 h \) has been eliminated, the leading error term is now \( c_2 h^2 \) (see above). Thus, the extrapolated values have error \( O(h^2) \).

We can extrapolate them again, this time with

\[ \text{true result} \approx \frac{4A(h/2) - A(h)}{3}, \]

to get a new column with error \( O(h^3) \), and for the last value with

\[ \text{true result} \approx \frac{8A(h/2) - A(h)}{7}. \]

If you look at the corresponding errors, you can verify that the error in each column behaves as it should: In the column where the error is \( O(h) \), each error value is approximately 1/2 of the one before. In the column where the error is \( O(h^2) \), each error value is approximately 1/4 of the one before, etc.

**Example:** Repeat the previous example, using the better derivative formula. This time, the original error is \( O(h^2) \), and subsequent columns have error \( O(h^4), O(h^6) \), etc.

<table>
<thead>
<tr>
<th>( h )</th>
<th>( \frac{f(x+h) - f(x-h)}{2h} )</th>
<th>extrapolated values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2.722814564</td>
<td>2.718281261</td>
</tr>
<tr>
<td>0.05</td>
<td>2.719414587</td>
<td>2.718281830</td>
</tr>
<tr>
<td>0.025</td>
<td>2.718564992</td>
<td>2.718281829</td>
</tr>
<tr>
<td>0.0125</td>
<td>2.718352618</td>
<td>2.718281827</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( h )</th>
<th>error</th>
<th>error of extrapolated values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.004532735</td>
<td>-0.000000567</td>
</tr>
<tr>
<td>0.05</td>
<td>0.001132759</td>
<td>0.000000001</td>
</tr>
<tr>
<td>0.025</td>
<td>0.000283163</td>
<td>0.000000001</td>
</tr>
<tr>
<td>0.0125</td>
<td>0.000079789</td>
<td>-0.000000002</td>
</tr>
</tbody>
</table>

Again, compare the error columns to the predicted behavior.

### 2.6. Historical Perspective

Read this on your own.

### 2.7. Examples

Let us look at some examples of the principles mentioned in this chapter.
2.7.1. Avoiding Ill-conditioned Algorithms. As mentioned above, some problems are ill-conditioned, and no clever algorithm will save you. All you can do is to avoid ill-conditioned algorithms for well-conditioned problems.

To determine whether your problem is ill-conditioned or not, ask yourself the question: if I change the data just a little bit, how will that affect the result?

For example, in numerical integration a small change in the function will produce a small change in the area under the curve: numerical integration is well-conditioned. In numerical differentiation, a small change in the function can produce a large change in the result (see example 2.2 in NMS): numerical differentiation is ill-conditioned.

If you get bad results for a well-conditioned problem, there is usually cancellation in your algorithm somewhere. All algorithms break down into basic additions, multiplications, etc. at the lowest level. Unless you are doing billions and billions of these, only cancellation would produce a noticeable error in the result. Sometimes, the errors can be avoided by some simple rearrangement of the algorithm. Details depend on the individual case.

Example: Find \( f(x) = 1 - \cos(x) \) for \( x = 10^{-5} \) with eight correct decimals on your hand calculator.

If you simply punch it out, you get zero. The reason is cancellation: \( \cos(x) \approx 1 \) for small \( x \). But

\[
\begin{align*}
\cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots, \\
f(x) &= \frac{x^2}{2!} - \frac{x^4}{4!} + - \cdots, \\
f(10^{-5}) &= \frac{1}{2} \cdot 10^{-10} - \frac{1}{24} \cdot 10^{-20} + - \cdots \\
&= 5 \cdot 10^{-11}
\end{align*}
\]

to 10 places of accuracy. \( \Box \)

Example: Calculate \( f(x) = \sqrt{x + 1} - \sqrt{x} \). Again, you get zero for large \( x \). With a little algebra

\[
\begin{align*}
f(x) &= (\sqrt{x + 1} - \sqrt{x}) \cdot \frac{\sqrt{x + 1} + \sqrt{x}}{\sqrt{x + 1} + \sqrt{x}} \\
&= \frac{1}{\sqrt{x + 1} + \sqrt{x}}
\end{align*}
\]
and you can calculate this very accurately. \( \Box \)

Example: See example 2.7 in NMS (power series of \( e^x \) for large negative \( x \)). \( \Box \)

2.7.2. Interaction of Errors. It happens frequently that the total error is a sum of two or more errors which depend on some parameter \( h \). When \( h \) changes, one of the errors decreases and another one increases. The result is that there is a certain minimum error that you cannot get below.

This happens especially in ill-posed problems; a standard example is numerical differentiation (examples 2.2, 2.8 in NMS). The following example is basically example 2.8 in NMS. I am just repeating it here for completeness.

Example: We approximate the derivative of \( f \) by the formula

\[
f'(x) \approx \frac{f(x + h) - f(x)}{h}.
\]

There is a discretization error, due to replacing the derivative by a difference quotient. From the Taylor series expansion, the (absolute) discretization error is

\[
\text{discretization error} = f'(x) - \frac{f(x + h) - f(x)}{h} \approx \frac{1}{2} f''(\xi) h \approx \frac{1}{2} f''(x) h,
\]
since \( \xi \) must be very close to \( x \).
There is also arithmetic error, since \( f(x + h) \) and \( f(x) \) are close together for small \( h \). This cancellation error can be estimated as follows: Assume the relative error in \( f(x + h) \), \( f(x) \) is each about \( \epsilon \), so the absolute error in each case is about \( f(x) \cdot \epsilon \). (This estimate is good for both, since \( f(x + h) \approx f(x) \)). If one error is positive, the other negative, they could add up to

\[
\text{absolute error} \approx 2f(x)\epsilon/h.
\]

The total error is therefore about

\[
\frac{f''(x)h}{2} + \frac{2f(x)\epsilon}{h}.
\]

Using calculus, we find that the minimum occurs when

\[
\text{optimal } h = 2\sqrt{\frac{\epsilon f(x)}{f''(x)}}.
\]

and the minimal total error is about

\[
\text{minimum total error} = 2\sqrt{\epsilon f(x)f''(x)}.
\]

We can do this much faster by using some rules of thumb. We have no idea what \( f(x), f''(x) \) are in general, anyway, so we set all constants except \( h \) and \( \epsilon \) to 1. The minimal total error is approximately achieved when the discretization and cancellation errors are the same. This leads to

\[
\text{discretization error} \approx h,
\]

\[
\text{arithmetic error} \approx \frac{\epsilon}{h}.
\]

The total error is minimal when \( h \approx \epsilon/h \), or \( h \approx \sqrt{\epsilon} \). For seven digits of accuracy, this means \( h \) should be near \( 10^{-3} \) or \( 10^{-4} \).

Note that the minimal total error is also of order \( \sqrt{\epsilon} \). This means that you cannot calculate a numerical derivative to more than three or four places in single precision.

In example 2.2 in NMS, the actual value for the optimal \( h \) was more like \( 10^{-5} \), but the above is just a crude estimate. Theory and practice are closer in example 2.8, which was done on a Cyber. The Cyber has 60-bit numbers, so \( \epsilon \approx 10^{-14}, h \approx 10^{-7} \).
CHAPTER 3

Linear Systems of Equations

3.1. Introduction

We will cover the material in sections 3.1, 3.2, 3.3, 3.6, 3.7.

Read section 3.1 in the book on your own. I am going to do a lot more stuff here, namely a complete review of vectors and matrices. “Review” means that you should have seen this stuff before, so we will be going quite fast.

I will also cover some of the material of sections 3.2.1 and 3.7 in NMS at this point. It seems to fit better.

I will use the following notation in this section: bold face uppercase letters (like $A$) are matrices, bold face lowercase letters (like $v$) are vectors. All non-bold letters stand for numbers.

3.1.1. Vectors. For the purposes of this course, a vector is a column of real or complex numbers, like this:

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}.$$  

We will be working with vectors of varying sizes. If I want to emphasize the length of a vector, I will call it an $n$-vector. A scalar is a real or complex number.

In this course we will only use real numbers as scalars and vector and matrix entries. However, I will state definitions etc. for complex numbers separately whenever there is a difference from the real case. There is a good chance you will run into complex matrices somewhere along the line. If my remarks about the complex case don’t make sense, just ignore them. Maybe they will make more sense later, when and if you ever look at these notes again. If I don’t say anything, it works the same way for both real and complex numbers.

A special vector is the zero vector

$$0 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$  

Technically speaking, there are many zero vectors, one for each size. We ignore this detail.

We can perform the following arithmetic operations on vectors: addition, subtraction, scalar multiplication and dot product (also called scalar product or inner product). We don’t need a cross product in this course.
The following brief examples should refresh your memory:

\[
\begin{align*}
\mathbf{v} + \mathbf{w} &= \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 2 \\ -1 \\ 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 6 \\ 9 \end{pmatrix}, \\
\mathbf{v} - \mathbf{w} &= \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} - \begin{pmatrix} 2 \\ -1 \\ 3 \\ 5 \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ 0 \\ -1 \end{pmatrix}, \\
3\mathbf{v} &= 3 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 9 \\ 12 \end{pmatrix}, \\
\mathbf{v} \cdot \mathbf{w} &= \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -1 \\ 3 \\ 5 \end{pmatrix} = 2 - 2 + 9 + 20 = 29.
\end{align*}
\]

For complex vectors \( \mathbf{v}, \mathbf{w} \), the dot product is defined as

\[
\mathbf{v} \cdot \mathbf{w} = \mathbf{v}^\ast \mathbf{w}^\ast + \cdots,
\]

where the bar denotes complex conjugation.

A linear combination of vectors is any sum of the form

\[
\sum_{i=1}^{m} \lambda_i \mathbf{v}_i = \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \cdots + \lambda_m \mathbf{v}_m,
\]

where the \( \lambda_i \) are scalars.

A collection of vectors is called linearly dependent, if one of them can be expressed as a linear combination of the others. If none of the vectors can be written as a linear combination of the others, they are called linearly independent.

The collection of all \( n \)-vectors forms the \( n \)-dimensional space \( \mathbb{R}^n \). The standard basis of \( \mathbb{R}^n \) is formed by the vectors

\[
\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \quad \mathbf{e}_n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}.
\]

Every vector can be written in exactly one way as a linear combination of these basis vectors, namely

\[
\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \sum_{i=1}^{n} v_i \mathbf{e}_i.
\]

The “size” of a vector is measured by a norm.

A vector norm is a rule which assigns to each vector \( \mathbf{v} \) a norm (or length) \( \| \mathbf{v} \| \) with the properties

(1) \( \| \mathbf{v} \| \geq 0 \)
(2) \( \| \mathbf{v} \| = 0 \) if and only if \( \mathbf{v} \) is the zero vector.
(3) \( \| \lambda \mathbf{v} \| = |\lambda| \| \mathbf{v} \| \)
(4) \( \| \mathbf{v} + \mathbf{w} \| \leq \| \mathbf{v} \| + \| \mathbf{w} \| \) (triangle inequality)

The best known example is the geometric length of a vector, also called the 2-norm

\[
\| \mathbf{v} \|_2 = \sqrt{v_1^2 + \cdots + v_n^2}
\]

It is easy to visualize, and has other advantages, too.
Other possible norms are the \(L\)-norm, which is defined as the sum of the absolute values of all entries

\[\|v\|_1 = |v_1| + \cdots + |v_n|,\]

and the \(\infty\)-norm, which is the largest absolute value of any entry

\[\|v\|_\infty = \max|v_i|.
\]

**Remark:** The \(p\)-norm, for any \(p \geq 1\), is defined as

\[\|v\|_p = \{|v_1|^p + \cdots + |v_n|^p\}^{1/p}.
\]

The \(1\)-norm and \(2\)-norm are special cases of the \(p\)-norm. The \(\infty\)-norm is the limit of the \(p\)-norms as \(p \to \infty\). You don’t have to know this.

For example, if

\[v = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix},\]

then

\[\|v\|_1 = 6, \quad \|v\|_2 = \sqrt{14}, \quad \|v\|_\infty = 3.
\]

The norms are all different, but they are always of the same order of magnitude.

### 3.1.2. Matrices.

A matrix is a two-dimensional array of real or complex numbers. For example,

\[A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix}.
\]

In everything that has to do with matrices, the first subscript always refers to the row, the second to the column. Thus, this particular \(A\) is a \(2 \times 3\) matrix (2 rows, 3 columns). Its entry \(a_{23}\) is in row 2, column 3.

Note that we can consider an \(n\)-vector as a \(n \times 1\) matrix whenever that is convenient.

An \(n \times n\) matrix is called square. Most matrices we will encounter will be square.

Special kinds of matrices are the following (all except the first one have to be square):

- The **null matrix** or **zero matrix**

\[
\begin{pmatrix}
0 & \cdots & 0 \\
\vdots & \ddots & \vdots \\
0 & \cdots & 0
\end{pmatrix}
\]

There are really many zero matrices, one for each size. We ignore this detail.

- The **identity matrix** \(I\), which has ones on the diagonal and zeros everywhere else:

\[
I = \begin{pmatrix}
1 & 0 & \cdots & 0 \\
0 & 1 & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & 1
\end{pmatrix}
\]

Again, there are really many identity matrices.

- **Diagonal matrices** have nonzero numbers only on the diagonal

\[
D = \begin{pmatrix}
a_{11} & 0 & \cdots & 0 \\
0 & a_{22} & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & a_{nn}
\end{pmatrix}
\]
• Upper and lower triangular matrices

\[ U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix}, \quad L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{n,n-1} & l_{nn} \end{pmatrix}. \]

• A tridiagonal matrix looks like this

\[ \begin{pmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \vdots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n-1,n} & a_{nn} \end{pmatrix}. \]

More generally, a banded matrix is one with nonzero entries only on the diagonal and a few lines parallel to the diagonal.

• Symmetric matrices. A symmetric matrix is symmetric about its main diagonal, so that \( a_{ij} = a_{ji} \).

An example is

\[ A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & -3 & -1 \\ 3 & 1 & 4 \end{pmatrix}. \]

For complex matrices, the corresponding concept is a Hermitian matrix, where \( a_{ij} = \overline{a_{ji}} \). The bar stands for complex conjugation. Note that the diagonal entries have to be real, since \( a_{ii} = \overline{a_{ii}} \).

Two matrices of the same size can be added and subtracted entry by entry. For example,

\[ A + B = \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 3 & 3 \\ 5 & -1 & 3 \end{pmatrix}, \]
\[ A - B = \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix} - \begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -1 & -3 \\ 1 & -1 & 1 \end{pmatrix}. \]

A matrix can be multiplied by a scalar:

\[ 3A = 3 \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 3 & 0 \\ 9 & -3 & 6 \end{pmatrix}. \]

A \( m \times n \)-matrix \( A \) can be multiplied with a \( n \times k \)-matrix \( B \), the result being a \( m \times k \) matrix \( C \). The rule is expressed in words as follows:

The \( ij \)-entry of \( C \) is the dot product of the \( i \)-th row of \( A \) with the \( j \)-th column of \( B \).

Mathematically, this is expressed as follows:

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \]

For example,

\[ AB = \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 2 & 0 \\ 3 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 4 \\ 7 & 8 \end{pmatrix}, \]
\[ BA = \begin{pmatrix} 1 & 2 \\ 2 & 0 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \end{pmatrix} = \begin{pmatrix} 7 & -1 & 4 \\ 4 & 2 & 0 \\ 9 & 2 & 2 \end{pmatrix}. \]

Note that in general \( AB \neq BA \). One product may be defined, but not the other one; they may both be defined, but of different sizes, as in the above example; even if \( A, B \) are both square and of the same size, you get different products in general.
All the other usual rules of arithmetic hold, for example

\[ A + (B + C) = (A + B) + C, \]
\[ A(BC) = (AB)C, \]
\[ A(B + C) = AB + AC, \]

and so on.

If for a matrix \( A \) there exists a matrix \( A^{-1} \) with the property

\[ AA^{-1} = A^{-1}A = I, \]

then \( A^{-1} \) is called the inverse matrix of \( A \), pronounced \( A \) inverse. The following statements can be proved:

- Only square matrices can have inverses.
- Not every square matrix has an inverse.
- If \( A^{-1} \) exists, it is unique.
- \( (A^{-1})^{-1} = A \).
- \( (AB)^{-1} = B^{-1}A^{-1} \).

A square matrix which does not have an inverse is called singular. A square matrix which has an inverse is called non-singular.

In general, the inverse matrix is hard to calculate. For \( 2 \times 2 \) matrices, however, there is a simple formula:

\[
\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.
\]

The matrix is singular if and only if \( ad - bc = 0 \).

The transpose \( A^T \) of a matrix \( A \) is defined as the original matrix with rows and columns interchanged.

\[ a_{ij}^T = a_{ji}. \]

For example,

\[ A = \begin{pmatrix} 2 & 1 \\ 3 & -1 \end{pmatrix}, \quad A^T = \begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix}. \]

For complex matrices, one uses the conjugate transpose or Hermitian transpose \( A^H \) instead, which is defined by

\[ a_{ij}^H = \overline{a_{ji}}. \]

Transposition has the following properties:

- \( (A^T)^T = A \).
- \( (AB)^T = B^T A^T \).
- \( (A^{-1})^T = (A^T)^{-1} \).
- \( A \) is symmetric if and only if \( A^T = A \).
- For any \( A \) (square or not), \( A^T \) and \( AA^T \) are always square, symmetric matrices.
- The dot product of vectors is sometimes written as \( v^T w \). You can verify for yourself that that is indeed the correct formula. This formula is not to be confused with \( vw^T \), which is an \( n \times n \) matrix (work it out!).

These properties are valid for complex matrices if you replace “transpose” by “Hermitian transpose” and “symmetric” by “Hermitian”.

For square matrices, we can define a number called the determinant. Its general definition is a little beyond the scope of this course. The geometric interpretation is this:

Consider the columns of the matrix as vectors. They span an \( n \)-dimensional parallelepiped (something like a skewed cube). The determinant is the volume of this parallelepiped.

If you consider the matrix as a mapping from \( \mathbb{R}^n \) to \( \mathbb{R}^n \), the unit cube gets mapped into this parallelepiped. Thus, the determinant measures how volumes are distorted. If \( \det(A) = 5 \), then the image of anything bounded by straight lines will be five times as large as the original. In multi-dimensional integration, you should have run across the Jacobian, the determinant of a certain matrix. It measures how locally the area or volume is magnified or shrunk because of a substitution.
Some properties of determinants are

- \( \det(AB) = \det(A) \det(B) \)
- \( \det(A^{-1}) = 1/\det(A) \) if \( A^{-1} \) exists
- \( \det(I) = 1 \)
- The following conditions are equivalent:
  - \( A \) is singular
  - \( \det(A) = 0 \)
  - The columns of \( A \) are linearly dependent
  - The rows of \( A \) are linearly dependent

For the cases \( n = 1, n = 2, n = 3 \) the determinant is defined as

- \( \det(a_{11}) = a_{11} \),
- \( \det\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21} \),
- \( \det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31} \)

There is one case where determinants are easy to calculate, and that is for triangular matrices (upper or lower). Here the determinant is simply the product of the diagonal entries.

- \( \det\begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix} = u_{11}u_{22} \cdots u_{nn} \).

The “size” of a matrix is measured by a matrix norm. A matrix norm is a rule which assigns to each matrix \( A \) a norm \( \|A\| \) with the properties

1. \( \|A\| \geq 0 \)
2. \( \|A\| = 0 \) if and only if \( A \) is the zero matrix
3. \( \|\lambda A\| = |\lambda| \|A\| \)
4. \( \|A + B\| \leq \|A\| + \|B\| \) (triangle inequality)
5. \( \|Ax\| \leq \|A\| \|x\| \)
6. \( \|AB\| \leq \|A\| \|B\| \)

Every vector norm generates a corresponding matrix norm, by the formula

\[ \|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|. \]

Thus, we get a matrix 1-norm, 2-norm, and \( \infty \)-norm. As with vector norms, we can assume that all three of them are of the same order of magnitude.

One can show that the matrix 1-norm is the largest column sum:

\[ \|A\|_1 = \max_j \sum_i |a_{ij}| \]

In words: Calculate the vector 1-norm of all columns of \( A \), then take the largest of those.

The matrix \( \infty \)-norm is the largest row sum:

\[ \|A\|_\infty = \max_i \sum_j |a_{ij}| \]

In words: Calculate the vector 1-norm of all rows of \( A \), then take the largest of those.

The matrix 2-norm is the largest singular value. The singular values of \( A \) are the square roots of the eigenvalue of \( A^TA \) (which are all \( \geq 0 \)). This is not something you want to calculate by hand.
Other matrix norms exist which do not come from vector norms. An example is the Frobenius norm, which is
\[ \|A\|_F = \left( \sum_{i,j} |a_{ij}|^2 \right)^{1/2} \]

**Example:** The matrix
\[
\begin{pmatrix}
1 & 2 & 3 \\
2 & -1 & 1 \\
2 & 4 & 0
\end{pmatrix}
\]
has the column sums 5, 7, and 4, so \( \|A\|_1 = 7 \). The row sums are 6, 4, and 6, so \( \|A\|_\infty = 6 \). The Frobenius norm is \( \|A\|_F = \sqrt{40} \approx 6.32 \). The 2-norm is approximately 5.25269.

### 3.1.3. Matrix Equations.

Linear systems of equations of the form
\[
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
\vdots = \vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m
\]
are very common in many scientific fields. They can be written very concisely in the form
\[ Ax = b. \]
where
\[
A = \begin{pmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{pmatrix}, \quad x = \begin{pmatrix}
x_1 \\
\vdots \\
x_n
\end{pmatrix}, \quad b = \begin{pmatrix}
b_1 \\
\vdots \\
b_m
\end{pmatrix}.
\]
This is called a system of \( m \) linear equations in \( n \) unknowns. The \( a_{ij} \) and \( b_i \) are given, the \( x_i \) are the unknowns we want to find.

The basic questions mathematicians always ask when they are faced with an equation are

**Existence:** Is there a solution at all?

**Uniqueness:** If there is a solution, is it unique or are there several, possibly infinitely many?

**Algorithms:** How do you calculate the solution?

For linear equations, the following facts are well-known:

- **If** \( m < n \), the system is called underdetermined. There are fewer equations than unknowns. In general, underdetermined systems have infinitely many solutions. Roughly speaking, if there are five unknowns more than there are equations, you can pick the values of any five of the unknowns arbitrarily, and then solve for the rest. You have “five degrees of freedom” left.
- **If** \( m > n \), the system is called overdetermined. In general, overdetermined systems have no solution.
  
  You can define the concept of least-squares solutions, something we will do in chapter 6.
- **If** \( m = n \), then in general we have a unique solution.

In all these cases, the phrase “in general” refers to the same thing: the assumption that the equations are linearly independent. In the case of a square matrix, this is the same as saying that \( A \) is non-singular.

In this chapter, we will assume that our system is square and non-singular, so that we always have a unique solution. Treatment of the general case will be postponed until chapter 6. We just have to be able to recognize when a matrix is singular.
3.2. Gaussian Elimination

Section 3.2 in NMS is entitled *Linear Systems for Stored Matrices*, but what is really covered is a particular algorithm called *Gaussian Elimination*. This is the most popular algorithm for general matrices, i.e. matrices without any special structure. It is the only algorithm for matrix equations we will consider in this course.

There is a common method of attack for all matrix problems, whether solving a system of equations, finding eigenvalues, finding a generalized inverse, or anything else.

(1) For which types of matrices is the problem easy to solve?
(2) What manipulations can we use to bring a general problem into that form, without changing the solution?

For the case of solving systems of equations, the answer is

(1) Triangular matrices.
(2) The following operations are allowed:
   • Interchanging two equations.
   • Multiplying an equation by a constant.
   • Adding a multiple of one equation to another.

3.2.1. The Basic Algorithm. Consider an example of a triangular system. The system

\[
\begin{align*}
  x_1 - x_2 + 2x_3 &= 2 \\
  2x_2 - x_3 &= 0 \\
  3x_3 &= 6
\end{align*}
\]

can be solved, starting with the last equation. We find \( x_3 = 2 \), substitute this into the next to last equation to find \( x_2 = 1 \), then substitute both into the first equation to find \( x_1 = -1 \).

Any upper triangular system can be solved this way. This process is known as *back substitution*. An analogous process, starting at the top, works for lower triangular systems.

Now, how do we get a general matrix to triangular form, using the allowed operations?

Example:

\[
\begin{align*}
  3x_1 + x_2 + 6x_3 &= 2 \\
  2x_1 + x_2 + 3x_3 &= 7 \\
  x_1 + x_2 + x_3 &= 4
\end{align*}
\]

**Step 1.** Subtract 2/3 of row 1 from row 2. Subtract 1/3 of row 1 from row 3. This produces zeros in the first column. The numbers 2/3, 1/3 are called the *multipliers*.

\[
\begin{align*}
  3x_1 + x_2 + 6x_3 &= 2 \\
  (1/3)x_2 - x_3 &= 17/3 \\
  (2/3)x_2 - x_3 &= 10/3
\end{align*}
\]

Now subtract 2 times row 2 from row 3.

\[
\begin{align*}
  3x_1 + x_2 + 6x_3 &= 2 \\
  (1/3)x_2 - x_3 &= 17/3 \\
  x_3 &= -8
\end{align*}
\]

**Step 2.** Solve by back substitution: \( x_3 = -8, x_2 = -7, x_1 = 19 \). □

This is Gaussian elimination in its simplest form:

• *Forward elimination* to transform the matrix to upper triangular form;
• *Back substitution* to solve the triangular system.
### 3.2.2. Programming Technique

When one wants to program this technique, one usually splits the work up differently:

**Step 1a.** Do the forward elimination on the matrix alone, not on the right-hand side. Save the multipliers for later.

**Step 1b.** Do the forward elimination on the right-hand side, using the stored multipliers.

**Step 2.** Solve by back substitution.

The reason for doing it this way is that it takes no extra computer time and no extra storage, but steps 1b and 2 can now be repeated several times without having to do step 1a over again.

This is useful if you have to solve the same matrix equation with several different right-hand sides. This happens more frequently than you think: even if you only want to solve one equation, estimating the condition number requires solving it twice. There is also a technique known as *iterative improvement* which requires solving the system several times.

In all implementations I am aware of, the work is split up into one subroutine which does step 1a, and another subroutine which does steps 1b and 2. On the disk in your book, these routines are called `SGEFA` and `SGESL`, both of which are called from inside `SGEFS`.

Let us do the previous example over again. We start with the matrix

\[
A = \begin{pmatrix}
3 & 1 & 6 \\
2 & 1 & 3 \\
1 & 1 & 1 \\
\end{pmatrix}.
\]

**Step 1a.** Subtract \((2/3)\) of row 1 from row 2. Subtract \((1/3)\) of row 1 from row 3. This produces zeros in the first column. Store the multipliers.

**Intermediate Result**

\[
\begin{pmatrix}
3 & 1 & 6 \\
2/3 & 1/3 & -1 \\
1/3 & 2/3 & -1 \\
\end{pmatrix}
\]

Note that storage for the multipliers becomes available where we know the zeros to be. We don’t actually have to store the zeros.

Subtract 2 times row 2 from row 3. Store the multiplier.

**Result**

\[
\begin{pmatrix}
3 & 1 & 6 \\
2/3 & 1/3 & -1 \\
1/3 & 2 & -1 \\
\end{pmatrix}
\]

**Step 1b.** Now look at the right-hand side

\[b = \begin{pmatrix} 2 \\ 7 \\ 4 \end{pmatrix}.
\]

Subtract \(2/3\) of \(b_1\) from \(b_2\). Subtract \(1/3\) of \(b_1\) from \(b_3\).

**Intermediate Result**

\[
\begin{pmatrix}
2 \\
17/3 \\
10/3 \\
\end{pmatrix}
\]

Subtract 2 times \(b_2\) from \(b_3\). The result is

**Result**

\[
\begin{pmatrix}
2 \\
17/3 \\
-8 \\
\end{pmatrix}
\]

We are now where we were after forward elimination before.

**Step 2.** Back substitution as before.
3.2.3. Pivoting. Consider two examples.

Example: Consider the system

\[
\begin{align*}
3x_1 & + x_2 + 6x_3 = 1 \\
3x_1 & + x_2 + 3x_3 = 0 \\
x_1 & + x_2 + x_3 = 0
\end{align*}
\]

After one step of elimination, we get

\[
\begin{align*}
3x_1 & + x_2 + 6x_3 = 1 \\
-3x_3 & = -1 \\
(2/3)x_2 & - x_3 = -1/3
\end{align*}
\]

To go on we have to interchange rows 2 and 3:

\[
\begin{align*}
3x_1 & + x_2 + 6x_3 = 1 \\
(2/3)x_2 & - x_3 = -1/3 \\
-3x_3 & = -1
\end{align*}
\]

and solve: \(x_1 = -1/3, \ x_2 = 0, \ x_3 = 1/3\).

Example: Consider the system

\[
\begin{align*}
0.005x_1 & + x_2 = 0.5 \\
x_1 & + x_2 = 1
\end{align*}
\]

The exact solution is \(x_1 = 0.502512 \ldots, \ x_2 = 0.49748 \ldots\). If we use Gaussian elimination on a hypothetical computer with three decimals accuracy, we get

\[
\begin{align*}
0.005x_1 & + x_2 = 0.5 \\
-199x_2 & = -99
\end{align*}
\]

\[
\begin{align*}
x_2 & = -99 \\
199 & = 0.497 \quad \text{(rounded to three digits)}
\end{align*}
\]

\[
\begin{align*}
x_1 & = 0.5 - 0.497 \\
0.005 & = 0.003 \\
0.005 & = 0.600
\end{align*}
\]

This is bad accuracy.

If we interchange the equations and start with

\[
\begin{align*}
x_1 & + x_2 = 1 \\
0.005x_1 & + x_2 = 0.5
\end{align*}
\]

we get

\[
\begin{align*}
x_1 & + x_2 = 1 \\
0.995x_2 & = 0.495
\end{align*}
\]

and therefore

\[
\begin{align*}
x_2 & = \frac{0.495}{0.995} = 0.497, \\
x_1 & = 1 - x_2 = 0.503.
\end{align*}
\]

This is perfect accuracy with three digits.

These examples are intended to motivate the necessity of pivoting. In Gaussian elimination, the diagonal elements used to eliminate the entries below them are called pivots. Pivoting is the process by which we interchange rows to bring different elements to this position. The method by which we select the next pivot is called the pivoting strategy.

What you should have learned from the above examples is

- Pivoting is absolutely necessary, namely when a zero shows up in the pivot position (first example).
- Correct pivoting can increase the accuracy of the result (second example).
It can be shown in general that numerical difficulties are associated with small pivots. The larger the pivots, the better the accuracy.

The main pivoting strategy employed in practice is partial pivoting, where we select the largest element (in absolute value) in the current column. This will automatically take care of zero pivots (example 1) and also handle example 2 correctly. As mentioned above, we want to have large numbers in pivoting position. The routine \texttt{SGEFS} in your textbook uses this strategy.

Partial pivoting does an adequate job for most equations. Other pivoting strategies are available, but they require extra work. Also, if your matrix is so badly conditioned that partial pivoting is not good enough, you should be asking yourself whether you are solving a useful problem.

3.2.4. Singular Matrices. How do you notice if a matrix is singular? In theory, you would find only zeros as your potential pivots at some point. Any decent Gaussian elimination program has to be prepared for this possibility and should signal a singular matrix.

In practice, round-off errors will perturb these exact zeros into small numbers (near the machine epsilon). It is very hard for a program to decide whether these small numbers are legitimate or are an indication of a singular matrix.

The approach that is usually taken is to estimate the condition number (see below). A singular matrix has an infinite condition number. When it gets perturbed a little by round-off error, it will probably end up being a matrix with a very large condition number ($10^6$ or more). A large condition number is a warning signal that the matrix is very close to singular. It is up to the user to decide what is acceptable and what is not.

For practical purposes, there is really no difference between a singular matrix and an almost singular matrix. If the matrix is singular, you cannot find a solution. If the matrix is almost singular, the condition number is so large that you cannot trust any digits in your answer. The effect is the same.

Example: Consider the matrix

$$\begin{pmatrix} 2 & 3 \\ 0.2 & 0.3 \end{pmatrix}$$

This matrix is singular from a mathematical point of view. On a computer, the 0.2 and 0.3 get rounded a little, so that the matrix that is actually stored in the machine is not singular. You cannot solve a system with such a matrix and expect meaningful answers.

3.2.5. The Complete Algorithm. Putting all the previous strategies together, here is the final algorithm the way it is programmed.

First subroutine:

- For each of columns 1, 2, \ldots, $(n - 1)$
  - Select a pivot by some strategy; signal an error if you cannot find a nonzero pivot.
  - Interchange rows to bring the pivot to pivoting position and save the necessary bookkeeping information.
  - Use the pivot to eliminate the entries below it. Store the multipliers in these spaces.

Second subroutine:

- Do the forward elimination on the right-hand side, using the stored multipliers and the bookkeeping information from pivoting, in the same order in which they were applied to the left-hand side (i.e. column by column)
- Do the back substitution to find the solution.
- Estimate the condition number (desirable step).

3.2.6. Operation Count. To determine the computational effort, one usually counts multiplications and divisions. The assumption is that everything else (program overhead, additions, subtractions) is negligible compared to the computer time spent on multiplication/division. This is not as true as it used to be, but it is still a pretty good first approximation.

For the forward elimination on the matrix, we have the following count:

- To get zeros in the first column, we have one division per row to determine the multiplier, followed by $(n - 1)$ multiplications per row to subtract this multiple of row 1 from the rest of the row. That makes $n$ operations per row. We are doing this on $(n - 1)$ rows. Altogether we have $(n - 1)n$ operations.
To get zeros in the second column, we get \((n - 2)(n - 1)\), since there is one row less, and each row is one entry shorter.

Altogether we get

$$\text{total operations} = \sum_{i=1}^{n-1} i(i + 1) \approx n^3/3$$

for forward elimination.

To apply the multipliers to the right-hand side, we have as many multiplications as multipliers, namely

\[(n - 1) + (n - 2) + \cdots + 2 + 1 = n(n - 1)/2.\]

For the back substitution, it takes one division for \(x_n\), one multiplication and one division for \(x_{n-1}\), two multiplications and one division for \(x_{n-2}\), and so on. Together

\[1 + 2 + \cdots + n = n(n + 1)/2.\]

The total of these two is exactly \(n^2\).

Thus: It takes about \(n^3/3\) operations to reduce the matrix to triangular form, plus \(n^2\) operations for each right-hand side. There is a factor of \(n/3\) difference, so for a large matrix most of the work is in the first part. The extra work done by solving the system with a different right-hand side is negligible.

### 3.2.7. LU Decomposition

Gaussian elimination is often called Gaussian decomposition or LU decomposition or LU factorization. The reason for that is the following.

Consider again the matrix

\[
\begin{pmatrix}
3 & 1 & 6 \\
2/3 & 1/3 & -1 \\
1/3 & 2 & 1
\end{pmatrix},
\]

which was the result of applying Gaussian elimination to the matrix

\[
A = \begin{pmatrix}
3 & 1 & 6 \\
2 & 1 & 3 \\
1 & 1 & 1
\end{pmatrix}.
\]

Take the multipliers below the diagonal and add ones on the diagonal:

\[
L = \begin{pmatrix}
1 & 0 & 0 \\
2/3 & 1 & 0 \\
1/3 & 2 & 1
\end{pmatrix}.
\]

Take the numbers above the diagonal, that is, the triangular matrix that \(A\) was transformed into:

\[
U = \begin{pmatrix}
3 & 1 & 6 \\
0 & 1/3 & -1 \\
0 & 0 & 1
\end{pmatrix}.
\]

Then it turns out that

\[
A = LU.
\]

We have factored (decomposed) the matrix \(A\) into a product of a lower triangular matrix \(L\) and an upper triangular matrix \(U\).

Pivoting messes up this scheme a little, but we don’t need to go into that.

### 3.2.8. Calculating the Determinant

After you have factored the matrix into an upper and a lower triangular one (ignoring the pivoting)

\[
A = LU,
\]

you get the determinant basically for free: The determinant of a triangular matrix is the product of the numbers on the diagonal, so

\[
\det(A) = \det(L)\det(U) = \det(U) = u_{11} \cdots u_{nn}
\]

(remember, \(L\) had only ones on the diagonal). If you did an odd number of interchanges during pivoting, you need to multiply this by \((-1)\).
3.2.9. **Calculating** $A^{-1}$. Most of the time, you really don’t need $A^{-1}$. You may think you do, but you really don’t. What you really want is to calculate $A^{-1}x$ or $A^{-1}B$.

Suppose you want to calculate

$$X = A^{-1}B,$$

where $B$ is some given matrix. This is equivalent to solving the matrix equation

$$AX = B,$$

which you do by solving

$$Ax_1 = b_1$$
$$Ax_2 = b_2$$
$$\ldots$$

Here $b_1$ is the first column of $B$, $b_2$ is the second column of $B$, and so on. Solving the first equations gives you $x_1$, which is the first column of $X$, and so on.

Count the operations: If $A$, $B$, $X$ are all square $n \times n$ matrices, it takes $(n^3/3)$ operations to factor $A$ and $n^2$ operations for each of the $n$ right-hand sides, for a total of $(4/3)n^3$ operations.

If you calculate $A^{-1}$ first, that will take you $(2/3)n^3$ operations (you don’t know that, but I do), plus $n^3$ operations to multiply $A^{-1}$ and $B$, for a total of $(5/3)n^3$ operations. This does not even take into account that you get more round-off error this way.

Now, if you are absolutely convinced that you do need $A^{-1}$, you can find it by solving the matrix equation

$$AA^{-1} = I$$

as outlined above. That is, you solve

$$Ax = e_1$$

where $e_1$ is the first basis vector. This will give you the first column of $A^{-1}$. Then solve

$$Ax = e_2$$

to find the second column of $A^{-1}$, and so on.

You can streamline this by taking advantage of the known zero entries of $e_i$, which is why it only takes $(2/3)n^3$ operations instead of $(4/3)n^3$ operations, but in principle this is how to do it.

3.3. **Subroutine SGEFS**

Read this on your own.

3.4. **Historical Perspective**

Read this on your own.

3.5. **Column-Oriented Algorithms**

You can skip this section.
3.6. The Condition Number

How do we measure the error on solving a system of linear equations? Linear equations are already discrete, so the only source of error is round-off.

The error can be measured in a couple of different ways. What we are usually interested in is the error in the solution, that is either

$$\text{absolute error} = \|x - x^*\|,$$

where $x$ is the true solution, $x^*$ the numerical solution, and we use one of the norms we have, or

$$\text{relative error} = \frac{\|x - x^*\|}{\|x\|}.$$

This error can be estimated from the condition number discussed below:

$$\text{relative error in the solution} \approx \text{condition number} \cdot \text{machine epsilon}.$$  

Another possibility is to measure how well our numerical solution $x^*$ solves the equation. For this, we define the residual

$$r = Ax^* - b.$$  

The residual is the difference between the true right-hand side of the equation, and the right-hand side we get when we plug the numerical solution back into the equation.

We won’t discuss details here, but it can be shown that the residual is always small if the equation has been solved correctly (i.e. with correct pivoting). The error can be quite large, for a badly behaved matrix, but the “incorrect” result will still almost solve the equation. You will see an example of this in chapter 4.

The definition of the condition number given in NMS appears to be different from the following, but the two definitions are in fact exactly the same. I am just taking a slightly different approach.

Recall that in general

$$\text{cond} = \max \frac{\text{relative error in output}}{\text{relative error in input}}$$

In solving a linear system

$$Ax = b,$$

the “output” is clearly $x$. The “input” could be $b$, $A$ or both.

If we assume that $b$ is the input, we can calculate the condition number with respect to changes in the right-hand side as follows. Solve two systems

$$Ax = b \quad \text{and} \quad Ax^* = b^*,$$

where $b$ and $b^*$ are pretty close. Then we have

$$\text{relative error in input} = \frac{\|b - b^*\|}{\|b\|} \quad \text{relative error in output} = \frac{\|x - x^*\|}{\|x\|}.$$  

Likewise, if we assume $A$ is the input, we solve the two systems

$$Ax = b \quad \text{and} \quad A^*x^* = b^*$$

where $A$ and $A^*$ are not too different. Then

$$\text{relative error in input} = \frac{\|A - A^*\|}{\|A\|} \quad \text{relative error in output} = \frac{\|x - x^*\|}{\|x\|}.$$  

The values we get depend of course on the choice of norm, but all of the standard norms give the same order of magnitude. This is really all we need. It turns out that the conditions numbers for changes in the right-hand side and for changes in the matrix are the same.
Both of them are equal to
\[ \text{cond} \approx \| A \| \| A^{-1} \|. \]

\( \| A \| \) is easy to estimate, in either the 1-norm or \( \infty \)-norm. \( \| A^{-1} \| \) is not so easy to get. SGEFS uses the fact that
\[ \| A^{-1} \| = \max_{x \neq 0} \frac{\| A^{-1} x \|}{\| x \|} \approx \frac{\| A^{-1} y \|}{\| y \|} \]
for a suitably chosen \( y \) close to the worst case. Read section 3.8 in NMS to see how this \( y \) is chosen.

Note that the condition number can never be smaller than 1. (Why?). However, it can be equal to one (see chapter 6).

What you need to remember about the condition number is this: A small condition number means that the equation is well-conditioned, and you can trust your calculated answer. A large condition number means that you cannot trust some of the digits. For example, a condition number of \( 10^4 \) means that the rightmost four digits of the answer are suspect. If you work in seven digit accuracy, you can only trust three digits. A very large condition number indicates an almost singular matrix. Your calculated answer may be worthless.

You can try to go to double precision in some cases, and there are other ways to deal with this problem (see chapter 6), but you should ask yourself if you are really calculating something useful here.

Most good Gaussian elimination routines return an estimate of the condition number. Your main program should inspect that. If it is very large, this indicates an almost singular matrix. As I said before, there is very little difference in practice between a singular matrix and an almost singular matrix.

Actually, what SGEFS returns is the reciprocal of the condition number. That is because the condition number might be infinite, but the reciprocal is always finite.
CHAPTER 4

Interpolation

4.1. Introduction

We will cover sections 4.1 through 4.12 in the book. Read section 4.1 in the book on your own.

The basic problem of one-dimensional interpolation is this:

Given a set of data points \( x_i, \ i = 0, \ldots, m \), given function values \( y_i \), and given a class of admissible functions, find a function \( f \) in this class so that the graph of \( f \) goes through these points:

\[
f(x_i) = y_i.
\]

More generally, we could prescribe not just function values, but also derivative values. An example is so-called Hermite interpolation, where we demand

\[
f(x_i) = y_i,\]

\[
f'(x_i) = y'_i.
\]

Depending on the class of admissible functions, we get different kinds of interpolation. The classes most frequently used in practice are

- **Polynomials** of a certain degree \( n \), so that

\[
f(x) = \sum_{k=0}^{n} c_k x^k.
\]

- **Piecewise Polynomials** or **Splines**. We will talk about these later.

- **Rational Functions**, that is, quotients of polynomials.

\[
f(x) = \frac{p(x)}{q(x)} = \frac{\sum_{i=0}^{m} c_i x^i}{\sum_{j=0}^{n} d_j x^j}
\]

- **Trigonometric Polynomials**, which are defined as

\[
f(x) = \sum_{k=0}^{n} a_k \cos(kx) + \sum_{k=1}^{n} b_k \sin(kx)
\]

This type of interpolation is related to Fourier series.

We will only consider polynomial and piecewise polynomial interpolation in this course.

A problem related to interpolation is the problem of approximation. Again, we are given a set of data points and a class of admissible functions. However, we do not require that the approximating function actually goes through these points. Rather, we are looking for an admissible function which gets “as close as possible” to these points. The exact meaning of “as close as possible” must be specified.

A typical example of approximation is trying to put a straight line through 10 data points. We will see some examples of this in chapter 6. All classes of functions used in interpolation can also be used in approximation.
Most interpolation problems are linear. This means that the class of admissible functions consists of linear combinations of some basis functions:
\[ f(x) = \sum_{k=0}^{n} \alpha_k b_k(x). \]

The set of basis functions is not unique. For polynomial interpolation, one possible basis is \( b_k(x) = x^k \). Other possible bases are the Lagrange polynomials (see below) or the polynomials used for Newton interpolation. In trigonometric interpolation, the basis functions are sines and cosines. One possible basis for splines is the set of B-splines defined in section 4.14 (we skip that section).

Rational interpolation is an example of nonlinear interpolation. (Note that when you add two rational functions, the sum will usually have higher degree polynomials in numerator and denominator).

The general questions for any type of interpolation (and many other types of problems, too), are

- **Existence**: Is there a solution?
- **Uniqueness**: Is the solution unique?
- **Algorithms**: How do you find the solutions in practice?

### 4.1.1. Solving Linear Interpolation Problems

There are two simple ways to attack linear interpolation problems. They are not always good ways to do it, but they often work. If you are faced with a linear interpolation problem with an unusual set of basis functions, you may want to start with one of these.

**The Matrix Approach**. Since we are solving a linear problem, any admissible function is of the form
\[ f(x) = \sum_{k=0}^{n} \alpha_k b_k(x), \]

where the \( b_k \) are known basis functions, and the \( \alpha_k \) unknown coefficients to be determined. Suppose we want to satisfy
\[ f(x_i) = y_i \quad \text{for } i = 0, \ldots, m. \]

This leads to a linear system of equations
\[
\begin{align*}
    b_0(x_0)\alpha_0 + \cdots + b_n(x_0)\alpha_n &= y_0 \\
    b_0(x_1)\alpha_0 + \cdots + b_n(x_1)\alpha_n &= y_1 \\
    &\vdots \\
    b_0(x_m)\alpha_0 + \cdots + b_n(x_m)\alpha_n &= y_m
\end{align*}
\]

The coefficients \( \alpha_j \) are the unknowns, the rest is known. If we have derivative information, we get equations containing \( b'_k(x_j) \) or higher derivatives. These are also known, since we know the \( b_k \).

This approach reduces the interpolation problem to a matrix problem. If \( m = n \), we can hope to find a unique solution.

**The Basis Function Approach**. Suppose we could find basis functions \( B_i(x) \) with the special property
\[ B_i(x_j) = \delta_{ij}. \]

The \( \delta_{ij} \) is called the Kronecker delta and is defined as
\[
\delta_{ij} = \begin{cases} 
1 & \text{if } i = j, \\
0 & \text{otherwise.}
\end{cases}
\]

Thus, \( B_i(x) \) has the value 1 at \( x_i \), and value 0 at the other \( x_j \). The function
\[ f(x) = \sum_{k=0}^{m} y_k B_k(x) \]

then has the required property
\[ f(x_j) = y_j. \]

All we need to do is to find the \( B_i(x) \) once and for all, which is often possible.
This approach can be modified if derivative information is given. For example, for Hermite interpolation you use the basis functions $B_i$ and $\hat{B}_i$ which satisfy

$$B_i(x_j) = \delta_{ij},$$
$$B'_i(x_j) = 0,$$
$$\hat{B}_i(x_j) = 0,$$
$$\hat{B}'_i(x_j) = \delta_{ij}.$$ 

4.2. Polynomial Interpolation

Read this section in the book on your own. I have combined the material with the following section.

4.3. Using Other Basis Functions

I will cover the book sections 4.2 and 4.3 here. The book covers only the two approaches I mentioned in the introduction. I will also cover Newton’s approach to solving the polynomial interpolation problem, which is not in the book. In my opinion, Newton’s approach is preferable to the other two.

Before we look at algorithms, let us answer the existence/uniqueness question.

**Fact:** Assume we are given a sequence of distinct interpolation points $x_0, x_1, \ldots x_m$. At the point $x_j$, we are given $n_j$ conditions

$$f(x_j) = y_j,$$
$$f'(x_j) = y'_j,$$
$$\ldots$$
$$f^{(n_j-1)}(x_j) = y_{(n_j-1)}^j,$$

for a total of $N = n_0 + n_1 + \ldots n_m$ conditions. Then there is a unique interpolating polynomial of degree $N - 1$ which satisfies these conditions.

This fact covers both standard interpolation ($n_j = 1$ for all $j$) and Hermite interpolation ($n_j = 2$ for all $j$).

The polynomial can be found using either one of the two approaches mentioned in the introduction.

**Remark:** I would like to emphasize two points that may not be obvious.

First of all, the theorem mentioned above states that **all possible methods produce exactly the same polynomial, except for roundoff error.** This polynomial can be written in a number of different ways, but there is only one. There is no reason to favor one approach over the other because it produces a better or worse interpolating polynomial.

Second, none of the methods treated here requires that the points $x_j$ are in order. This is the way they usually come, but all three algorithms can handle points in any order.

From now on, we only consider the standard polynomial interpolation problem. We assume we have $(n + 1)$ distinct points $x_0, \ldots, x_n$, in any order. We are given $y_0, \ldots, y_n$, but no derivative values. We want to interpolate with a polynomial of degree $n$ (which has $(n + 1)$ coefficients). We know that there is a unique solution.

4.3.1. The Matrix Approach. We use any convenient basis for the set of polynomials. Usually, this means $b_j(x) = x^j$ or some shifted version $b_j(x) = (x - x_0)^j$.

**Example:** Put a polynomial through the points $(-1, 1), (1, 1), (3, 2)$ and $(5, 3)$.

We have 4 conditions, so we use a cubic polynomial (with 4 coefficients). We set up

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

and get the equations

$$c_0 - c_1 + c_2 - c_3 = 1$$
$$c_0 + c_1 + c_2 + c_3 = 1$$
$$c_0 + 3c_1 + 9c_2 + 27c_3 = 2$$
$$c_0 + 5c_1 + 25c_2 + 125c_3 = 3$$
The solution is
\[ p(x) = (39 + x + 9x^2 - x^3)/48. \]

4.3.2. The Basis Function Approach. The basis functions with the property \( B_i(x_j) = \delta_{ij} \) for standard polynomial interpolation are called Lagrange polynomials.

They have an explicit representation
\[ L_{n,k}(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)} \]

where \( k \) is between 0 and \( n \). You can check by inspection that the \( L_{n,k} \) have the required properties

- Each \( L_{n,k} \) is a polynomial of degree \( n \). (Notice there are \( n \) factors in the numerator, each with one \( x \), and only constants in the denominator).
- \( L_{n,k} \) has the value 0 at all the \( x_i \) except at \( x_k \), where it has the value 1.

Note that the \( L_{n,k} \) depend on the points \( x_j \), so they have to be calculated from scratch for every interpolation problem.

Example: With the same numbers as in the last example, we get
\[
L_{3,0}(x) = \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} = \frac{(x - 1)(x - 3)(x - 5)}{(-1 - 1)(-1 - 3)(-1 - 5)} = \frac{(15 - 23x + 9x^2 - x^3)/48,}{L_{3,1}(x) = (15 + 7x - 7x^2 + x^3)/16,}
\]
\[
L_{3,2}(x) = (-5 + x + 5x^2 - x^3)/16, \quad L_{3,3}(x) = (3 - x - 3x^2 + x^3)/48
\]

and our solution is
\[ p(x) = 1 \cdot L_{3,0} + 1 \cdot L_{3,1} + 2 \cdot L_{3,2} + 3 \cdot L_{3,3} = (39 + x + 9x^2 - x^3)/48, \]
like before.

4.3.3. Newton’s Approach. Newton’s method of finding the interpolating polynomial is better than either one of the methods mentioned above, in my opinion. I will explain the reasons later, when I compare the methods.

Instead of calculating the interpolating polynomial in one fell swoop, Newton does it in stages. Define \( p_k(x) \) to be the polynomial of degree \( k \) which interpolates the data at \( x_0, x_1, \ldots, x_k \), and build up \( p_0, p_1 \), and so on. Once you get to \( p_n \), you are done.

A suitable representation is the following:

\[
p_0(x) = b_0 \\
p_1(x) = p_0(x) + b_1(x - x_0) \\
p_2(x) = p_1(x) + b_2(x - x_0)(x - x_1) \\
\ldots
\]
\[
p_n(x) = p_{n-1}(x) + b_n(x - x_0) \cdots (x - x_{n-1})
\]

It is obvious that \( p_k \) is a polynomial of degree \( k \). The \( b_j \) are coefficients to be determined. We need to find suitable values so that \( p_k \) interpolates the data at \( x_0 \) through \( x_k \).

We do that by induction:

\( p_0 \) is a polynomial of degree 0, i.e. a constant. We need to determine \( b_0 \) so that \( p_0(x_0) = y_0 \). The obvious solution is \( b_0 = y_0 \).
When we determine \( p_k \), we already know that \( p_{k-1} \) interpolates at \( x_0 \) through \( x_{k-1} \) (induction hypothesis). Since \( p_k = p_{k-1} + ( \text{a term that vanishes at } x_0 \text{ through } x_{k-1} ) \), \( p_k \) also automatically interpolates at \( x_0 \) through \( x_{k-1} \). The remaining condition \( p_k(x_k) = y_k \) can be satisfied by a suitable choice of \( b_k \).

It is easy to verify that

\[
\begin{align*}
b_0 &= y_0, \\
b_1 &= \frac{y_1 - y_0}{x_1 - x_0},
\end{align*}
\]

but after that it gets pretty messy.

Now, let us take a detour. Suppose we are given some points \( x_0 \) through \( x_n \), and function values \( y_0 \) through \( y_n \). We define divided differences recursively as follows:

**Zeroth Divided Difference:** \( y[x_j] = y_j \).

**First Divided Difference:** \( y[x_j, x_{j+1}] = \frac{y[x_{j+1}] - y[x_j]}{x_{j+1} - x_j} \).

**Second Divided Difference:** \( y[x_j, x_{j+1}, x_{j+2}] = \frac{y[x_{j+1}, x_{j+2}] - y[x_j, x_{j+1}]}{x_{j+2} - x_j} \).

In general,

\[
y[x_j, \ldots, x_{j+k}] = \frac{y[x_{j+1}, \ldots, x_{j+k}] - y[x_j, \ldots, x_{j+k-1}]}{x_{j+k} - x_j}.
\]

These divided differences can be calculated very rapidly in a triangular table:

\[
\begin{array}{c|ccc}
x_0 & y[x_0] & y[x_0, x_1] & y[x_0, x_1, x_2] \\
x_1 & y[x_1] & y[x_1, x_2] & y[x_1, x_2, x_3] \\
x_2 & y[x_2] & y[x_1, x_2, x_3] \\
x_3 & y[x_3] & &
\end{array}
\]

Each entry is calculated as the difference of its left neighbors, divided by the difference of the corresponding \( x \) values.

So what does that have to do with anything? It turns out that

\[
b_k = y[x_0, x_1, \ldots, x_k],
\]

so that the top row in this triangular table contains exactly the correct Newton coefficients.

**Example:** Same example as before. We get

\[
\begin{array}{c|cccc}
-1 & 1 & 0 & 1/8 & -1/48 \\
1 & 1 & 1/2 & 0 & 1/2 \\
3 & 2 & & &
\end{array}
\]

Therefore, Newton’s solution is

\[
p(x) = 1 + 0 \cdot (x + 1) + \frac{1}{8}(x + 1)(x - 1) - \frac{1}{48}(x + 1)(x - 1)(x - 3).
\]

If you multiply it out, you get the same polynomial as before. \( \square \)
4.3.4. Comparison of Methods. Let me repeat the point I made before: *All methods produce exactly the same polynomial*, except for roundoff error. The only difference between the methods is in efficiency (less work) and stability (less round-off error).

The matrix approach is easy to understand, but that is where its advantages end. It is not very efficient, and with increasing number of points, it rapidly becomes unstable.

The Lagrange basis function approach is very stable (the most stable of all methods, as a matter of fact), but again not very efficient.

The Newton approach is much faster than the other two, it is pretty stable, and it has one other advantage: it lets you add extra points to the existing ones without starting over again. You simply add another line or several to the divided difference table. This is useful if you don’t know how many points you need.

4.4. How Good is Polynomial Interpolation?

If the interpolation data comes out of the blue, there is no point in talking about error, since there is no “true solution” to compare with the result.

Let us therefore assume that the data values $y_j$ are values of a known function $g(x)$ at the points $x_j$, $j = 0, \ldots, n$. The *interpolation error* is the difference between the original function and the interpolating polynomial (ignoring roundoff error). How large can the interpolation error get?

There are two questions I want to address here:

- For a fixed set of points $x_j$, how large is the error?
- What happens as we increase the number of points more and more?

4.4.1. Fixed Interpolation Points. Assume the points $x_j$ are fixed. They may be table values of a function $g(x)$, or measured values.

If $g$ has $(n + 1)$ continuous derivatives, then it can be shown that

$$g(x) - p(x) = \frac{\omega(x)}{(n + 1)!} g^{(n+1)}(\xi),$$

where

$$\omega(x) = (x - x_0)(x - x_1)\ldots(x - x_n),$$

and $\xi$ is some point between $x_0$ and $x_n$.

If $g(x)$ is known, we can estimate the error, at least in principle.

One thing to observe is that as long as $x$ is somewhere among the $x_j$, $\omega(x)$ is not too bad. If we move $x$ around, some terms $x - x_j$ will get larger, others smaller. However, if we move $x$ outside, $\omega(x)$ will grow very rapidly (all terms get larger). *Polynomial interpolation becomes extremely unreliable if we try to extrapolate into an area where we don't have any data points.*

4.4.2. Increasing the Number of Points. We can see that if we keep the function $g(x)$ fixed and increase the number of interpolation points, the error estimate involves higher and higher derivatives of $g$. Even for nice functions $g$, higher derivatives may become large. Look at the Runge example in the book for a bad case.

How should we choose the points to keep the error growth under control?

Using equally spaced points is pretty bad. Using a different type of error estimate than shown here, one can show that the worst case error grows exponentially in $n$.

If one chooses the so-called Chebyshev points, the error grows only like $\log(n)$, which is much better. On the interval $[-1, 1]$, the Chebyshev points are given by

$$x_j = \cos\left(\frac{2j - 1}{2n} \pi\right), \quad j = 1, \ldots, n.$$ 

For other intervals, these points need to be scaled and shifted. (Note that the numbering is from 1 to $n$ here, instead of from 0 to $n$). The Chebyshev points are not the best possible, but they are close to the best possible, and they are easy to find.

*If you are using more than 4 or 5 points, you can’t expect good results from equally spaced points. Use the Chebyshev points.*

**Summary:** Even in the best case, we have to assume that with increasing $n$ the error in polynomial interpolation will grow. This is a basic shortcoming of polynomial interpolation.
4.5. Historical Perspective

Read this section on your own.

4.6. Evaluation of Polynomials

How do you evaluate
\[ p(x) = a_n x^n + \cdots + a_1 x + a_0 \]
on a computer, if the \( a_j \) and \( x \) are known?

The direct “multiply it out” approach is not very efficient. Even if you are smart enough to calculate \( x^j \) as \( x^j - 1 \ast x \), instead of from scratch, it takes \((n - 1)\) multiplications to generate all powers of \( x \), and another \( n \) multiplications to multiply them by the coefficients, for a total of \((2n - 1)\).

The best known method is Horner’s Rule, which is based on writing
\[ p(x) = ((...((a_n x + a_{n-1})x + a_{n-2})x \cdots)x + a_1)x + a_0. \]

This takes only \( n \) multiplications.

With a slight multiplication, the same thing works for Newton’s approach:
\[ p(x) = ((...((b_n (x - x_{n-1} + b_{n-1})(x - x_{n-2}) + \cdots)(x - x_0) + b_0. \]

Altogether, Newton’s approach takes about \( n^2/2 \) divisions to generate the divided difference table, and then \( n \) multiplications per evaluation. The interpolating polynomial itself is never written down.

Here are two little subroutines I wrote to calculate Newton’s interpolating polynomial. They might come in handy for a homework assignment. I will put these routines in /home/mathclasses/keinert/473 on PV and in the class: directory on Vax.

```fortran
subroutine ncoef(x,y,n)
    *
    * calculate coefficients for Newton interpolating polynomial
    * for interpolating x1..xn, y1..yn.
    * Results are returned in y.
    *
    integer n, i, j
    real x(n), y(n)
*
    do i = 2, n
        do j = n, i, -1
            y(j) = (y(j) - y(j-1)) / (x(j) - x(j-i+1))
        enddo
    enddo
end

real function neval(x,b,n,x0)
    *
    * evaluate the Newton polynomial given by x1..xn,
    * coefficients b1..bn, at the point x0
    *
    integer n, i
    real x(n), b(n), x0
 *
    neval = b(n)
    do i = n-1, 1, -1
        neval = neval * (x0 - x(i)) + b(i)
    enddo
end
```
To use them, you have to call `ncoef` once, which will construct the divided difference table. It turns out you can do that “in place”, so that the table itself is never stored, just its top row, which is all you need. When you enter `ncoef`, array `y` contains the function values. On return, it contains the coefficients $b_j$.

The second routine `neval` evaluates the polynomial at a point $x_0$, using the modified Horner scheme described above.

### 4.7. Piecewise Linear Interpolation

Read section 4.7 in the book.

Assume we are given an increasing sequence of points $x_j$, $j = 0, \ldots, n$, called the knots. A piecewise polynomial is a function which reduces to a polynomial on each subinterval $[x_j, x_{j+1}]$. Different subintervals can contain different polynomials. Usually we are only interested in piecewise polynomials which reduce to polynomials of the same degree $k$ on every subinterval.

**Meta-theorem:** If the function $g(x)$ has $(k + 1)$ continuous derivatives on $[a, b]$, if we divide $[a, b]$ into $n$ equally spaced subintervals of width $h = (b-a)/n$, and interpolate $g$ at the knots by a piecewise polynomial $s(x)$ of degree $k$, then the error satisfies
\[
|s(x) - g(x)| \leq C h^{k+1} |g^{(k+1)}(\xi)|,
\]
where $C$ is some constant (depending on the type of piecewise polynomials we use) and $\xi$ is some point in $(a, b)$.

By “meta-theorem” I mean that the statement is much too vague to be a theorem. It is more a “rule of thumb”, standing for a collection of theorems, one for each type of piecewise polynomials. For each type of piecewise polynomials, you have to prove it from scratch, producing a theorem for this particular case.

One way to remember the meta-theorem and to make it plausible is to observe that this looks remarkably like the remainder term of a Taylor series. If I used a Taylor polynomial of order $k$ on each subinterval, I would get this error estimate with $C = 1/(k + 1)$!

The important thing to observe is that as $n \to \infty$, the error goes to zero, since $h$ goes to zero, and the rest stays bounded. This is the big advantage piecewise polynomials have over polynomials.

### 4.8. Piecewise Cubic Functions

I will cover book sections 4.8, 4.10 and 4.11 in this section. Read sections 4.8 and 4.11 in the book on your own. Reading 4.10 is optional.

As you probably guessed already, a piecewise cubic function is a piecewise polynomial which reduces to a cubic polynomial in each subinterval. Two important types are Hermite cubics and cubic splines.

In piecewise polynomial interpolation, it is important that the points are in order: $x_0 \leq x_1 \leq \ldots \leq x_n$.

We will stick to the standard problem (function values given, but no derivatives).

**4.8.1. Hermite Cubics.** A Hermite cubic is a piecewise cubic polynomial with one continuous derivative. Thus, at every interior point where two cubic polynomials come together (at each knot), we demand that the function values and the first derivative fit together.

Suppose we are given data points $(x_j, y_j)$, $j = 0, \ldots, n$ to interpolate. We have $n$ subintervals and a cubic polynomial on each. That makes $4n$ coefficients to determine.

We have $2n$ constraints from the interpolation data (note that the interior points put constraints on two polynomials). The other $2n$ constraints come from prescribing derivative values $y'_j$. (In case you haven’t noticed it yet: the word “Hermite” in the context of interpolation usually means you are given function values and derivatives). Prescribing derivatives automatically makes sure that the derivatives fit together at the knots.

If you know the derivatives, that’s fine. What if you don’t? In that case, you can estimate the derivatives from the $y_j$. To estimate $y'_j$, you put a parabola through the interpolation points $(j - 1)$, $j$ and $(j + 1)$, and differentiate that. Details are on page 103. I have seen this process called Bessel interpolation.

So now we have data $y_j$ and $y'_j$. Since we have $4n$ constraints and $4n$ coefficients to determine, we expect to find a unique solution.

One possible method of attack would be to create a system of $4n$ equations in $4n$ unknowns (the “matrix method”). A better method is the “basis function approach” mentioned earlier. Details for that are given in section 4.10 in the book.
We will skip the details.

4.8.2. Cubic Splines. A cubic spline is a piecewise cubic polynomial with two continuous derivatives. Thus, at every interior point where two cubic polynomials come together, we demand that the function values and the first and second derivative fit together.

Remark: We can’t require that more than two derivatives fit together: if the function values and three derivatives of two cubic polynomials fit together, they are identical polynomials.

Remark: The reason for the name spline is the following: A spline originally was a flexible metal rod used as a drafting tool. In order to produce a smooth curve passing through a set of points, the draftsman would pin the rod down on the paper at these points and trace the shape of the rod.

The rod has a tendency to assume the shape which minimizes the internal energy. The energy is related to the curvature, and the curvature is related to the second derivative. Approximately, the rod would take on a shape which minimizes the average second derivative. The splines we consider here have the same property: they minimize the average second derivative. This is where they got their name.

The more derivatives a function has, the smoother it is. Hermite polynomials had one continuous derivative. We expect cubic splines to look smoother.

Suppose again that we are given data points \((x_j, y_j), j = 0, \ldots, n\) to interpolate. We have \(n\) subintervals and a cubic polynomial on each. That makes \(4n\) coefficients to determine.

Specifying the values \(y_i\) gives us a total of \(2n\) conditions, like before. Matching first derivatives left and right at the interior points gives \((n−1)\) conditions; likewise for the second derivatives. The total is \((4n−2)\) conditions, so we need two extra conditions.

There are many ways to get these conditions, and each one leads to a different kind of spline.

- Set the second derivative equal to zero at the endpoints:
  
  \[
  s''(x_0) = 0 \\
  s''(x_n) = 0
  \]

  This is the so-called natural or free spline. This is the type of curve the draftsman got out of his metal spline by letting the ends stick out free.

- If we prescribe the derivatives at the endpoints
  
  \[
  s'(x_0) = y'_0 \\
  s'(x_n) = y'_n
  \]

  we get the clamped or complete spline. Physically, this corresponds to clamping the ends of the metal spline into a certain position.

  If we don’t know what the derivatives are, we could estimate them. This is the same idea as with Hermite interpolation.

- We could prescribe the second derivatives at the endpoints
  
  \[
  s''(x_0) = y''_0 \\
  s''(x_n) = y''_n
  \]

  Again, if you don’t know the values, you could estimate them. The previous edition of your textbook contained a program based on this.

- We could demand that three derivatives match instead of two at the first and last interior points:

  \[
  s'''(x_1 - 0) = s'''(x_1 + 0) \\
  s'''(x_{n-1} - 0) = s'''(x_{n-1} + 0)
  \]

  This is the same as saying that we want the first two and last two cubic polynomials to be identical. (If two cubic polynomials match in value and three derivatives at a point, they are identical). This is called the not-a-knot spline.
The reason for the name is this: A knot is a place where different polynomials come together. Since the first two polynomials are actually the same, $x_1$ is not a knot, even though it may look like one. Likewise for $x_{n-1}$.

The spline portion of program PCHEZ in NMS uses this type of spline.

- For the periodic spline we demand that the first two derivatives match at the endpoints

\[ f'(x_0) = f'(x_n) \]
\[ f''(x_0) = f''(x_n) \]

(and of course $y_0 = y_n$, otherwise this does not make sense). This spline is obviously only useful to interpolate periodic functions.

We can choose a different type of conditions at each endpoint.

There are various ways to go about calculating a cubic spline.

The “matrix approach” is possible, but not very good.

The “basis function approach” works well. The basis functions in this case are called B-splines. Look at section 4.14 in the book, if you want, but don’t expect to understand much from the few crumbs of information the authors are doling out in that section.

The way it is done in the PCHEZ program in the book is by calculating the derivatives of the cubic spline at all $x_j$. Once we have those, we are done: on each subinterval, we have 4 coefficients and 4 constraints (function value and derivative at each end). We can write down a formula for evaluating the spline $s$ and its derivative $s'$ at any point from this data.

The derivative values are found by solving a tridiagonal system of equations of size $(n + 1) \times (n + 1)$. As you recall, this is very fast.

By doing it this way, most of the programming is identical for Hermite cubics and for cubic splines: For Hermite cubics, we estimate the derivatives by interpolation. For cubic splines, we solve a tridiagonal system. After that, everything is identical.

A similar approach with which I am more familiar is to calculate the second derivatives at all $x_j$. This also leads to a tridiagonal system, and we also get formulas to calculate the 4 coefficients on each subinterval from 4 constraints (2 function values and 2 second derivatives, in this case).

### 4.9. PCHIP

Read this section in the book on your own.

Note that there are two programs: PCHEV is called once at the beginning, to get the coefficients set up. Afterwards, PCHEZ is called repeatedly, once or for every set of values where you want the interpolating cubic evaluated.

This is not unlike the internal division between SGEFA and SGESL in Gaussian elimination, or the division between ncoef and neval in my own subroutines.

### 4.10. Cubic Hermite Interpolation: Details

You can skip this section in the book, or read it for your own amusement.

### 4.11. Cubic Splines

Read this section in the book. I have covered the material under section 4.8 above.

### 4.12. Practical Differences Between Splines and cubic Hermite polynomials

Read this section on your own.

The most important piece of information you should remember from this section is that the error estimate for both cubic splines and cubic Hermites looks like

\[ s(x) - g(x) \leq C h^4 |g^{(4)}(\xi)|. \]

(Remember the meta-theorem from earlier?). For equally spaced points, the $h$ is the subinterval size. For unequally spaced points, $h$ is the size of the largest subinterval.
In contrast to polynomial interpolation, the derivative estimate stays fixed for any number of points. If we take more and more points, the error goes to zero (and quite rapidly, too), as long as the length of each subinterval goes to zero.

**Remark:** In addition to better error estimates, piecewise polynomials have another advantage over regular polynomials: if you change the data at one point, this usually produces significant changes in the interpolating polynomial over the entire interval. In contrast, the changes in the interpolating piecewise polynomial tend to “die out” quite rapidly away from the point of change. Changes tend to remain localized.
CHAPTER 5

Numerical Quadrature

5.1. Introduction

The bulk of material that I want to cover in this chapter is in sections 5.2 through 5.4. You should read section 5.7 on your own. As time permits, I may talk about the material in sections 5.5, 5.6 and 5.9.

Read section 5.1 in the book on your own.

The problem we want to investigate in this chapter is that of finding the definite integral of a function, that is, evaluating

\[ \int_{a}^{b} f(x) \, dx \]

numerically. We will only consider the simplest case: a one-dimensional integral, a finite interval \([a,b]\), and a continuous function \(f(x)\). If time permits, I may say a few things about infinite intervals and/or discontinuous functions. We will not cover multidimensional integrals.

Many numerical analysts, including the authors of your textbook, feel that the word integration should be reserved for finding the mathematically exact antiderivative of a function or the area under a curve, while the numerical approximation to that should be called quadrature.

Personally, I have no problem with the phrase numerical integration. I will use the word quadrature in these notes, but in class I will probably use integration and quadrature interchangeably.

5.2. One-dimensional Quadrature Rules and Formulas

I will cover book sections 5.2 and 5.4 in this section of my lecture notes.

The problem we want to solve is the following:

Given a function \(f(x)\) on a finite interval \([a,b]\), calculate numerically an approximation to the definite integral

\[ \int_{a}^{b} f(x) \, dx. \]

We assume that the function \(f\) is at least continuous, and that we can evaluate it at any point in \([a,b]\). For the error estimates later, we have to assume that \(f\) is sufficiently often differentiable.

What form can a quadrature rule take? It is obvious that we can only evaluate \(f\) at a finite number of points \(x_i\). Also, we note that integration is a linear process, which means

\[ \int [\alpha f(x) + \beta g(x)] \, dx = \alpha \int f(x) \, dx + \beta \int g(x) \, dx. \]

A quadrature algorithm should have the same property. This means that our quadrature formula has to be a linear combination of the \(f(x_i)\)

\[ \int_{a}^{b} f(x) \, dx \approx \sum_{i=0}^{n} w_i f(x_i). \]

The \(x_i\) are called the nodes, the \(w_i\) are called the weights. It is desirable that all weights are positive, otherwise we may run into problems with cancellation.
The methods I want to consider are based on the following approaches:

- Choose the $x_i$ equally spaced, and determine the $w_i$ so that the formula is exact for polynomials of as high a degree as possible. This leads to the Newton-Cotes formulas.
- Choose both the $x_i$ and the $w_i$ so that the formula is exact for polynomials of as high a degree as possible. This leads to Gaussian quadrature.
- Choose the $x_i$ any way you want, interpolate a spline through these points, and integrate the spline. Subroutine PCHQA in this chapter is based on this approach. We will not discuss this method in detail.

5.2.1. Newton-Cotes Formulas. The idea behind the Newton-Cotes formulas is to choose the $x_i$ equally spaced throughout $[a,b]$, and to determine the $w_i$ so that the formula is exact for polynomials of as high a degree as possible.

The $w_i$ can be calculated in at least two ways:

1. We replace the function $f$ by the polynomial which interpolates $f$ at the points $x_i$, and integrate the polynomial.
2. We write down a series of equations:
   - numerical integral of $x^0 = \text{exact integral of } x^0$
   - numerical integral of $x^1 = \text{exact integral of } x^1$
   - numerical integral of $x^2 = \text{exact integral of } x^2$
   \ldots

   and solve this system.

I will illustrate both methods in detail when we derive Simpson’s rule.

Newton-Cotes formulas come in two flavors: open and closed.

The closed formulas use the end points of subintervals. If we use $n$ subintervals, the stepsize is $h = (b-a)/n$, and we get $(n+1)$ points

$$x_0 = a, \quad x_1 = a + h, \quad \ldots, \quad x_{n-1} = b - h, \quad x_n = b.$$

The open formulas use the midpoints of subintervals. With $n$ subintervals, the stepsize is again $h = (b-a)/n$, but we get $n$ points

$$x_1 = a + h/2, \quad x_2 = a + 3h/2, \quad \ldots, \quad x_n = b - h/2.$$

The open formulas are useful if the function cannot be evaluated at $a$ and/or $b$.

Newton-Cotes formulas are not recommended for more than 7 points, because the weights become large, and some of them are negative, which leads to cancellation. This is related to the instability of polynomial interpolation we observed before.

There is still an easy way to use more points, for higher accuracy: we simply subdivide $[a,b]$ into smaller intervals, and use a lower order Newton-Cotes formula on each subinterval. These are the repeated or compound Newton-Cotes formulas.

The word “repeated” is often left out. If you read about the “trapezoidal rule” in a book, the author usually means the repeated trapezoidal rule.

We will only consider three types of Newton-Cotes formulas in detail: the midpoint rule (open, $n = 1$), the trapezoidal rule (closed, $n = 1$), and Simpson’s rule (closed, $n = 2$). Using these rules repeatedly, together with some simple extrapolation, is actually more efficient than using higher order rules. If you ever need the higher order rules, you can look them up in a book.

5.2.1.1. Midpoint Rule. The midpoint rule is is the open Newton-Cotes formula for $n = 1$. The only node is $x_1 = (a+b)/2$, the midpoint of the interval.

With only one point, we can only interpolate a polynomial of degree 0, i.e. a constant. From drawing a picture, we see right away

$$\int_a^b f(x) \, dx \approx (b-a)f(x_1).$$

Amazingly enough, this formula also integrates polynomials of degree 1 correctly. (Lucky coincidence).
The error is
\[
\text{error} = \int_a^b f(x) \, dx - (b-a)f(x_1)
\]
\[
= \frac{(b-a)^3}{24} f''(x_1) + \frac{(b-a)^5}{1920} f^{(4)}(x_1) + \ldots
\]
assuming that \( f \) is smooth enough. (See p. 141/142 in the book for the derivation).

The repeated midpoint rule uses points
\[
x_1 = a + h/2, \quad x_2 = a + 3h/2, \quad \ldots, \quad x_n = b - h/2,
\]
where \( h = (b-a)/n \). The formula is
\[
\int_a^b f(x) \, dx \approx h \left[ f(x_1) + f(x_2) + \cdots + f(x_n) \right].
\]
The error on the subinterval \([a + ih, a + (i+1)h]\) is \((h^3/24)f''(x_i) + \ldots\). When we add up the local errors, we get
\[
\text{error} = \frac{h^3}{24} [f''(x_1) + f''(x_2) + \cdots + f''(x_n)] + \ldots
\]
With a little fancy mathematical footwork, we can replace the sum by a single term \(nf''(\xi)\), where \(\xi\) is some unknown point in \([a,b]\). Since \(n = (b-a)/h\), we lose one power of \(h\) for a final error estimate of
\[
\text{error} \approx \frac{b-a}{24} h^2 f''(\xi).
\]

More generally, we can show
\[
\text{error} = c_2 h^2 + c_4 h^4 + c_6 h^6 + \ldots
\]

**Example:** Evaluate numerically
\[
\int_1^2 \log(x) \, dx.
\]
The true answer is \(2 \log(2) - 1 \approx 0.386294361\).

Let us talk about extrapolation briefly.

Except for evaluating \(f(x_i)\), computer time for numerical quadrature is absolutely negligible. There are only a few multiplications and additions. The computer time required to calculate \(f(x_i)\) is unknown, and could potentially be large. An efficient algorithm tries to minimize the number of function evaluations, even at the cost of some other calculations.

Therefore, we want to arrange the extrapolation table so that we can re-use points as much as possible.

If we look at the repeated midpoint rule from this angle, it is apparent that we have to *triple* the number of points between levels, instead of doubling it. In this example, I will evaluate \(f\) at 9 points, which lets me calculate the repeated midpoint rule for 1, 3, and 9 points.

Extrapolation using factors of 3 works almost the same way as the extrapolation using factors of 2 we had before. If the error is \(O(h^p)\), we use
\[
\text{true result} \approx \frac{3^p A(h/3) - A(h)}{3^p - 1}.
\]

For extrapolating twice, we use first \(p = 2\), then \(p = 4\) (see form of error above). This gives the following table:

<table>
<thead>
<tr>
<th>(h)</th>
<th>repeated midpoint rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.405465108</td>
</tr>
<tr>
<td>0.386473708</td>
<td></td>
</tr>
<tr>
<td>1/3</td>
<td>0.388583864</td>
</tr>
<tr>
<td>0.386294955</td>
<td></td>
</tr>
<tr>
<td>0.386297162</td>
<td></td>
</tr>
<tr>
<td>1/9</td>
<td>0.386551240</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.2.1.2. **Trapezoidal Rule.** The trapezoidal rule is the closed Newton-Cotes formula for \( n = 1 \). The nodes are \( x_0 = a, \ x_1 = b \) (the endpoints).

We have two points, so we can interpolate a polynomial of degree 1, i.e. a straight line. By drawing a picture, we see right away that

\[
\int_a^b f(x) \, dx \approx \frac{b-a}{2} \left[ f(x_0) + f(x_1) \right].
\]

This formula integrates polynomials of degree 1 correctly. The area under the line looks like a trapezoid, which is where the formula got its name.

The error is

\[
\text{error} = \int_a^b f(x) \, dx - \frac{b-a}{2} \left[ f(x_0) + f(x_1) \right]
\]

assuming that \( f \) is smooth enough. The notation \( x_{1/2} \) is pretty common: it means the point halfway between \( x_0 \) and \( x_1 \). This is simply the midpoint again, just like in the midpoint rule.

The repeated trapezoidal rule uses points

\[
x_0 = a, \quad x_1 = a + h, \quad \ldots, \quad x_n = b,
\]

where \( h = (b-a)/n \). The formula is

\[
\int_a^b f(x) \, dx \approx h \left[ \frac{1}{2} f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2} f(x_n) \right].
\]

With the same kind of math as before, the total error estimate becomes

\[
\text{error} \approx -\frac{b-a}{12} h^2 f''(\xi),
\]

or in general

\[
\text{error} = c_2 h^2 + c_4 h^4 + c_6 h^6 + \ldots
\]

For the single rule, the trapezoidal rule uses twice as many points as the midpoint rule (two versus one), and the error is twice as large. That does not look very promising.

The error for the repeated trapezoidal rule is still twice as large as for the repeated midpoint rule, but the difference in work is negligible \((n+1)\) points versus \( n \). The fact that we can now extrapolate in steps of 2 instead of 3 makes the repeated trapezoidal rule superior.

The method of using the repeated trapezoidal rule for stepsizes \( h, h/2, h/4 \) etc. and extrapolating is called **Romberg quadrature**.

**Example:** Same integral as before, again 9 points \( (n = 8) \). With these points, we can calculate values for 1, 2, 4, and 8 subintervals, and extrapolate more effectively.

\[
\begin{array}{c|ccc}
  h & \text{error} & \text{error} & \text{error} \\
  \hline
  1 & -0.019170747 & -0.000179347 & -0.000002801 \\
  1/3 & -0.002289503 & -0.000000594 & \\
  1/9 & -0.000256879 & \\
\end{array}
\]
<table>
<thead>
<tr>
<th>$h$</th>
<th>repeated trapezoidal rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.346573590</td>
</tr>
<tr>
<td>1/2</td>
<td>0.376019349</td>
</tr>
<tr>
<td>1/4</td>
<td>0.383699509</td>
</tr>
<tr>
<td>1/8</td>
<td>0.385643910</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$h$</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.039720771</td>
</tr>
<tr>
<td>1/2</td>
<td>0.010275012</td>
</tr>
<tr>
<td>1/4</td>
<td>0.002594852</td>
</tr>
<tr>
<td>1/8</td>
<td>0.000650451</td>
</tr>
</tbody>
</table>

5.2.1.3. **Simpson’s Rule.** Simpson’s rule is the closed Newton-Cotes formula for $n = 2$. The nodes are $x_0 = a$, $x_1 = (a + b)/2$, $x_2 = b$ (endpoints and midpoint).

We have three points, so we can interpolate a polynomial of degree 2, i.e. a parabola. I will show the details here, so you can see how the derivation of Newton-Cotes rules works in general.

For simplicity, we use $[a,b] = [0,1]$, and scale the result later (see section 5.3). The points are $x_0 = 0$, $x_1 = 0.5$, $x_2 = 1$.

**Method 1.** We calculate the interpolating polynomial and integrate that. Any method for finding the polynomial can be used; I will use Lagrange polynomials.

\[
L_{20}(x) = \frac{(x - 1/2)(x - 1)}{(0 - 1/2)(0 - 1)} = 2(x - \frac{1}{2})(x - 1)
\]

\[
L_{21}(x) = -4x(x - 1)
\]

\[
L_{22}(x) = 2x(x - \frac{1}{2})
\]

We integrate those

\[
\int_0^1 L_{20}(x) \, dx = \int_0^1 (2x^2 - 3x + 1) \, dx = \frac{1}{6},
\]

\[
\int_0^1 L_{21}(x) \, dx = \frac{4}{6},
\]

\[
\int_0^1 L_{22}(x) \, dx = \frac{1}{6}.
\]

If $p(x)$ interpolates $f(x)$ at $x_0$, $x_1$, $x_2$, we get the formula

\[
\int_0^1 f(x) \, dx \approx \int_0^1 p(x) \, dx = \int_0^1 [f(x_0)L_{20}(x) + f(x_1)L_{21}(x) + f(x_2)L_{22}(x)] \, dx
\]

\[
= \left[ f(x_0) \int_0^1 L_{20}(x) \, dx + f(x_1) \int_0^1 L_{21}(x) \, dx + f(x_2) \int_0^1 L_{22}(x) \, dx \right]
\]

\[
= \frac{1}{6} \left[ f(x_0) + 4f(x_1) + f(x_2) \right].
\]
Method 2. We know that we want a formula
\[ \int_{0}^{1} f(x) \, dx \approx w_0 f(x_0) + w_1 f(x_1) + w_2 f(x_2) \]
which can integrate polynomials of degree 2 or less correctly. We write out
\[ w_0 x_0^0 + w_1 x_0^1 + w_2 x_0^2 = \int_{0}^{1} x^0 \, dx \]
\[ w_0 x_1^0 + w_1 x_1^1 + w_2 x_1^2 = \int_{0}^{1} x^1 \, dx \]
\[ w_0 x_2^0 + w_1 x_2^1 + w_2 x_2^2 = \int_{0}^{1} x^2 \, dx \]
simplified
\[ w_0 + w_1 + w_2 = 1 \]
\[ \frac{1}{2} w_1 + w_2 = \frac{1}{2} \]
\[ \frac{1}{4} w_1 + w_2 = \frac{1}{3} \]
This leads to the same result.

When scaling to \([a, b]\), we pick up an extra factor of \((b - a)\) (see section 5.3). The final formula is
\[ \int_{a}^{b} f(x) \, dx \approx \frac{b - a}{6} \left[ f(x_0) + 4f(x_1) + f(x_2) \right]. \]
This formula integrates polynomials of degree 2 correctly, by construction, and also works for polynomials of degree 3 (lucky accident, again).

Actually, it can be proved in general that every other Newton-Cotes formula is one degree more accurate than it ought to be. The closed Newton-Cotes rule for \(n = 3\) (called the 3/8-rule) also works for polynomials of degree 3 only. The one after that works for polynomials of degree 5, and so on.

In Germany, Simpson’s rule is also known as the barrel rule. Barrel makers used it to calculate the volume of a barrel very accurately from the circumference in the middle and at the ends.

The error in Simpson’s rule is
\[ \text{error} = \int_{a}^{b} f(x) \, dx - \frac{b - a}{6} \left[ f(x_0) + 4f(x_1) + f(x_2) \right] \]
assuming that \(f\) is smooth enough.

The repeated Simpson rule uses points
\[ x_0 = a, \quad x_1 = a + h, \quad \ldots, \quad x_{2n} = b. \]
where \(h = (b - a)/(2n)\). The formula is
\[ \int_{a}^{b} f(x) \, dx \approx \frac{h}{3} \left[ f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n}) \right]. \]
The total error estimate becomes
\[ \text{error} \approx -\frac{b - a}{180} h^4 f^{(4)}(\xi), \]
or in general
\[ \text{error} = c_4 h^4 + c_6 h^6 + c_8 h^8 + \ldots \]
How did the number 6 in the single formula turn into 3 in the repeated formula? Likewise, how did the number 2880 in the error estimate turn into 180? Well, the \(h\) in the repeated formula corresponds to \( \frac{1}{2} \) the subinterval width. The missing factors of 2 and \(2^4 = 16\) are hiding in the \(h\) and \(h^4\).
**Example:** Same integral as before, again using 9 points \((n = 8)\). With these points, we can calculate values for 1, 2, and 4 subintervals, and extrapolate.

<table>
<thead>
<tr>
<th>(h)</th>
<th>repeated Simpson rule</th>
<th>(h)</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>0.385834602</td>
<td>1/2</td>
<td>0.000459759</td>
</tr>
<tr>
<td></td>
<td>0.386287894</td>
<td>0.000000468</td>
<td></td>
</tr>
<tr>
<td>1/4</td>
<td>0.386259563</td>
<td>1/4</td>
<td>0.000034798</td>
</tr>
<tr>
<td></td>
<td>0.386294309</td>
<td>0.000000052</td>
<td></td>
</tr>
<tr>
<td>1/8</td>
<td>0.386292043</td>
<td>1/8</td>
<td>0.000002318</td>
</tr>
</tbody>
</table>

These numbers look remarkably like the extrapolation columns from the repeated trapezoidal rule. This is no coincidence. Let’s prove it.

We can use extrapolation on entire formulas, instead of just on the numbers these formulas produce. Suppose we have \(f(x)\) given at \(x_0, x_1, x_2\), stepsize \(h\). If \(A(2h)\) is the result of using the single trapezoidal rule on \([x_0, x_2]\), and \(A(h)\) is the result of using two trapezoidal rules on the subintervals, then

\[
A(2h) = h [f(x_0) + f(x_2)] \\
A(h) = h \left[ \frac{1}{2} f(x_0) + f(x_1) + \frac{1}{2} f(x_2) \right] \\
\text{extrapolated value} = \frac{4A(h) - A(2h)}{3} \\
= \frac{h}{3} [f(x_0) + 4f(x_1) + f(x_2)].
\]

In other words: one step of extrapolation based on the (single or repeated) trapezoidal rule produces the (single or repeated) Simpson’s rule! Extrapolating Simpson’s rule produces the rule for \(n = 4\), called the Milne rule, but I don’t want to pursue this.

**5.2.2. Gaussian Quadrature.** Gaussian quadrature is based on polynomial interpolation, just like the Newton-Cotes formulas, except the choice of nodes is different.

In the Newton-Cotes formulas, the idea was: let the \(x_j\) be given, and determine the \(w_j\) so that we can integrate polynomials up to a certain order exactly. With \(n\) points, we have \(n\) degrees of freedom, and we can hope to integrate polynomials up to degree \((n - 1)\) exactly. That is indeed what happens, except that in some formulas we get lucky and can do it up to degree \(n\).

In Gaussian quadrature, we let both the \(x_j\) and the \(w_j\) be negotiable. With \(n\) points, we now have \(2n\) degrees of freedom, and we hope to be able to integrate polynomials up to degree \((2n - 1)\) exactly. This is indeed possible.

The details of the derivation are beyond the scope of this course. The result is that for quadrature on \([-1, 1]\), the nodes \(x_i\) should be chosen to be the zeros of the \(n\)-th Legendre polynomial \(P_n(x)\) (whatever that is). The weights are then calculated from a system of equations, like before. For quadrature on intervals other than \([-1, 1]\), nodes and weights are scaled (see section 5.3).

A table with some nodes and weights is on page 147 in the textbook.

The error for the \(n\)-point rule can be shown to be

\[
\frac{(b - a)^{2n+1}(n!)^4}{(2n + 1) [(2n)!]^3} f^{(2n)}(\xi),
\]

as long as \(f\) has \((2n)\) continuous derivatives. A repeated \(n\)-point Gaussian rule would have global error \(O(h^{2n})\).
**Example:** Same integral as before.
For $n = 2$, the scaled nodes and weights are

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.211324865</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1.788675135</td>
<td>0.5</td>
</tr>
</tbody>
</table>

(See section 5.3 for the derivation).

The approximate integral is 0.386594944, with an error of 0.00300583. This is very good compared to other rules with only two points.

For $n = 4$, the scaled nodes and weights are

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.069431844</td>
<td>0.173927423</td>
</tr>
<tr>
<td>2</td>
<td>1.33009478</td>
<td>0.326072577</td>
</tr>
<tr>
<td>3</td>
<td>1.66990522</td>
<td>0.326072577</td>
</tr>
<tr>
<td>4</td>
<td>1.930568156</td>
<td>0.173927423</td>
</tr>
</tbody>
</table>

The approximate integral is 0.386294497, with an error of $-0.000000136$. This is excellent accuracy for only four points.

### 5.3. Change of Interval

Assume we have a quadrature rule on the interval $[\alpha, \beta]$ with nodes $\xi_i$ and weights $\omega_i$.

$$
\int_{\alpha}^{\beta} f(x) \, dx \approx \sum_i \omega_i f(\xi_i).
$$

We now want to integrate a function on a different interval $[a, b]$. How do we transform the nodes and weights?

With the substitution

$$
t = a + \frac{b - a}{\beta - \alpha}(x - \alpha)
$$

$$
x = \alpha + \frac{\beta - \alpha}{b - a}(t - a)
$$

we get

$$
\int_{a}^{b} f(t) \, dt = \int_{\alpha}^{\beta} f \left( \alpha + \frac{b - a}{\beta - \alpha}(x - \alpha) \right) \, dx
$$

$$
\approx \sum_i \left( \frac{b - a}{\beta - \alpha} \omega_i \right) f \left( \alpha + \frac{b - a}{\beta - \alpha}(\xi_i - \alpha) \right)
$$

$$
= \sum_i w_i f(x_i),
$$

so we see

$$
w_i = \frac{b - a}{\beta - \alpha} \omega_i
$$

$$
x_i = a + \frac{b - a}{\beta - \alpha}(\xi_i - \alpha)
$$

**Example:** The nodes and weights for 4-point Gaussian quadrature on $[-1, 1]$ are

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.861136311594052</td>
<td>0.347854845137454</td>
</tr>
<tr>
<td>2</td>
<td>-0.339981043584856</td>
<td>0.652145154862546</td>
</tr>
<tr>
<td>3</td>
<td>0.339981043584856</td>
<td>0.652145154862546</td>
</tr>
<tr>
<td>4</td>
<td>0.861136311594052</td>
<td>0.347854845137454</td>
</tr>
</tbody>
</table>
If we want to use these for quadrature on \([1,2]\), we have \(\alpha = -1, \beta = 1, a = 1, b = 2\). The formulas become

\[
\begin{align*}
  w_i &\rightarrow \frac{2 - 1}{1 - (-1)}w_i = \frac{1}{2}w_i \\
  x_i &\rightarrow 1 + \frac{1}{2}(x_i + 1) = \frac{3}{2} + \frac{1}{2}x_i
\end{align*}
\]

and we get the new nodes/weights I used above

<table>
<thead>
<tr>
<th>(i)</th>
<th>(x_i)</th>
<th>(w_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.069431844</td>
<td>0.173927423</td>
</tr>
<tr>
<td>2</td>
<td>1.330009478</td>
<td>0.326072577</td>
</tr>
<tr>
<td>3</td>
<td>1.66990522</td>
<td>0.326072577</td>
</tr>
<tr>
<td>4</td>
<td>1.930568156</td>
<td>0.173927423</td>
</tr>
</tbody>
</table>

\[\square\]

### 5.4. Compound Quadrature Rules and Error Estimates

Read this section in the book. I have included the material from this section in my section 5.2.

### 5.5. Gauss-Kronrod Quadrature Rules

Before I go into Gauss-Kronrod rules, let me talk a little about the uses of extrapolation.

So far, we have used extrapolation as a way to improve the accuracy of numerical results. A second use is as an error estimate.

Suppose we calculate a result twice, with different step sizes, and extrapolate. Assume the original results are \(A(h)\) and \(A(h/2)\), and \(X\) is the extrapolated value. We expect \(X\) to be noticeably more accurate than either one of the original results, so

\[
\text{error in } A(h/2) = |A(h/2) - \text{true result}| \approx |A(h/2) - X|.
\]

Since \(X\) is expected to be more accurate than \(A(h/2)\), its error should be even smaller. Thus, we use \(X\) as the approximate answer, and \(|A(h/2) - X|\) as an error bound. This error bound will usually be quite pessimistic, and we may be able to find a more realistic bound empirically. “Empirically” means by trial and error. We do a whole bunch of test examples and compare the error bound and the actual error, to see if we can derive a more realistic error bound that works most of the time. (Here is where numerical analysis turns into an art, instead of a science).

Now, let us go back to Gaussian quadrature.

As I have discussed before, a good quadrature routine tries to minimize the number of function evaluations.

When we do extrapolation on the Newton-Cotes formulas, we can re-use old points if we decrease the stepsize by a factor of 2 (closed rules) or 3 (open rules). For repeated Gaussian quadrature, there is no regular spacing, and standard extrapolation requires a whole new set of points every time.

Kronrod found a way around this. He started with an \(n\)-point Gaussian quadrature rule

\[
\int_{-1}^{1} f(x) \, dx \approx \sum w_i f(x_i)
\]

and tried to add \(m\) other points \(y_j\) and find a new formula

\[
\int_{-1}^{1} f(x) \, dx \approx \sum a_i f(x_i) + \sum b_j f(y_j)
\]

The \(x_i\) are fixed, the \(y_j, a_i\) and \(b_j\) are to be determined. We have \((2m + n)\) degrees of freedom, so we hope we can integrate polynomials up to degree \((2m + n - 1)\) exactly.

For the choice \(m = (n + 1)\) and some values of \(n\), this actually works. The smallest choice that works is \(n = 7\), where the details are given in the textbook. Other choices that work are \(n = 10, n = 15, n = 20, n = 25, n = 30\). (Kronrod must have used up a lot of paper deriving these).
Now, this is not really extrapolation. For the basic Gauss-Kronrod rule, we have a 7-point rule whose error depends on $f^{(14)}$, and a 15-point rule whose error depends on $f^{(23)}$. We cannot combine the two to form a more accurate estimate.

We can still use the two results for an error estimate: we use the 15-point rule as the more accurate result, and the difference between the 7-point rule and the 15-point rule as the error estimate. A more realistic empirical estimate is shown on page 154.

Thus, the (7, 15) Gauss-Kronrod pair is really a 15-point rule that comes with an error estimate derived from 7 of these points.

5.6. Automatic and Adaptive Quadrature Algorithms

An automatic algorithm is one that will accept a requested accuracy and will attempt to produce a result whose error estimate is less than the requested accuracy. This shifts the burden of error estimation from the user to the subroutine.

An adaptive algorithm is one that will try to adapt to the local behavior of the problem. In the case of quadrature, that means that the algorithm will use a small stepsize where the function seems to vary rapidly, and a larger stepsize where the function seems to be smooth.

An automatic adaptive algorithm could be programmed about like this: First, the programmer selects a pair of basic quadrature rules. This could be a Gauss-Kronrod pair, or two Simpson rules with different step sizes. When applied to a subinterval, this pair of rules will produce an approximate integral, along with an error estimate.

Then, we apply the basic rules to the whole interval. Suppose the requested accuracy is a number $\epsilon$. If the error estimate for the whole interval is less than $\epsilon$, we are done. If not, we divide the interval into two and try to integrate each half with an accuracy of $\epsilon/2$. For some pairs of basic rules, we can even re-use some of the previous points here. We keep subdividing until we achieve the required accuracy on each subinterval. (There should be some other stopping criterion, too, to avoid endless calculations).

The routine Q1DA in your textbook uses a similar approach, based on the 15-point Gauss-Kronrod rule.

5.7. Subroutines Q1DA and QK15

Read this section on your own.

5.8. Data Integration

We skip this subsection. (Now, shouldn’t that be Data Quadrature?)

5.9. Improper Integrals

A proper integral is the definite integral of a continuous function over a bounded interval. Anything else is improper. Thus, either we have a discontinuous function, or the interval is infinite (or both).

Discontinuities come in many forms, but the types that show up most frequently in practice are simple jumps and points where the function goes to infinity. Jumps are harmless: just subdivide the interval there. Likewise, if the function blows up in the middle somewhere, we subdivide the interval, so that we only need to worry about blowups at the endpoints.

By subdividing the interval, we only need to worry about two types of improper integrals: integrals where the function blows up at one endpoint, and integrals over infinite intervals.

There are three basic ways for handling integrals over infinite intervals:

- Truncate the interval to a finite one.
- Transform the interval to a finite one.
- Use a special quadrature rule for infinite intervals.

Two of these techniques can also be applied to integrals in which the function blows up at an endpoint:

- Use a transformation that makes the singularity disappear.
- Use a special quadrature rule for the singularity.

Let us look at these techniques in more detail.
5.9.1. **Truncation.** If the integrand decreases fast enough, we can cut off the integration.

**Example:** Evaluate \( \int_0^\infty e^{-x} \cos^2(x^2) \, dx \) numerically.

We estimate

\[
\int_A^\infty e^{-x} \cos^2(x^2) \, dx < \int_0^\infty e^{-x} \, dx = e^{-A}.
\]

If we want an absolute error of \( \epsilon = 10^{-5} \), we could use \( A = 11.5 \), since \( e^{-11.5} \approx 10^{-5} \). Thus,

\[
\int_0^\infty e^{-x} \cos^2(x^2) \, dx \approx \int_0^1 1.5e^{-x} \cos^2(x^2) \, dx,
\]

which can be evaluated by a quadrature routine.

**Example:** Evaluate \( \int_0^\infty \frac{dx}{1+x^2} \) numerically.

We estimate

\[
\int_A^\infty \frac{dx}{1+x^2} < \int_a^\infty \frac{dx}{x^2} = \frac{1}{A}.
\]

For an accuracy of \( \epsilon = 10^{-5} \), we would need \( A = 10^5 \). This does not seem practical.

5.9.2. **Transformation.** This technique works both for infinite intervals and for endpoint singularities in the integrand.

**Example:** Evaluate \( \int_0^\infty e^{-x} \cos^2(x^2) \, dx \) numerically.

We could use the transformation \( x = -\log t \), so that

\[
\int_0^\infty e^{-x} \cos^2(x^2) \, dx = \int_0^1 \cos^2(\log^2 t) \, dt.
\]

This does not work, since the integrand oscillates wildly near 0.

**Example:** Evaluate \( \int_0^\infty e^{-x} \cos^2(x^2) \, dx \) numerically.

Let us try \( x = -2\log t \). In this case

\[
\int_0^\infty e^{-x} \cos^2(x^2) \, dx = \int_0^1 2t \cos^2(4\log^2 t) \, dt.
\]

This works better. The integrand still oscillates rapidly, but at least it goes to zero near 0. (You can’t get rid of the oscillations, since the original function oscillates at infinity).

**Example:** Evaluate \( \int_0^\infty \frac{dx}{\sqrt{x}} \) numerically.

With the substitution \( x = t^2 \), we get

\[
\int_0^1 \frac{\cos x}{\sqrt{x}} = \int_0^1 \cos t^2 \, dt,
\]

which is easy to evaluate numerically.

5.9.3. **Weighted Quadrature Rules.** Many improper integrals are of the form

\[
\int_a^b f(x)w(x) \, dx,
\]

where \( w(x) \) is a *weight function*, which could be “bad”, and \( f(x) \) is a “nice” function. Standard weight functions are powers of \( x \), logarithms, and exponentials, or combinations of these.

The idea is to derive quadrature formulas which incorporate the weight functions, of the form

\[
\int_a^b f(x)w(x) \, dx \approx \sum w_i f(x_i).
\]

If a suitable weight function can be extracted from the integrand, this approach would be my first choice.
Let us look at a couple of examples.

**Example:** Evaluate $\int_0^1 \frac{\cos x}{\sqrt{x}} \, dx$ by a modified Simpson rule. Here $f(x) = \cos x$, and $w(x) = 1/\sqrt{x}$.

The modified Simpson rule is derived in the same way the standard Simpson rule was derived. We pick $x_0 = 0$, $x_1 = 1/2$, $x_2 = 1$, and try to find weights $w_i$ so that the rule

$$\int_0^1 f(x) \frac{1}{\sqrt{x}} \, dx \approx w_0 f(x_0) + w_1 f(x_1) + w_2 f(x_2)$$

is exact for $f(x) = x^0, x^1, x^2$. This leads to a system of equations

$$w_0 x_0^0 + w_1 x_1^0 + w_2 x_2^0 = \int_0^1 x^0 \frac{1}{\sqrt{x}} \, dx$$

$$w_0 x_0^1 + w_1 x_1^1 + w_2 x_2^1 = \int_0^1 x^1 \frac{1}{\sqrt{x}} \, dx$$

$$w_0 x_0^2 + w_1 x_1^2 + w_2 x_2^2 = \int_0^1 x^2 \frac{1}{\sqrt{x}} \, dx,$$

simplified

$$w_0 + w_1 + w_2 = 2$$

$$\frac{1}{2} w_1 + w_2 = \frac{2}{3}$$

$$\frac{1}{4} w_1 + w_2 = \frac{2}{5}$$

with solution $w_0 = 12/15$, $w_1 = 16/15$, $w_2 = 2/15$.

We get

$$\int_0^1 \frac{\cos x}{\sqrt{x}} \, dx \approx \frac{12}{15} \cos 0 + \frac{16}{15} \cos \frac{1}{2} + \frac{2}{15} \cos 1 = 1.808128373 \ldots$$

The true value is 1.809048476, the error is 0.000920103, which is not bad.

Example: Evaluate $\int_0^\infty e^{-x} x^{1.2} \, dx$ numerically (see book, page 167).

The modified Gaussian quadrature rules with weight function $e^{-x}$ on the interval $[0, \infty)$ are called *Gauss-Laguerre rules*. Look at the derivation in the book (pages 166/167).

The numerical value is 1.088468113, the exact value is 1.101802491, so the error is 0.0133344. Pretty good for only two points.

Remark: my textbook gives the exact integral as 1.046. That is wrong: 1.046 \ldots is the value for a power of 1.1.
CHAPTER 6
Linear Least-Squares Data Fitting

6.1. Introduction
Recall that in chapter 3 we were discussing linear systems of equations, written in shorthand in the form

\[ Ax = b. \]

In chapter 3, we just considered square, nonsingular systems, which have exactly one solution. In this chapter, we look at the general case. The system might not have any solution, or it might have infinitely many. We will define something called the least squares solution, and look at 3 different methods for finding it.

6.1.1. Linear Least Squares Problems. Let us assume we have a set of \( m \) linear equations in \( n \) unknowns

\[ Ax = b. \]

Recall from chapter 3 that if \( m < n \), the system is underdetermined, and in general has an infinite set of solutions. If \( m > n \), the system if overdetermined, and in general has no solution. Even if \( m = n \), there may be no solution or an infinite set of solutions, if the matrix is singular.

Let us tackle the case of no solution first. Recall the definition of residual

\[ r = Ax - b. \]

If \( x \) is a solution, the residual is zero. If there is no solution, the usual approach is to make \( r \) as small as possible, and to call the corresponding \( x \) a generalized solution.

Now \( r \) is a vector. Making it as small as possible means that we have to minimize its norm. Which norm? Well, any choice of norm leads to a corresponding mathematical problem, and in general different norms lead to different generalized solutions. The norm that is easiest to handle in practice is the 2-norm, and that is all we will do in this chapter. In practice, one minimizes the square of the 2-norm, since then we don’t have to carry all those square roots around. The result is the same.

If there is a unique \( x \) for which the 2-norm of the residual is a minimum, that works fine. What if there are an infinite number of \( x \) that minimize the residual? In the textbook, this is called the degenerate case, and is treated separately in section 6.8. I will treat it along with the non-degenerate case.

In the degenerate case, it is customary to choose the \( x \) which is itself the smallest, out of all \( x \) which minimize \( r \).

Altogether, we have the following definition: The least squares solution of a system

\[ Ax = b \]

of linear equations is the \( x \) which minimizes

\[ \|Ax - b\|. \]

If there is more than one such \( x \), we choose the one for which \( \|x\| \) is a minimum.

Note that if there is a solution, it will also be the only generalized solution.
6.1.2. Source of Linear Least Squares Problems. A common source of these problems, and in fact the only source we will discuss in this class, is linear approximation at a discrete set of points, or curve fitting.

Recall from chapter 4 that the object of interpolation is to find a curve from a certain family that takes on given values. The object of approximation is to find a curve from a certain family that passes as close as possible to some given points or to a given curve. The case of approximating a curve is more advanced; we will concentrate on finding a curve that passes close to a finite set of points.

In the linear case, the interpolating or approximating function must be of the form

\[ f(x) = \sum_j a_j \phi_j(x), \]

where the \( \phi_j \) are the basis functions (polynomials, splines, or whatever), and the \( a_j \) are the coefficients to be determined. If we have \( n \) basis functions and want the curve to go through or near \( m \) points, we get an \( m \times n \) system of linear equations.

**Example:** Put a straight line through the points \((1, 1), (1.5, 2), (2, 2), (2.5, 3)\). If we assume the equation

\[ f(x) = a_0 + a_1 x \]

for the line (i.e. \( \phi_0(x) = 1, \phi_1(x) = x \)), we get the equations

\[
\begin{align*}
    a_0 + a_1 &= 1 \\
    a_0 + 1.5a_1 &= 2 \\
    a_0 + 2a_1 &= 2 \\
    a_0 + 2.5a_1 &= 3
\end{align*}
\]

The source of the numbers might be a lab experiment: you know that the solution is supposed to be a straight line, but your measurements contain errors, and there is only an approximate line passing through the points. □

6.1.3. Weighted Least Squares Problems. Instead of minimizing the square of the 2-norm of the residual

\[ \|r\|^2 = \sum_j r_j^2 \]

we could introduce weights \( w_j \geq 0 \), and instead minimize

\[ \sum_j w_j r_j^2. \]

This is called a weighted least squares problem. This is useful for example if you know that some measurements are more accurate than others, and you want to put more emphasis on those than on some others.

The mathematics gets more complicated in weighted least squares problems, but the basic ideas remain the same. We will not consider this in detail.

6.2. Exploring Data

We will skip this section.

6.3. The Normal Equations

This method is only capable of minimizing the residual, i.e. it can only handle the non-degenerate case. This implies that we must have \( m \geq n \). (The case \( m < n \) is always degenerate).

There are two approaches to deriving this method. One approach is to take a semester of linear algebra, after which you can derive the equations in two or three lines. The other approach uses calculus and is a little messy, so I will just describe it in words:

You write out the square of the 2-norm of the residual as a quadratic function in the unknowns \( x_1 \) through \( x_n \). To minimize it, you set the \( n \) partial derivatives to zero, and get a system of \( n \) linear equations in \( n \) unknowns.
Either way, you end up with the so-called *normal equations*. In matrix notation, they are written as

\[ A^T A x = A^T b \]

This is now a square system of size \( n \times n \) and can be solved by Gaussian elimination. (Actually, the matrix \( A^T A \) is always symmetric, so there is a faster way known as *Cholesky decomposition*, but Gaussian elimination also works).

The matrix \( A^T A \) may turn out to be singular. If that happens, your matrix is degenerate, and you should switch to a different method for solving the least squares problem.

**Example:** For the equations from above, we get

\[
A = \begin{pmatrix}
1 & 1 \\
1 & 1.5 \\
1 & 2 \\
1 & 2.5
\end{pmatrix}, \quad A^T A = \begin{pmatrix} 4 & 7 \\
7 & 13.5 \end{pmatrix},
\]

\[
b = \begin{pmatrix} 1 \\
2 \\
2 \\
3 \end{pmatrix}, \quad A^T b = \begin{pmatrix} 8 \\
15.5 \end{pmatrix},
\]

The solution of the normal equations is \( a_0 = -0.1, a_1 = 1.2 \), so the line that best fits the data is

\[ y = 1.2x - 0.1 \]

The normal equations approach may work quite well for small values of \( n \). However, even for moderately large values of \( n \) (like 20 or 30), the normal equations tend to become unstable. The reason is that the condition number of \( A^T A \) is approximately the square of the condition number of \( A \). (The condition number of a non-square matrix is defined on page 62 of NMS).

Thus, if \( A \) has a reasonable condition number of about 1,000, then \( A^T A \) already has a condition number of \( 10^6 \). For this reason, the method of normal equations is not recommended for anything but very small systems.

### 6.4. Orthogonal Factorizations

The technique described in this section (called the *QR factorization*) takes about twice as much computer time as the normal equations, but it is much more stable. It can also handle the degenerate case, and is recommended for most cases.

**Remark:** In the degenerate case, this method does not actually find the least squares solution described earlier. It does find an \( x \) which minimizes the residual, but instead of resolving the problem of multiple solutions by choosing the \( x \) with minimum 2-norm, it is content to select an \( x \) for which the 2-norm is small, but not necessarily the minimum. The exact effect is hard to describe geometrically.

Recall the basic approach to solving a system of linear equations: we used a sequence of allowed operations to transform the system to a simpler one (triangular matrix), and then solved the simpler one. Allowable operations were those that did not change the solution, like interchanging rows and adding a multiple of one row to another.

The same approach is used here: we apply a sequence of allowed operations to a least squares problem to transform it to an easier one. The easier problem involves again a triangular matrix. Allowable operations in this case are those that do not change the 2-norms of vectors. This means rotations and reflections.

Let me state that again, for emphasis: The least squares solution of

\[ Ax = b \]

is in general not the same as the least squares solution of

\[ B Ax = Bb, \]
but they are the same if $B$ is a matrix corresponding to a rotation or reflection. These matrices are called orthogonal.

The official definition is this: a square matrix $Q$ is called orthogonal if $Q^T = Q^{-1}$. Orthogonal matrices have many nice properties; among the most important ones are

- $\det(Q) = \pm 1$.
- Orthogonal matrices correspond to either a rotation of $n$-dimensional space (if $\det = 1$), or a rotation combined with a reflection (if $\det = -1$).
- The column vectors of $Q$ all have 2-norm 1 and are mutually orthogonal. (Vectors are called orthogonal if their dot product is zero). Geometrically, this means they are perpendicular. From this property the matrices got their name.
- The same thing is true for the rows.
- Transpose, inverse and product of orthogonal matrices are orthogonal.
- If $v$ is any vector, then $v$ and $Qv$ have the same 2-norm. This is because the 2-norm is the geometric length, and a rotation or reflection does not change the length.
- The condition number of an orthogonal matrix is 1.

Orthogonal matrices are popular in numerical analysis because operations with them are very stable. The condition number of a matrix not only affects the solution of systems of equations, but also simple multiplication of a vector by the matrix. If there are roundoff errors in $v$, then $Av$ usually has larger errors.

Not so for orthogonal matrices, where the condition number is 1. You could multiply $v$ by a thousand orthogonal matrices in a row (sometimes you have to do that), and it would not magnify the roundoff error at all. If numerical stability is important, methods based on orthogonal matrices are usually recommended.

The property $Q^T = Q^{-1}$ is nice, too: If you need the inverse, it is right there.

Now, back to the main topic. Recall that Gaussian elimination was based on the factorization

$$A = LU,$$

where the matrix $A$ was factored into a lower triangular matrix $L$ and an upper triangular matrix $U$. There is another factorization (called the QR factorization)

$$A = QR,$$

where $A$ is factored into an orthogonal matrix $Q$ and an upper triangular matrix $R$.

Let us not worry about how exactly you do that, even though the book does that in some detail. I could talk about this topic for several days. There are two main techniques, based on Householder matrices and on Givens rotations. They sort of work like Gaussian elimination: we find an orthogonal matrix which wipes out a particular element below the diagonal (in the case of Givens rotations) or an entire column (in the case of Householder matrices). Then we find another orthogonal matrix which wipes out the next element or column, etc. In the end we multiply them all together to form $Q$.

Anyway, after we have factored $A$, we can transform the system to one with an upper triangular matrix, and the same solution:

$$Ax = QRx = b.$$

$$Rx = Q^Tb$$

We are mostly interested in the non-degenerate case. The matrix $R$ in this case looks like this:

$$R = \begin{pmatrix}
  r_{11} & \cdots & \cdots & r_{1n} \\
  0 & r_{22} & \cdots & \vdots \\
  \vdots & \ddots & \ddots & \vdots \\
  \vdots & \ddots & r_{nn} & \vdots \\
  0 & \cdots & \cdots & 0
\end{pmatrix}$$
where \( r_{ii}! = 0 \). The rows of \( R \) that are all zero do not contribute anything to the solution. We just throw them away and solve the rest.

**Example:** For the same example as before, the program **DQRLS** (the double precision version of program **SQRLS** in the book) produces the following factorization:

\[
A = \begin{pmatrix}
1 & 1 \\
1 & 1.5 \\
1 & 2.5
\end{pmatrix} = QR
\]

\[
= \begin{pmatrix}
-0.5 & 0.670820393250 & 0.023606797750 & 0.547213595500 \\
-0.5 & 0.223606797750 & -0.439344662917 & -0.712022659167 \\
-0.5 & -0.223606797750 & 0.807868932583 & -0.217595468167
\end{pmatrix} \begin{pmatrix}
-2 & -3.5 \\
0 & 1.1803398875 \\
0 & 0
\end{pmatrix}
\]

You can verify that this \( Q \) has all the right properties (all columns and rows are orthogonal and have length 1, etc.) Actually, the last two columns of \( Q \) are not needed in the following, but I calculated them, anyway. The last two columns of \( Q \) are only needed to calculate the last two entries in \( Q^T b \), and we know that those will be thrown away.

The new system is now

\[
Rx = Q^T b
\]

\[
\begin{pmatrix}
-2 & -3.5 \\
0 & 1.1803398875 \\
0 & 0
\end{pmatrix} \begin{pmatrix}
a_0 \\
a_1
\end{pmatrix} = \begin{pmatrix}
-4 \\
-0.1341640787 \\
-0.164809064
\end{pmatrix}
\]

The triangular system becomes

\[
\begin{pmatrix}
-2 & -3.5 \\
0 & 1.1803398875
\end{pmatrix} \begin{pmatrix}
a_0 \\
a_1
\end{pmatrix} = \begin{pmatrix}
-4 \\
-0.1341640787
\end{pmatrix}
\]

which gives the same solution \( a_0 = -0.1, a_1 = -1.2 \) as before. \( \square \)

If the problem is degenerate, the matrix looks like this:

\[
R = \begin{pmatrix}
r_{11} & \ldots & \ldots & \ldots & \ldots & r_{1n} \\
0 & r_{22} & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
\vdots & \ddots & r_{kk} & \ldots & r_{kn} \\
\vdots & 0 & \ldots & 0 \\
\vdots & \vdots & \vdots & \vdots \\
0 & \ldots & 0 & 0 & 0 & \ldots & 0 & \ldots & r_{nn}
\end{pmatrix}
\]

Here again we throw out the zero equations, but there is no unique way of solving the rest. Usually, one sets \( x_{k+1} \) through \( x_n \) to zero and solves the rest. In other words: we throw away the part of \( R \) which is not triangular.

As mentioned earlier, this does not correspond exactly to the solution \( \mathbf{x} \) with minimum 2-norm, but it does give a useful and unique answer.

Likewise, if \( m < n \), we get a matrix

\[
R = \begin{pmatrix}
r_{11} & \ldots & \ldots & \ldots & \ldots & r_{1n} \\
0 & r_{22} & \vdots \\
\vdots & \ddots & \ddots & \vdots \\
0 & \ldots & r_{mm} & \ldots & r_{mn}
\end{pmatrix}
\]

and again set \( x_{m+1} \) through \( x_n \) to zero.
6.5. Subroutine SQRLS

Read this section on your own.

6.6. Historical Perspective: Gauss

Read this section on your own.

6.7. Degenerate Least-Squares Problems

Read this section on your own. I have already incorporated degenerate problems above.

6.8. The Singular Value Decomposition

The ultimate in equation solving is the *singular value decomposition* (SVD). It is extremely stable and really tells you what is going on, if you know how to use it right. However, it takes 5 to 10 times as long as the normal equations and should be reserved for special cases.

The *singular value decomposition* of a matrix \( A \) is given by

\[
A = U \Sigma V^T,
\]

where \( U \) and \( V \) are orthogonal, and \( \Sigma \) is diagonal. If \( A \) is of size \( m \times n \), then \( U \) is \( m \times m \), \( V \) is \( n \times n \) and \( \Sigma \) is \( m \times n \). Depending on whether \( m \) is smaller or larger than \( n \), \( \Sigma \) may look like either

\[
\Sigma = \begin{pmatrix}
\sigma_1 & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & 0 \\
\vdots & \vdots & \ddots & \sigma_n \\
0 & \cdots & \cdots & 0
\end{pmatrix}
\quad \text{or} \quad
\Sigma = \begin{pmatrix}
\sigma_1 & 0 & \cdots & \cdots & \cdots & 0 \\
0 & \ddots & \ddots & \vdots & & \\
\vdots & \ddots & \ddots & \ddots & \ddots & \\
\vdots & \vdots & \ddots & \ddots & \ddots & \\
0 & \cdots & \cdots & \cdots & \cdots & \sigma_m \\
0 & \cdots & \cdots & \cdots & \cdots & 0
\end{pmatrix}
\]

All the \( \sigma_j \) are non-negative real numbers, even if the matrix is complex. They are called the *singular values*. Generally, one arranges them by size, so that

\[
\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0.
\]

We will assume this ordering from now on.

With this assumption on the ordering, the matrix \( \Sigma \) is uniquely determined. The other matrices \( U \) and \( V \) are not unique, but that makes no difference, as long as we can find some that work.

The singular values are the square roots of the eigenvalues of \( A^T A \) (or \( A^H A \) in the complex case). The quotient of the largest and smallest singular value is a good estimate of the condition number of \( A \):

\[
\text{cond}(A) \approx \frac{\sigma_1}{\sigma_n}.
\]

The matrix \( A \) is degenerate if and only if one or more of the singular values are zero. Very small singular values signal a numerical problem. However, once we have the singular value decomposition of a matrix, we can take steps to fix the situation. I will explain this more below.

Let us go back to the least squares problem. We want to solve

\[
Ax = b,
\]

or equivalently

\[
U \Sigma V^T x = b.
\]

We can do this in two steps:

**Step 1.** Multiply through by \( U^T \). As before, this will not affect the least squares solution, because \( U \) is orthogonal. We get

\[
\Sigma V^T x = U^T b.
\]
Let \( y = V^T x \), \( U^T b = c \) and solve the least squares problem
\[
\Sigma y = c
\]

**Step 2.** Once you have \( y \), solve the problem
\[
V^T x = y.
\]

This is actually trivial, namely
\[
x = V y.
\]

What we have achieved is to reduce a general least squares problem to one with a diagonal matrix. This is even better than the reduction to a triangular problem we had before, in the QR-algorithm.

Let us consider the general (possibly degenerate) case right away. The diagonal system will look like this:
\[
\begin{pmatrix}
\sigma_1 & 0 & \cdots & \cdots & \cdots & 0 \\
0 & \ddots & \ddots & \ddots & \ddots & \ddots \\
\vdots & \ddots & \sigma_k & 0 & \cdots & 0 \\
\vdots & \ddots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & \cdots & 0 & \cdots & \cdots & 0
\end{pmatrix}
\begin{pmatrix}
y_1 \\
y_2 \\
\vdots \\
y_k \\
y_{k+1} \\
y_{k+2} \\
\vdots \\
y_n
\end{pmatrix}
= \begin{pmatrix}
c_1 \\
c_2 \\
\vdots \\
c_k \\
c_{k+1} \\
c_{k+2} \\
\vdots \\
c_m
\end{pmatrix}
\]

As before, we have a number of zero equations which we throw away. The remaining equations have the solution
\[
y_1 = c_1/\sigma_1 \\
y_2 = c_2/\sigma_2 \\
\vdots \\
y_k = c_k/\sigma_k \\
y_{k+1} = \text{arbitrary} \\
\vdots \\
y_n = \text{arbitrary}
\]

If we want to pick a unique solution by choosing the \( y \) with the smallest norm, there is only one way to do that: set \( y_{k+1} \) through \( y_n \) to zero. This gives you the true least squares solution, in the sense defined before (minimize the residual first; if there are infinitely many solutions, pick the smallest one).

**Example:** Consider the same example as before, with
\[
A = \begin{pmatrix}
1 & 1 \\
1 & 1.5 \\
1 & 2 \\
1 & 2.5
\end{pmatrix}, \quad b = \begin{pmatrix}
1 \\
2 \\
2 \\
3
\end{pmatrix},
\]

Subroutine DSVDC (Double precision Singular Value Decomposition) from LINPACK gives the following:
\[
U = \begin{pmatrix}
-0.32586337856 & 0.770590569877 & 0.023606797750 & 0.547213595500 \\
-0.43236606927 & 0.336242646335 & -0.43934462917 & -0.712022659167 \\
-0.53884875998 & -0.098105277206 & 0.807868932583 & -0.217595468167 \\
-0.645363145069 & -0.32453200748 & -0.392131067417 & 0.384204531833
\end{pmatrix},
\]
\[
\Sigma = \begin{pmatrix}
4.148428878243 & 0.0 & 0 \\
0 & 0.539015623295 & 0 \\
0 & 0 & 0
\end{pmatrix}, \quad V^T = \begin{pmatrix}
-0.468240633469 & -0.883600989796 \\
-0.883600989796 & -0.468240633469
\end{pmatrix}
Again, the last two columns of $U$ are not really needed. We find

$$c = U^T b = \begin{pmatrix} -4.204420738913 \\ -0.350494294107 \\ -0.415737865167 \\ -0.164809063667 \end{pmatrix}, \quad y = \begin{pmatrix} -1.013497124408 \\ -0.650248859142 \end{pmatrix},$$

and finally

$$x = V y = \begin{pmatrix} -0.1 \\ 1.2 \end{pmatrix}$$
as before. \(\square\)

To understand why small singular values are bad, consider the example

$$\begin{pmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 10^{-8} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

The solution is obviously

$$y_1 = c_1 / 10,$$
$$y_2 = c_2,$$
$$y_3 = 10^8 c_3$$

Whatever error was contained in $c_3$ has been magnified by a huge factor and is now probably larger than the solution itself.

The same thing would have happened with the original matrix $A$, before you did a SVD, but you would not have been able to see it. Now we have rotated the coordinate system so that the “bad” part points in the direction of $e_3$, and we see exactly what the problem is. In this particular case, it would be better to set $y_3$ to zero.

There is an analogy between this approach and filtering of signals in Electrical Engineering: The Fourier transform decomposes a signal into its frequency components. The noise is contained mostly in the high frequency components, so you can try to cut those off. You lose some of the fine detail of the signal, but you get rid of a lot of the noise. Where exactly you cut depends on the nature of the signal.

The singular value decomposition decomposes the matrix into its effect on various directions. The error comes mostly from the very small singular values, so you can try to cut those off. You lose some fine detail in the solution, but you get rid of a lot of the error. Where exactly you cut depends on the nature of the problem.

Note that we can sometimes significantly lower the condition number of the rest of the matrix by throwing out a few of the very small singular values. In the above example, getting rid of $\sigma_3$ lowers the condition number from $10^9$ to 10. In this way, you can still get a halfway satisfactory solution out of a very ill-conditioned system. However, the singular value decomposition is much more computer intensive than other methods and should be used sparingly.

### 6.8.1. Summary

We have discussed three methods for finding the least squares solution of a system of linear equations. They were

- The method of normal equations. This is the fastest method, but it has two drawbacks: it only works for non-degenerate, overdetermined systems, and it often is numerically unstable.
- QR factorization. This method is recommended for most cases. It can handle all cases and is usually stable. It takes about twice as much computer time as the method of normal equations.
- SVD. This is for cases where QR factorization fails. It takes five to ten times more computer time than the normal equations, but is extremely stable. It can also be used for other purposes, for example when Gaussian elimination fails.
CHAPTER 7
Solution of Nonlinear Equations

7.1. Introduction

We will cover the material in sections 7.1, 7.2 and 7.5 in the book. Read sections 7.3 and 7.6 on your own.

7.1.1. Nonlinear Equations. We have discussed the solution of linear equations in chapter 3. However, many equations that come up in real life are nonlinear. In this chapter, we will discuss the numerical solution of the general nonlinear equation \( f(x) = 0 \) (one-dimensional case) and \( f(x) = 0 \) (higher dimensions).

Example: Solve the equation

\[ e^x = x + 2. \]

In standard form, this is written

\[ f(x) = e^x - x - 2 = 0. \]

Example: Solve the system of equations

\[
\begin{align*}
  x_1 x_2 - x_2^3 &= 1 \\
  x_1^2 x_2 + x_2 &= 5
\end{align*}
\]

In standard form, this is written

\[ f(x) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} x_1 x_2 - x_2^3 - 1 \\ x_1^2 x_2 + x_2 - 5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \]

Some of the differences between linear and nonlinear equations are

- A system of linear equations has either no solution, exactly one solution or an infinite number of solutions. If there are infinitely many solutions, they form a linear subspace (straight line, plane, etc.)

In contrast, nonlinear equations can have any number of solutions. (Think of the equation \( \sin(x) = 0 \)). It is usually not obvious how many solutions there are at the start. This causes a problem, namely: when do you know that you are finished? Even if you have already found 10 solutions, you do not know if there are any more.

We will ignore this question. All we want to do is find one solution. If we need to find several, we will find one solution several times.

- There is in general no algorithm that finds a solution of a nonlinear equation in a finite number of steps. We have to use iterative techniques (see 7.1.3).
7.1.2. Root Finding and Optimization. There is a strong connection between the following problems:

- Solving \( f(x) = 0 \), where \( f \) is a function from \( \mathbb{R}^n \) to \( \mathbb{R}^n \). This is what this chapter is about.
- Finding a local minimum of a function \( F(x) \), where \( F \) is a function from \( \mathbb{R}^n \) to \( \mathbb{R} \). This is called optimization and is the subject of chapter 9.

The problem of solving \( f(x) = 0 \) can be converted to the problem of finding a minimum of

\[
F(x) = [f_1(x)]^2 + [f_2(x)]^2 + \cdots + [f_n(x)]^2
\]

Wherever \( f \) has a zero, \( F \) has a minimum. (\( F \) may have other minima, too).

The problem of finding a minimum of \( F \) can be converted to a root-finding problem if \( F \) is differentiable: at a local minimum, all partial derivatives are zero, so a local minimum of \( F \) is a zero of \( f = \nabla F \). (See my section 7.5 for a review of multivariate calculus). \( f \) may have other zeros, as well (local maxima, points of inflection, saddle points).

There are methods for solving equations, and there are methods for finding minima. With the appropriate conversion, you can apply either approach to either problem.

If you look at the software that comes with this chapter, you will find two subroutines. \textsc{FZERO} can find a zero of a function of a single variable. It uses a combination of methods from this chapter. \textsc{SNSQE} can find a zero of a system of equations. It converts the problem internally to an optimization problem and uses a combination of methods from chapter 9.

7.1.3. Iterative Methods and Errors. A deterministic method is one which produces an answer in a finite number of steps. If all calculations were done in infinite precision, we would get the exact answer. Gaussian elimination is a deterministic method.

An iterative method is one which produces a sequence \( x_0, x_1, \ldots \) of approximate solutions which converges to a solution. Even with infinite precision, none of the approximate solutions will ever be the true solution. In practice, you can quit once you get as close as the machine epsilon, of course. All methods in this chapter are iterative.

Let \( \overline{x} \) be the exact solution. We define the error at the \( n \)th step as

\[
e_n = |x_n - \overline{x}|.
\]

The method is said to be of order \( p \) if

\[
e_{n+1} \leq ce_n^p,
\]

where \( c \) is some positive constant. In many methods used in practice, \( p \) is an integer (usually between one and four), but not always.

**Theorem 7.1.** A method of order 1 converges if and only if \( c < 1 \). A method of order \( p > 1 \) always converges if \( x_0 \) is close enough to \( \overline{x} \).

**Proof:** Assume \( p = 1 \). Then

\[
e_n \leq ce_{n-1} \leq c^2 e_{n-2} \leq \cdots \leq c^n e_0.
\]

You can see that \( e_n \) goes to zero if and only if \( c < 1 \).

Assume \( p > 1 \). Then

\[
e_1 \leq ce_0^p = ce_0^{p-1} e_0 = de_0.
\]

If \( e_0 \) is small enough, then \( d = ce_0^{p-1} \) is less than 1, so \( e_1 \) is less than \( e_0 \). You can then verify that

\[
e_2 \leq ce_1^p = ce_1^{p-1} e_1 \leq ce_0^{p-1} e_1 = de_1 \leq d^2 e_0,
\]

and so on as before. \( \square \)

What is important in practice is the speed of convergence associated with various \( p \). In general, we want \( p \) to be as large as possible. Assume, for example, that \( c = 0.1, e_0 = 0.1 \). Then we find

\[
\begin{array}{c|c|c|c}
p & e_1 & e_2 & e_3 \\
1 & 10^{-2} & 10^{-4} & 10^{-6} \\
2 & 10^{-3} & 10^{-6} & 10^{-9}
\end{array}
\]

Notice that every iteration step increases the accuracy by one digit if \( p = 1 \). If \( p = 2 \), every iteration step roughly doubles the number of correct digits.
7.2. Methods for Computing Real Roots

The title seems to imply that there is another section Methods for Computing Complex Roots, but there is not. What is covered in this section is the one-dimensional problem, while section 7.5 covers the higher-dimensional problem.

7.2.1. The Bisection Method. This is a very basic method, but it always works if you can get it started, and converges in a predictable number of iterations.

We need to assume that \( f \) is continuous. The intermediate value theorem then tells us that if \( f \) is positive at some point and negative at another point, there has to be a zero in between.

The algorithm works as follows:

- Start with initial points \( a_0, b_0 \) so that \( f(a_0)f(b_0) < 0 \).

  (This is just a way of saying that \( f \) has different signs at \( a_0 \) and \( b_0 \)).

- Let \( x_0 \) be the midpoint \( x_0 = \frac{a_0 + b_0}{2} \) and calculate \( f(x_0) \). If \( f(x_0) = 0 \), we are done. Otherwise, one of two things happens: Either, \( f(a_0)f(x_0) < 0 \), in which case we set \( a_1 = a_0, b_1 = x_0 \); or \( f(a_0)f(x_0) > 0 \), in which case we set \( a_1 = x_0, b_1 = b_0 \).

- Set \( x_1 = \frac{a_1 + b_1}{2} \) and continue like this until \( b_n - a_n \) is small enough.

In words, we are doing the following: We start with an interval where \( f \) is positive at one end, negative at the other, so we know there is a zero in between. We keep dividing the interval in two and continue with either the left half or the right half, whichever is the one on which \( f \) has different signs.

Either we find the zero by accident or we keep going until the interval is small enough for our purposes. In single precision arithmetic we have 24 bits for the mantissa. Each iteration gains one bit, so we need exactly 24 iterations for full precision.

It is impossible to estimate \( e_n \) exactly for this method, but on the average, we have \( e_{n+1} \approx \frac{1}{2} e_n \).

The method is linearly convergent.

**Example:** Consider the equation \( f(x) = e^x - x - 2 = 0 \).

A graph shows that this equation has exactly two solutions, one positive and one negative. We determine by trial and error that \( f(1) < 0, f(2) > 0 \), so we can start with \( a_0 = 1, b_0 = 2, x_0 = 1.5 \), to find the positive
solution. We get

\[
\begin{align*}
x_0 &= 1.5 \\
x_1 &= 1.25 \\
x_2 &= 1.125 \\
x_3 &= 1.1875 \\
x_4 &= 1.15625 \\
x_5 &= 1.140625 \\
x_6 &= 1.1484375 \\
x_7 &= 1.14453125 \\
x_8 &= 1.146448375 \\
x_9 &= 1.145507813 \\
x_{10} &= 1.145996094 \\
\ldots
\end{align*}
\]

Since \( e_0 \leq 0.5 \), \( e_{n+1} \approx 0.5e_n \), we expect \( e_{10} \) to be of the order of \( 2^{-11} \approx 5 \cdot 10^{-4} \). The true solution (to 10 decimals) is

\[ x = 1.146193221. \]

The true error \( e_{10} \) is about \( 2 \cdot 10^{-4} \). \( \square \)

7.2.2. Fixed Point Iteration. A solution of the equation

\[ g(x) = x \]

is called a fixed point of \( g \). We could try to find a fixed point by fixed point iteration, which is the following algorithm:

\[
\begin{align*}
x_0 &= \text{initial guess} \\
x_1 &= g(x_0) \\
x_2 &= g(x_1) \\
x_3 &= g(x_2) \\
\ldots
\end{align*}
\]

and so on.

**Theorem 7.2.** Let \( \overline{x} \) be a fixed point of \( g \). If \( g \) is continuously differentiable, and \( |g'(\overline{x})| < 1 \), then fixed point iteration will converge linearly to \( \overline{x} \) if \( x_0 \) is close enough to \( \overline{x} \). If \( |g'(\overline{x})| > 1 \), fixed point iteration will not converge for \( \overline{x} \).

**Proof:** Everything depends on the simple formula

\[ e_{n+1} = |x_{n+1} - \overline{x}| = |g(x_n) - g(\overline{x})| = |g'(\xi)| |x_n - \overline{x}| = |g'(\xi)| e_n, \]

where \( \xi \) is some point between \( x \) and \( \overline{x} \).

If \( |g'(\overline{x})| < 1 \), then we can find a neighborhood of \( \overline{x} \) in which \( |g'| \leq c < 1 \). As long as \( x_0 \) is chosen inside that neighborhood, \( e_{n+1} \leq c e_n \), and we get linear convergence.

If \( |g'(\overline{x})| > 1 \), then errors increase in the neighborhood of \( \overline{x} \), and we can’t have convergence. \( \square \)

**Remark:** The proof is based on a Taylor series expansion of \( g(x) \) around \( \overline{x} \). If \( g'(\overline{x}) = 0 \) and \( g \) is twice continuously differentiable, we get similarly

\[ e_{n+1} = \frac{1}{2} |g''(\xi)| e_n^2, \]

so the method is quadratically convergent in this case.

How do we apply fixed point iteration to the problem \( f(x) = 0 \)? First, we have to rewrite the equation in fixed point form \( g(x) = x \). There are many ways to do that. For any particular \( g \), fixed point iteration may work or not, depending on what the derivative of \( g \) does near the solutions.
The necessity to do manipulations by hand, without guarantee whether it will work or not, and the slow linear convergence rate, make fixed point iteration one of the less popular methods. Still, there are uses for it, usually in the case when the equation already looks like \( g(x) = x \) (for example predictor-corrector methods in the numerical solution of ODEs).

**Example:** Consider again our example

\[ f(x) = e^x - x - 2 = 0. \]

**Attempt 1.** A simple way to get this into fixed point form is

\[ g(x) = e^x - 2 = x. \]

Here, \( g'(x) = e^x \), so fixed point iteration will converge for negative \( x \), but not for positive \( x \). Indeed,

<table>
<thead>
<tr>
<th>( x )</th>
<th>negative root</th>
<th>positive root</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 )</td>
<td>-1.</td>
<td>2.</td>
</tr>
<tr>
<td>( x_1 )</td>
<td>-1.1632120559</td>
<td>5.389056099</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>-1.804485466</td>
<td>216.996576863</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>-1.835440894</td>
<td>1.73946592 \cdot 10^4</td>
</tr>
<tr>
<td>( x_4 )</td>
<td>-1.840456855</td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_{12} )</td>
<td>-1.841405660</td>
<td></td>
</tr>
</tbody>
</table>

**Attempt 2.** Solve for the \( x \) in the exponent:

\[ g(x) = \log(x + 2) = x. \]

Here, \( g'(x) = 1/(x + 2) \), and we get convergence everywhere except in \([-3, -1]\). We can use this to find the positive root.

<table>
<thead>
<tr>
<th>( x )</th>
<th>positive root</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 )</td>
<td>1.</td>
</tr>
<tr>
<td>( x_1 )</td>
<td>1.098612289</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>1.130954362</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>1.141337866</td>
</tr>
<tr>
<td>( x_4 )</td>
<td>1.144648781</td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>( x_{17} )</td>
<td>1.146193221</td>
</tr>
</tbody>
</table>

With some experimentation, I am sure we could discover a single \( g(x) \) which works for both roots.  

7.2.3. **Aitken’s \( \Delta^2 \) Method.** This is not a method for finding zeros, but a method for speeding up the convergence of any linearly convergent method (like fixed point iteration, in this case).

Consider any linearly convergent method, where a sequence \( \{x_n\} \) converges to a solution \( \bar{x} \). We can take three approximate solutions in a row and try to eliminate the error term, like we did in extrapolation. Starting from

\[
\begin{align*}
  x_n &= \bar{x} - e_n \\
  x_{n+1} &= \bar{x} - e_{n+1} \approx \bar{x} - ce_n \\
  x_{n+2} &= \bar{x} - e_{n+2} \approx \bar{x} - c^2 e_n
\end{align*}
\]

we do a bunch of algebra, and find

\[
\bar{x} \approx x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} = x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n}
\]

The \( \Delta \) notation is called a **forward difference** and is just a shorthand for the formula in the middle.

The most efficient way to use it is to just do 3 iterations, extrapolate, then do 3 more iterations with the extrapolated value, and so on. This is called **Steffenson’s method.**
Example: Consider again our example
\[ f(x) = e^x - x - 2 = 0. \]

We use the fixed point formulation
\[ x = \log(x + 2) \]
to find the positive root. We get the values (the horizontal lines represent extrapolations)
\[
\begin{align*}
x_0 &= 1 \\
x_1 &= 1.098612289 \\
x_2 &= 1.130954362 \\
x_3 &= 1.146738371 \\
x_4 &= 1.146366479 \\
x_5 &= 1.146248288 \\
x_6 &= 1.146193227 \\
x_7 &= 1.146193223 \\
x_8 &= 1.146193221 \\
\end{align*}
\]
The last value is already correct.

7.2.4. Newton’s Method. This method is also known as the Newton-Raphson method. It requires a differentiable function \( f \) and works like this:

- Find an initial guess \( x_0 \).
- Calculate the tangent to \( f \) at \( x_0 \) and intersect it with the \( x \)-axis. This is the next point \( x_1 \). The equation is
\[
x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}
\]
- Repeat this procedure.

This method converges quadratically \((p = 2)\) as long as we have a simple zero (one where the derivative is not also zero). This can be shown by treating it as a fixed point iteration, with
\[
g(x) = x - \frac{f(x)}{f'(x)}.
\]

Newton’s method is very easy to program as long as \( f \) is easily differentiable, and it is very popular. Still, Newton’s method is not as foolproof as bisection. Possible problems you need to be prepared for are

- Division by zero, if \( f'(x_n) = 0 \). The line we want to intersect with the \( x \)-axis happens to be parallel to it.
- Huge jump from \( x_n \) to \( x_{n+1} \), if \( f'(x_n) \approx 0 \). The line we want to intersect with the \( x \)-axis happens to be almost parallel to it, so the intersection is far away.
- Cycles in the \( x_n \). If \( x_2 \) happens to be equal to \( x_0 \), then \( x_3 = x_1 \), \( x_4 = x_2 = x_0 \), and so on. Cycles of any length can happen.
- Divergence (the \( x_n \) get bigger and bigger).

All of these are fixed the same way: stop, and try again with a different \( x_0 \). As long as there is a solution, the method will work if your initial guess is appropriate.

Example: For the equation
\[ f(x) = e^x - x - 2 = 0 \]
the initial guess \( x_0 = 1 \) leads to
\[
\begin{align*}
x_0 &= 1 \\
x_1 &= 1.163953414 \\
x_2 &= 1.146421185 \\
x_3 &= 1.146193259 \\
x_4 &= 1.146193221 \\
\end{align*}
\]
and \( x_4 \) is already accurate to 10 places.  

The main disadvantage of Newton’s method is that we need to evaluate derivatives. The method is not applicable if the function has no derivative or if the derivative takes excessive amounts of user time or computer time to calculate. One way out is to approximate the derivative by a difference quotient. This leads to the secant method.

### 7.2.5. The Secant Method

The secant method can be derived in a number of different ways. I find it easiest to consider it as a discrete version of Newton’s method. Instead of a tangent at \( x_n \), we use the secant through \( x_{n-1}, x_n \).

The secant method works as follows:

- Choose two initial points \( x_0, x_1 \).
- Put a straight line through \((x_0, f(x_0))\) and \((x_1, f(x_1))\). The next point \( x_2 \) is the place where this line intersects the \( x \)-axis. We can calculate that

\[
x_2 = \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)},
\]

or equivalently

\[
x_2 = x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)}.
\]

The first form looks more symmetric. The second form makes the connection with Newton’s method clearer, and is also more numerically stable.

- Repeat this process: \( x_3 \) is found by putting a line through \((x_1, f(x_1))\) and \((x_2, f(x_2))\), and so on.

The secant method has the same potential problems as Newton’s method (division by zero, loops, etc.), and they can be fixed by starting over again with different \( x_0, x_1 \). When the method works, it works quite well and is of order \( p \approx 1.618 \).

**Example:** For the same equation

\[
f(x) = e^x - x - 2 = 0
\]

and the same initial guesses as for the bisection method, we find

\[
\begin{align*}
x_0 &= 1 \\
x_1 &= 2 \\
x_2 &= 1.076746253 \\
x_3 &= 1.113782265 \\
x_4 &= 1.147913165 \\
x_5 &= 1.146151855 \\
x_6 &= 1.146193168 \\
x_7 &= 1.146193220 \\
x_8 &= 1.146193221
\end{align*}
\]

The last number is full accuracy on my calculator.

### 7.2.6. Summary of One-Dimensional Methods

We have discussed four methods for finding the zero of a one-dimensional function.

- **Bisection.** This method is slow but reliable. It converges in a predictable number of steps.
- **Fixed Point Iteration.** This method is mostly useful if the equation is already in the form \( g(x) = x \), and if it can be shown beforehand that it will converge. Not recommended for general problems.
- **Newton’s method.** This is the fastest of the four and is the method of choice if the derivative of \( f \) is easily available.
- **Secant method.** This is a form of Newton’s method with the derivative replaced by a difference quotient. It is slower than Newton’s method, but still converges faster than linear methods.
The best general purpose method is a combination of the Bisection and Secant methods: Use the secant method of the time. If it gives crazy answers or tries to divide by zero, use bisection for a step or two. This is what routine \texttt{FZERO} does.

7.3. Subroutine \texttt{FZERO}

Read this section on your own.

7.4. Historical Perspective

Read this section on your own.

7.5. Systems of Nonlinear Equations

In this section, I will first review multidimensional calculus. You should have seen this stuff before, of course, but it doesn’t hurt to see it again. After that, we will look at methods.

There is no counterpart in higher dimensions for the bisection method, because the concepts of “positive” and “negative” cease to exist. You can get from any point in a plane to any other point without passing through the origin.

Fixed point iteration and Newton’s method work the same way as in one dimension. The secant method, which we considered as a discretized version of Newton’s method, is replaced by \textit{Quasi-Newton methods}, which we only mention in passing.

7.5.1. Review of Multivariate Calculus. Let $f$ be a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$

$$f(x) = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_m(x_1, \ldots, x_n) \end{pmatrix}.$$  

The first derivative of $f$ at the point $x$ is a linear mapping between the same spaces, i.e. an $m \times n$ matrix

$$Df(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$Df(x) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{pmatrix}.$$  

$$[Df(x)](h) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{pmatrix} \begin{pmatrix} h_1 \\ \vdots \\ h_n \end{pmatrix} = \left( \sum_j \frac{\partial f_j(x)}{\partial x_j} h_j \right)$$

The last line shows how the mapping $[Df(x)]$ maps an $n$-vector $h$ into an $m$-vector.

The second derivative of $f$ at the point $x$ is a bilinear mapping between the same spaces. A bilinear mapping takes two arguments and is linear in each one when the other one is held fixed. This cannot be written in matrix form any more except when $m = 1$ (see below).

$$D^2f(x) : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$[D^2f(x)](h, k) = \left( \sum_{j,l} \frac{\partial^2 f_j(x)}{\partial x_j \partial x_l} h_j k_l \right)$$

The third derivative of $f$ at the point $x$ is a trilinear mapping, and so on. The multi-dimensional Taylor series now looks just like its one-dimensional counterpart:

$$f(x + h) = f(x) + [Df(x)](h) + \frac{1}{2!} [D^2f(x)](h, h) + \frac{1}{3!} [D^3f(x)](h, h, h) + \ldots$$  

(Mathematicians love to hide the complications inside the notation.)
Let us look more closely at the two special cases we will really need in this class:

**Case 1:** \( m = n \). This is what we need in this chapter. The first derivative is a square matrix, called the **Jacobian**, and we don’t mess with higher derivatives.

\[
Df(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n
\]

\[
Df(x) = \begin{pmatrix}
\frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_n(x)}{\partial x_1} & \cdots & \frac{\partial f_n(x)}{\partial x_n}
\end{pmatrix}
\]

**Case 2:** \( m = 1 \). This is what we need in chapter 9. The first derivative is a \( 1 \times n \) matrix, which is a single row.

\[
Df(x) = \begin{pmatrix}
\frac{\partial f(x)}{\partial x_1} \\
\vdots \\
\frac{\partial f(x)}{\partial x_n}
\end{pmatrix}
\]

The transpose of the derivative is a vector called the **gradient**

\[
\nabla f(x) = [Df(x)]^T = \begin{pmatrix}
\frac{\partial f(x)}{\partial x_1} \\
\vdots \\
\frac{\partial f(x)}{\partial x_n}
\end{pmatrix}
\]

In this special case, the second derivative can be written as a matrix, called the **Hessian**.

\[
D^2 f(x) = \begin{pmatrix}
\frac{\partial^2 f(x)}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_n}
\end{pmatrix}
\]

which is applied to its two arguments like this

\[
[D^2 f(x)] (h, k) = h^T [D^2 f(x)] k
\]

You can check that this matrix multiplication results in a \( 1 \times 1 \) matrix whose single entry is the sum

\[
\sum_{j,l} \frac{\partial^2 f(x)}{\partial x_j \partial x_l} h_j k_l,
\]

just like it should be.

We don’t mess with the third and higher derivatives. All we need is

\[
f(x + h) \approx f(x) + [Df(x)] h + \frac{1}{2} h^T [D^2 f(x)] h
\]

### 7.5.2. Fixed Point Iteration.

Fixed point iteration works in any dimension. The one-dimensional condition

\[
|g'(\tau)| < 1
\]

gets replaced by

\[
\rho(Dg(\tau)) < 1
\]

where \( \rho \) is the **spectral radius**, which is the size of the largest eigenvalue. Except for special cases (such as predictor-corrector methods) it is hard to pick a suitable \( g \), so this method is very hard to apply. Not recommended in general.

**Example:** Consider the equations

\[
x^2 - 10x + y^2 + 8 = 0 \\
xy^2 + x - 10y + 8 = 0
\]
One way of turning this into a fixed point problem is

\[
x = \frac{x^2 + y^2 + 8}{10}, \\
y = \frac{xy^2 + x + 8}{10}
\]

Starting with

\[
x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix},
\]

we get

\[
x_1 = \begin{pmatrix} 0.8 \\ 0.8 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 0.928 \\ 0.9312 \end{pmatrix}
\]

and so on. Fixed point iteration converges to the solution \(x = 1, y = 1\). In this case, the spectral radius is 0.4, so it works.

7.5.3. Newton’s Method. Newton’s method also generalizes easily to higher dimensions. We start with

\[
f(\mathbf{x}) \approx f(x) + [Df(x)](\mathbf{x} - x)
\]

Now if \(\mathbf{x}\) is a zero of \(f\), and \(x\) is any old vector, then

\[
0 \approx f(x) + [Df(x)](\mathbf{x} - x) \\
\mathbf{x} \approx x - [Df(x)]^{-1} f(x)
\]

We get Newton’s method

\[
x_0 = \text{guess} \\
x_{n+1} = x_n - [Df(x_n)]^{-1} f(x_n)
\]

Compare this to the one-dimensional version

\[
x_{n+1} = x_n - \frac{f(x)}{f'(x)}
\]

**Example:** Consider the equations

\[
x^2 - 10x + y^2 + 8 = 0 \\
xy^2 + x - 10y + 8 = 0
\]

The derivative of \(f\) is a \(2 \times 2\) matrix

\[
Df(x, y) = \begin{pmatrix} 2x - 10 & 2y \\ y^2 + 1 & 2xy - 10 \end{pmatrix}
\]

If we start with the initial guess

\[
x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix},
\]
we find

\[
Df(x_0) = Df(0, 0) = \begin{pmatrix} -10 & 0 \\ 1 & -10 \end{pmatrix},
\]

\[
f(x_0) = f(0, 0) = \begin{pmatrix} 8 \\ 8 \end{pmatrix},
\]

\[
x_1 = x_0 - [Df(x_0)]^{-1} f(x_0)
\]

\[
= \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \frac{1}{100} \begin{pmatrix} -10 & 0 \\ -1 & -10 \end{pmatrix} \begin{pmatrix} 8 \\ 8 \end{pmatrix}
\]

\[
= \begin{pmatrix} 0.8 \\ 0.88 \end{pmatrix}
\]

At the next step,

\[
Df(x_1) = Df(0.8, 0.88) = \begin{pmatrix} -8.4 & 1.76 \\ 1.7744 & -8.592 \end{pmatrix},
\]

\[
f(x_1) = f(0.8, 0.88) = \begin{pmatrix} 1.4144 \\ 0.61952 \end{pmatrix},
\]

\[
x_2 = x_1 - [Df(x_1)]^{-1} f(x_1)
\]

\[
= \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \frac{1}{0.69049856} \begin{pmatrix} -8.592 & -1.76 \\ -1.7744 & -8.4 \end{pmatrix} \begin{pmatrix} 8 \\ 8 \end{pmatrix}
\]

\[
= \begin{pmatrix} 0.991787 \ldots \\ 0.9917 \ldots \end{pmatrix}
\]

and so on. This iteration converges to the solution \((1, 1)\). □

Most of what you need to know about higher-dimensional Newton’s method is already apparent from this example:

• Newton’s method still works very well and is still quadratically convergent.
• It takes a lot more work in higher dimensions.

Another (negative) fact is not so obvious: The higher the dimension, the better your initial guess has to be.

This is not something you can really prove. It is something which is observed in practice. For example, to find the positive zero of the function \(e^x - x - 2\) by Newton’s method, you can start with almost any positive number. If you guess \(x_0 = 10^6\), it will take longer, but the method will still converge to the solution. In higher dimensions, your initial guess needs to be pretty close to the answer.

Newton’s method in higher dimensions is not as popular as in one dimension.

7.5.4. Quasi-Newton methods. It is easy to see that the main work in Newton’s method is in calculating and inverting the \(Df\) matrix, not to mention that you have to calculate the derivatives of \(f\) by hand and provide subroutines for them.

A lot of effort has gone into trying to cut corners here. What you get are various forms of so-called quasi-Newton methods. They all try to replace the inverse derivative matrix by another matrix which is not quite the right one, but close, and is much easier to find. There are various ways to do that. Typically, these methods do not converge quadratically, but still superlinear (better than linear, that is \(p > 1\)). They are higher-dimensional counterparts of the secant method.

7.6. Subroutine SNSQE

Read this section on your own. As I mentioned above, SNSQE converts a root-finding problem to a minimization problem, and applies techniques from chapter 9.
CHAPTER 8

Ordinary Differential Equations

8.1. Introduction

My section 8.1 will cover the material in sections 8.1 and 8.2 in the book. Read the book sections on your own.

I don’t like the order of things in the textbook. I will cover the sections in the following order: 8.1, 8.2, 8.4, 8.5, 8.6, 8.19, 8.8, 8.9, 8.10, 8.11. Read section 8.7 on your own, and skip the rest.

The material on Theory of ODES in this section should have been covered in a previous course on ordinary differential equations. It is included to refresh your memory, not to teach you the material.

8.1.1. Theory of ODEs. An ordinary differential equation (ODE) is an equation involving an unknown function \( y(t) \) and its derivatives. The order of the ODE is the order of the highest derivative. Assuming that we can always solve for this highest derivative, the most general form of an nth order ODE is

\[ y^{(n)}(t) = f(t, y(t), y'(t), \ldots, y^{(n-1)}(t)). \]

The function \( f \) can be quite complicated.

The functions \( y \) and \( f \) could also be vectors

\[
y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{pmatrix}, \quad f(t, y(t), y'(t), \ldots, y^{(n-1)}(t)) = \begin{pmatrix} f_1(t, y_1(t), \ldots, y_m(t), y'_1(t), \ldots, y^{(n-1)}_m(t)) \\ f_2(t, y_1(t), \ldots, y_m(t), y'_1(t), \ldots, y^{(n-1)}_m(t)) \\ \vdots \\ f_m(t, y_1(t), \ldots, y_m(t), y'_1(t), \ldots, y^{(n-1)}_m(t)) \end{pmatrix}.
\]

The resulting equation

\[ y^{(n)}(t) = f(t, y(t), y'(t), \ldots, y^{(n-1)}(t)). \]

is called a vector ODE or a system of ODEs.

The solution of an ODE is not a number, but a set of functions defined on some interval. To solve an nth order ODE, you have to perform \( n \) integrations, and you pick up \( n \) arbitrary constants in the process. The solution is an \( n \)-parameter family of curves.

To find a unique solution function, you need \( n \) extra pieces of information. If these extra conditions are given in terms of the values of \( y, y', y'', \ldots \) at some point \( t_0 \), this is called an initial value problem (IVP). The extra conditions are then called initial conditions (IC).

If the extra conditions involve values of \( y, y', \ldots \) at two or more points, we have a boundary value problem (BVP), and the conditions are the boundary conditions (BC). In BVP, the independent variable is often called \( x \) instead of \( t \), since typically it represents a space variable. In IVP, \( t \) usually represents time.

In this course, we will concentrate only on IVP. The numerical techniques for IVP and BVP are quite different. The techniques used for BVP are similar to the techniques used for solving partial differential equations, and really belong in a course on numerical solution of PDE.

In a course on ODEs, great emphasis is placed on the distinction between linear and non-linear ODEs. Linear ODEs are much easier to solve by hand. For the numerical methods covered in this chapter, there is no difference between linear and nonlinear equations.
As a preparation for numerical solution, we can transform any higher order IVP into a first order system.

**Example:** Suppose we want to solve the IVP

\[ u'''(t) = -u'(t)u^2(t) + \sin t \]

\[ u(0) = -1 \]

\[ u'(0) = 1 \]

\[ u''(0) = 2 \]

We introduce new variables

\[ y_1 = u \]

\[ y_2 = u' \]

\[ y_3 = u'' \]

and get the equations

\[ y'_1 = y_2 \]

\[ y'_2 = y_3 \]

\[ y'_3 = -y_2y_1^2 + \sin t \]

\[ y_1(0) = -1 \]

\[ y_2(0) = 1 \]

\[ y_3(0) = 2 \]

or, in mathematical shorthand,

\[ y' = f(t, y) \]

\[ y(0) = y_0 \]

where

\[ f(t, y) = \begin{pmatrix} f_1(t, y_1, y_2, y_3) \\ f_2(t, y_1, y_2, y_3) \\ f_3(t, y_1, y_2, y_3) \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_2y_1^2 + \sin t \end{pmatrix}, \]

\[ y_0 = \begin{pmatrix} -1 \\ 1 \\ 2 \end{pmatrix}. \]

Any IVP can be transformed in this way into a first order system. For example, we could transform a system of 3 second order ODEs into a system of 6 first order ODEs.

As I mentioned in an earlier chapter, the basic questions a mathematician asks when confronted with a problem are

**Existence:** Is there a solution?

**Uniqueness:** Is there only one solution?

**Algorithms:** How do you find the solution?

The first two questions have been answered in the theory of ODEs:

**Theorem 8.1.** Consider the first-order system

\[ y'(t) = f(t, y(t)) \]

\[ y(t_0) = y_0 \]

(a) (Peano) If \( f \) is continuous, the equation has a solution.

(b) (Picard-Lindelöf) If \( f \) has continuous partial derivatives with respect to the \( y_j \) and is continuous in \( t \), the solution exists and is unique. (There are less restrictive conditions under which the solution is unique, but that is getting technical).
Both of these facts are \textit{local} statements: if \( f \) is continuous (or differentiable) in a neighborhood of the point \((t_0, y_0)\), the solution is only guaranteed to exist (or to be unique) in a small neighborhood of this point.

\textbf{Example:} The equation
\[
y'(t) = 2\sqrt{|y(t)|} \\
y(0) = 0
\]
has the solution \( y(t) = t^2 \). It also has the solution \( y(t) = 0 \). There is actually an infinite number of solutions:
\[
y(t) = \begin{cases} 
0 & \text{for } 0 \leq t \leq a \\
(t - a)^2 & \text{for } t > a 
\end{cases}
\]
where \( a \) is arbitrary. In words: you follow the \( t \)-axis from the origin for a while, and wherever you feel like, you take a branch of a parabola upwards.

Notice that \( f(t, y) = 2\sqrt{|y|} \) is continuous, but is differentiable in \( y \) only when \( y \) is non-zero. Therefore, there is always a solution, but where \( y \) is zero (on the \( t \)-axis), the solution may not be unique. After you leave the \( t \)-axis, there are no more problems. \( \Box \)

\textbf{Example:} Consider
\[
y'(t) = y^2(t) \\
y(0) = 1
\]
This \( f \) is continuously differentiable, so we know that the solution exists and is unique. The solution is not hard to find:
\[
y(t) = \frac{1}{1 - t}
\]
However, this solution “blows up” as \( t \) approaches 1.

The moral is that even if a solution exists locally, it may not exist for all \( t \). \( \Box \)

\subsection{Numerical Methods}

As shown above, we can restrict attention to solving first order systems. To make life even easier, we will only investigate methods for the one-dimensional case
\[
y'(t) = f(t, y(t)) \\
y(t_0) = y_0
\]
It turns out that everything in this chapter that works for the one-dimensional case works the same way for vectors. We will do some vector examples, so you can see how to do that, but all the derivations will be done for the one-dimensional case.

We assume that \( f \) is at least continuous, so that a solution always exists. We ignore the possibilities that the solution is not unique, or that it “blows up” in finite time. These problems must be dealt with by examining the original problem you are trying to solve in more detail.

All of the methods we will consider are \textit{marching methods}. That means that we start at the point \( t_0 \), where the initial condition is given, calculate the approximate solution at some point \( t_1 \), then at \( t_2 \), and so on. We are “marching” along the graph of the solution in the direction of increasing \( t \).

This is in contrast to BVP and PDE, where you usually solve the equation globally. That means you discretize the interval on which you want to solve the equation at \( t_0, \ldots, t_n \) and set up a system of equations to be solved simultaneously.

The classes of methods we will consider are
\begin{itemize}
\item Taylor series methods (including Euler’s method)
\item Runge-Kutta methods
\item multistep methods (predictor-corrector)
\end{itemize}
Other types of methods that are frequently used are \textit{extrapolation methods} and \textit{multi-value methods}. We will not discuss them here.
8.1.3. Errors. In the error analysis, we have to distinguish between the local error and the global error. The local error is the error committed in one step, from $t_{n-1}$ to $t_n$. The global error is the accumulated error from all the previous steps. In order to keep things straight, we have to introduce some notation.

$y(t)$ is the true solution of the IVP. Thus, $y(t_n)$ is the value of the true solution at the point $t_n$.

$y_n$ is the value of the approximate solution at the point $t_n$. The numerical and true solutions agree at $t_0$, so that $y(t_0) = y_0$, but probably nowhere else.

The global error $E_n$ at the point $t_n$ is the difference between true and numerical solution:

$$E_n = y(t_n) - y_n.$$ 

To define the local error $e_n$ at the point $t_n$, we need to ignore everything that happened before $t_{n-1}$, and re-start the ODE at that point. Let $z(t)$ be the true solution of the IVP re-started at the point $t_{n-1}$. That is, $z(t)$ satisfies

$$z'(t) = f(t, z(t))$$
$$z(t_{n-1}) = y_{n-1}.$$ 

Then the local error $e_n$ at the point $t_n$ is defined as

$$e_n = z(t_n) - y_n.$$ 

In order to visualize this, you have to look not only at the solution curve you are after, but at the entire family of all solution curves.

For example, consider the equation $y' = y$. The solution family consists of all curves of the form

$$y(t) = ce^t,$$ 

where $c$ is an arbitrary constant. Every time you take a step, you step off the solution curve you were on, and land on a nearby curve, with a different $c$. The local error is the difference between the last two curves. The global error is the difference between the last curve and the original one (see figure).

The local error is determined by the method you use, and is usually fairly easy to estimate. The global error is determined partly by the method you use and also partly by the geometry of the family of solution curves.
Consider again the test equation

\[ y'(t) = y(t) \]

The solution curves are \( y(t) = ce^t \), which are exponentially diverging. No matter what method you use, your global error will grow exponentially.

In contrast, the equation

\[ y'(t) = -y(t) \]

has the family of solution curves \( y(t) = ce^{-t} \). Here, the curves are going to zero at an exponential rate, and any decent numerical solution should do the same. Global errors actually get smaller.

These test problems are sometimes referred to as **stable** and **unstable** equations, respectively. The names are a little misleading, in my opinion. It is not necessarily bad if the errors grow exponentially, as long as the solution does the same. If you look at the relative errors, they are actually the same in both test problems.

### 8.2. Stable and Unstable Equations, Numerical Methods

This section in the book is covered in my section 8.1.

#### 8.4. Euler’s Method

We want to solve the IVP

\[
\begin{align*}
  y'(t) &= f(t, y(t)) \\
  y(t_0) &= y_0
\end{align*}
\]

numerically at a sequence of points \( t_0, t_1, t_2, \ldots \). To make life easier, we use a constant stepsize \( h \), so that

\[ t_n = t_0 + nh. \]

Assuming that \( y \) is sufficiently often differentiable,

\[ y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2} y''(t_n) + \cdots \]

If we truncate this series at a certain point, we get a Taylor series method. We will talk about them in more detail in section 8.19. Right now, we just want to look at the simplest one.

If we stop the expansion after the second term, we get Euler’s method. It is given by

\[
\begin{align*}
  y_{n+1} &= y_n + hy'_n \\
  &= y_n + hf(t_n, y_n).
\end{align*}
\]

**Example:** Consider the equation

\[ y'(t) = -2ty^2(t) \]

\[ y(0) = 1 \]

with exact solution

\[ y(t) = \frac{1}{1 + t^2} \]

With stepsize \( h = 0.5 \), we find

\[
\begin{align*}
  t_0 &= 0, & y_0 &= 1 \\
  t_1 &= 0.5, & y_1 &= y_0 + hf(t_0, y_0) = 1 + 0.5 (-2 \cdot 0.5^2) = 1 \\
  t_2 &= 1, & y_2 &= y_1 + hf(t_1, y_1) = 1 + 0.5 (-2 \cdot 0.5 \cdot 1^2) = 0.5 \\
  t_3 &= 1.5, & y_3 &= 0.25 \\
  t_4 &= 2, & y_4 &= 0.15625
\end{align*}
\]

The true value is \( y(2) = 0.2 \), so the global error at \( t = 2 \) is \( E_4 = 0.04375 \).
With stepsize $h = 0.25$, we get
\[
\begin{align*}
t_0 &= 0, & y_0 &= 1 \\
t_1 &= 0.25, & y_2 &= 1 \\
t_2 &= 0.5, & y_3 &= 0.875 \\
t_4 &= 1, & y_4 &= 0.508356094 \\
t_8 &= 2, & y_8 &= 0.181628009
\end{align*}
\]
The global error at the same point $t = 2$ is now $E_8 = 0.018371991$.

With stepsize $h = 0.125$, you get
\[
\begin{align*}
t_0 &= 0, & y_0 &= 1 \\
t_8 &= 1, & y_8 &= 0.504548613 \\
t_{16} &= 2, & y_{16} &= 0.191547485
\end{align*}
\]
The global error at $t = 2$ is now $E_{16} = 0.008452515$.

Observe that when we cut the stepsize in half, the global error is also approximately cut in half. We will prove this in the next section.

### 8.5. Accuracy and Stability

My section 8.5 will cover the material in sections 8.5 and 8.6 in the book. Read the book sections on your own.

**8.5.1. Accuracy.** Let us investigate the local and global errors of Euler’s method in detail. The local error is easy to find: we expand $z(t)$ in a Taylor series around the point $t_{n-1}$ and evaluate at $t_n$
\[
z(t_n) = z(t_{n-1}) + h z'(t_{n-1}) + \frac{h^2}{2} z''(t_{n-1}) + \ldots \]
\[= y_{n-1} + h f(t_{n-1}, y_{n-1}) + \frac{h^2}{2} z''(t_{n-1}) + \ldots \]
while
\[
y_n = y_{n-1} + h f(t_{n-1}, y_{n-1}).
\]
Subtracting gives
\[
\text{local error } e_n = z(t_n) - y_n = \frac{h^2}{2} z''(t_{n-1}) + \ldots = O(h^2).
\]
A quick and dirty estimate for the global error is this: assume the global error at the point $t$ is the sum of all local errors. It takes $(t - t_0)/h$ steps to reach $t$, and each step has a local error $O(h^2)$, so the global error is $O(h)$. This is not a mathematical proof, but it is correct reasoning: if the local error is $O(h^{p+1})$, the global error is $O(h^p)$. We saw the same argument for numerical quadrature before.

A method for solving ODEs numerically is said to be of order $p$ if
\[
\begin{align*}
\text{local error} &= O(h^{p+1}), \\
\text{global error} &= O(h^p).
\end{align*}
\]
Remember that the only way to verify the order experimentally is to do the calculation with two different values of $h$. If we cut $h$ in half, then the error should go down by a factor of 2 for a first order method, by a factor of $2^2 = 4$ for a second order method, and so on.

I have tried to do that in the examples. For example, when I cut the stepsize for Euler’s method in half, the final error is approximately cut in half. Unfortunately, this does not work in all cases. The reason is that the geometry of the solution curves also plays a role. However, if you took smaller and smaller $h$, eventually you would be able to observe this behavior in all cases.
A more accurate estimate of the error for Euler’s method is the following.

\[
E_{n+1} = y(t_{n+1}) - y_{n+1}
\]

\[
= [y(t_n) + hf(t_n, y(t_n)) + O(h^2)] - [y_n + f(t_n, y_n)]
\]

\[
= (y(t_n) - y_n) + h [f(t_n, y(t_n)) - f(t_n, y_n)] + O(h^2)
\]

\[
\approx E_n + h \frac{\partial f}{\partial y} [y(t_n) - y_n] + O(h^2)
\]

\[
\approx (1 + h\lambda)E_n + O(h^2),
\]

where we assume that the \(y\)-derivative of \(f\) is approximately equal to some constant \(\lambda\). As a prototype equation, think of

\[y'(t) = \lambda y(t)\]

with the solution \(y(t) = ce^{\lambda t}\).

Let me repeat the final formula:

\[E_{n+1} \approx (1 + h\lambda)E_n + O(h^2)\]

The interpretation of this formula is the following: The new global error \(E_{n+1}\) is the old global error \(E_n\), magnified by a factor of \((1 + h\lambda)\), plus the new local error \(O(h^2)\). If \(|1 + h\lambda| > 1\), the global error grows exponentially. If \(|1 + h\lambda| < 1\), the global error stays small.

The recursive equation for \(E_n\) can be solved in closed form, and again leads to \(E_n = O(h)\).

### 8.5.2. Stability.

A numerical method for solving ODEs is called **stable** (for a particular stepsize \(h\) and for a particular \(\lambda\)) if the numerical solution to the model problem

\[y'(t) = \lambda y(t)\]

remains bounded as \(t \to \infty\).

You should note a few things about this definition:

- Stability is different from accuracy. The error, especially the relative error, may still be large, but at least things don’t blow up.
- We are mostly interested in negative values of \(\lambda\). If \(\lambda\) is positive, the solution itself blows up, and we expect the numerical solution to do the same. Actually, what we are really interested in is complex \(\lambda\) with negative real part. I assume this sort of stuff comes up in EE (oscillating signals with exponentially decaying amplitude).
- Stability depends on the stepsize \(h\) and on \(\lambda\). Thus, a method may be stable for some choices of \(h\) and \(\lambda\) and not for others. In practice, stability always depends only on the product \(h\lambda\), not on \(h\) or \(\lambda\) separately. This makes sense, because a change of unit in \(t\) will affect both \(h\) and \(\lambda\), but \(h\lambda\) will remain the same.

For Euler’s method, the stability behavior is easy to see. Euler’s method for the test problem reduces to

\[y_{n+1} = (1 + h\lambda)y_n\]

The method is stable if \(|1 + h\lambda| \leq 1\), which means for real \(\lambda\) that

\[-2 \leq h\lambda \leq 0.\]

The **region of stability** of a method is defined as the set of all \(h\lambda\) for which the method is stable. The region of stability is a set in the complex plane.

The true solution of the test problem goes to zero if \(\Re(\lambda) < 0\), and blows up if \(\Re(\lambda) > 0\). A numerical method with ideal stability characteristics has the same properties: the numerical solution goes to zero if \(\Re(h\lambda) < 0\), and blows up if \(\Re(h\lambda) > 0\). Thus, the ideal region of stability is the left half plane.

For most methods, the region of stability is a small subset of that. For Euler’s method, the region of stability is a circle (see figure). This means that for a fixed \(\lambda\), the method is stable for some \(h\), but not for others. For some \(\lambda\), there may not be any stable \(h\).

In some cases, the region of stability extends into the right half plane. This is not desirable, since it means that in some cases the numerical solution goes to zero, even though the true solution blows up.
Example: Solve the equation

\[ y' = -y \]
\[ y(0) = 1 \]

by Euler’s method with step sizes \( h = 0.5 \) and \( h = 3 \). The results are

<table>
<thead>
<tr>
<th></th>
<th>( h = 0.5 )</th>
<th>( h = 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_0 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( y_1 )</td>
<td>0.5</td>
<td>-2</td>
</tr>
<tr>
<td>( y_2 )</td>
<td>0.25</td>
<td>4</td>
</tr>
<tr>
<td>( y_3 )</td>
<td>0.125</td>
<td>-8</td>
</tr>
<tr>
<td>( y_4 )</td>
<td>0.0625</td>
<td>16</td>
</tr>
</tbody>
</table>

Clearly, the choice \( h = 0.5 \) is stable, while \( h = 3 \) is not. □

We will discuss the stability behavior of other methods in section 8.9.3.

8.6. Order of an Integration Method

This section in the book is covered in my section 8.5.

8.19. Taylor Series and Runge-Kutta Methods

8.19.1. Taylor Series Methods. The basic idea was mentioned before, when we derived Euler’s method. Assuming that \( y \) is sufficiently often differentiable,

\[ y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2} y''(t_n) + \cdots \]

If we truncate this series at a certain point and replace \( y(t_n) \) by \( y_n \), we get a Taylor series method.

The simplest case is Euler’s method, which we already covered. The next higher Taylor series method uses the first three terms, so

\[ y_{n+1} = y_n + hy'_n + \frac{h^2}{2} y''_n \]
The second derivative is obtained by differentiating the ODE once more.

**Example:** For our standard example, \( f(t, y) = -2ty^2 \), so

\[
y''(t) = \frac{d}{dt} y'(t) = \frac{d}{dt} (-2ty^2) = -2y^2 - 4tyy'
\]

\[
= -2y^2 - 4t(-2ty^2) = -2y^2 + 8t^2y^3.
\]

We get the method

\[
y_{n+1} = y_n + h \left[ -2t_n y_n^2 \right] + \frac{h^2}{2} \left[ -2y_n^2 + 8t_n y_n^3 \right]
\]

We find for \( h = 0.5 \)

\[
y_0 = 1 \\
y_1 = 0.75 \\
y_2 = 0.43359375 \\
y_3 = 0.280106485 \\
y_4 = 0.192250483
\]

with a global error \( E_4 = 0.007749517 \).

For \( h = 0.25 \), we get

\[
y_0 = 1 \\
y_4 = 0.487213029 \\
y_8 = 0.199720953
\]

with a global error \( E_8 = 0.000279047 \).

For \( h = 0.125 \), we get

\[
y_0 = 1 \\
y_8 = 0.4972455756 \\
y_{16} = 0.199994786
\]

with a global error \( E_{16} = 0.000005214 \). \( \square \)

You can observe that this method is more accurate than Euler’s method, and also that the errors decrease faster. This method should be of order 2, even though the error seems to decrease faster than that.

Something else you will observe if you try this on a more complicated problem, or for a higher-order method, is that this differentiation business gets harder pretty fast. This is the main reason that other methods are more popular. Runge-Kutta and multistep methods give the same accuracy as higher order Taylor series methods, without the hassle of taking all these derivatives.

**8.19.2. Runge-Kutta Methods.** Runge-Kutta methods (I will call them RK methods from now on) were developed in an effort to achieve the accuracy of higher order Taylor series methods without having to calculate a lot of derivatives. This is done by inserting intermediate points between \( t_n \) and \( t_{n+1} \).

An *m-stage explicit RK method* is given by

\[
k_1 = hf(t_n, y_n) \\
k_2 = hf(t_n + \alpha_2h, y_n + \beta_{21}k_1) \\
k_3 = hf(t_n + \alpha_3h, y_n + \beta_{31}k_1 + \beta_{32}k_2) \\
\ldots \\
k_m = hf(t_n + \alpha_mh, y_n + \beta_{m1}k_1 + \cdots + \beta_{m,m-1}k_{m-1}) \\
y_{n+1} = y_n + \gamma_1k_1 + \gamma_2k_2 + \cdots + \gamma_mk_m
\]

The \( k_j \) are something like intermediate points we put between \( t_n \) and \( t_{n+1} \). The \( \alpha_j, \beta_{ij}, \gamma_j \) are chosen to produce a method of as high an order as possible, by matching the RK formula with the Taylor series expansion of the true solution.
Note that the calculation of each \( k_j \) involves other known \( k_i \). An implicit RK method also uses \( k_i \) that are not known yet. An implicit method requires the solution of a system of nonlinear equations at each step. In between the two, we have diagonally implicit RK methods, where each \( k_j \) involves itself, but not higher ones. A diagonally implicit method requires the solution of a sequence of one-dimensional nonlinear equations, one for each \( k_j \). We won’t discuss implicit methods.

Consider the general case of two-stage explicit RK methods. We set up

\[
\begin{align*}
    k_1 & = hf(t_n, y_n) \\
    k_2 & = hf(t_n + \alpha h, y_n + \beta k_1) \\
    y_{n+1} & = y_n + \gamma_1 k_1 + \gamma_2 k_2
\end{align*}
\]

We want to determine \( \alpha, \beta, \gamma_1 \) and \( \gamma_2 \) to minimize the local error.

Let us use the shorthand notation

\[
\begin{align*}
    f & = f(t_n, y_n) \\
    f_y & = \frac{\partial f}{\partial y}(t_n, y_n) \\
    f_t & = \frac{\partial f}{\partial t}(t_n, y_n)
\end{align*}
\]

Note that by the chain rule,

\[
\frac{d}{dt} f(t, y(t)) = f_t + f \cdot f_y.
\]

Let \( z(t) \) solve

\[
\begin{align*}
    z'(t) & = f(t, z(t)) \\
    z(t_n) & = y_n
\end{align*}
\]

Do a Taylor series expansion

\[
\begin{align*}
    z(t_{n+1}) & = z(t_n) + hz'(t_n) + \frac{h^2}{2} z''(t_n) + \ldots \\
    & = y_n + hf + \frac{h^2}{2} [f_t + f \cdot f_y] + \ldots
\end{align*}
\]

For the RK formula, we have

\[
\begin{align*}
    k_1 & = hf \\
    k_2 & = hf(t_n + \alpha h, y_n + \beta k_1) = h[f + \alpha h f_t + \beta k_1 f_y + \ldots] \\
    & = hf + \alpha h^2 f_t + \beta h^2 f \cdot f_y + \ldots \\
    y_{n+1} & = y_n + \gamma_1 k_1 + \gamma_2 k_2 \\
    & = y_n + h [\gamma_1 f + \gamma_2 f] + h^2 [\gamma_2 \alpha f_t + \gamma_2 \beta f \cdot f_y] + \ldots
\end{align*}
\]

Thus, the local error is

\[
\begin{align*}
    e_{n+1} & = z(t_{n+1}) - y_{n+1} \\
    & = h [f - \gamma_1 f - \gamma_2 f] + h^2 \left[ \left( \frac{1}{2} f_t + \frac{1}{2} f \cdot f_y \right) - (\gamma_2 \alpha f_t + \gamma_2 \beta f \cdot f_y) \right] + \ldots
\end{align*}
\]

We can eliminate the coefficients of \( h \) and \( h^2 \) by demanding

\[
\begin{align*}
    \gamma_1 + \gamma_2 & = 1 \\
    \gamma_2 \alpha & = 1/2 \\
    \gamma_2 \beta & = 1/2
\end{align*}
\]
These are 3 equations in 4 unknowns, so we have one degree of freedom left. If we choose \( \alpha \) as a parameter, we get

\[
\beta = \alpha \\
\gamma_1 = 1 - 1/(2\alpha) \\
\gamma_2 = 1/(2\alpha)
\]

and the general two-stage explicit RK method:

\[
\begin{align*}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \alpha h, y_n + \alpha k_1) \\
y_{n+1} &= y_n + \left(1 - \frac{1}{2\alpha}\right)k_1 + \frac{1}{2\alpha}k_2
\end{align*}
\]

Each choice of \( \alpha \) leads to a different method. There is no choice of \( \alpha \) which allows you to also match the coefficient of \( h^3 \). Thus, the local error is of order 3, the order of the method is 2 for all \( \alpha \).

For \( \alpha = 0.5 \) you get the midpoint method, for \( \alpha = 1 \) you get the modifier Euler method.

**Example:** Use the midpoint method with \( h = 0.5 \) on our standard example

\[
\begin{align*}
y'(t) &= -2ty^2(t) \\
y(0) &= 1
\end{align*}
\]

We have here \( \alpha = \beta = 0.5, \gamma_1 = 0, \gamma_2 = 1 \). The equations are

\[
\begin{align*}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\
y_{n+1} &= y_n + k_2
\end{align*}
\]

We start with \( t_0 = 0, y_0 = 1 \). The first step is

\[
\begin{align*}
k_1 &= hf(t_0, y_0) = 0 \\
k_2 &= hf(t_0 + h/2, y_0 + k_1/2) = hf(0.25, 1) = -0.25 \\
y_1 &= y_0 + k_2 = 0.75
\end{align*}
\]

The next step is

\[
\begin{align*}
k_1 &= hf(t_1, y_1) = -0.28125 \\
k_2 &= hf(t_1 + h/2, y_1 + k_1/2) = hf(0.75, 0.609375) = -0.278503418 \\
y_2 &= y_1 + k_2 = 0.4714965820
\end{align*}
\]

and so on. We get

\[
\begin{align*}
y_3 &= 0.309188574 \\
y_4 &= 0.2104856219
\end{align*}
\]

The error at the end is \( E_4 = 0.0104856219 \).

**Example:** As an example of how do this with a system, consider the problem

\[
\begin{align*}
y'(t) &= y(t) + 1/z(t) \\
z'(t) &= -t/y(t) \\
y(1) &= e \\
z(1) &= 1/e
\end{align*}
\]

with true solution

\[
y(t) = \begin{pmatrix} y(t) \\ z(t) \end{pmatrix} = \begin{pmatrix} te^t \\ e^{-t} \end{pmatrix}
\]
Again, we use $\alpha = 0.5$, $h = 0.5$.

The first step goes like this:

$$k_1 = hf(t_0, y_0) = 0.5f(1, e, 1/e) = 0.5 \left( e + 1/(1/e) \right) = \left( 2.7182818285, -0.1839397206 \right)$$

$$k_2 = hf(t_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1) = 0.5f(1.25, e + \frac{1}{2}2.718 \ldots , 1/e - \frac{1}{2}0.183 \ldots ) = \left( 3.8508992570, -0.1532831005 \right)$$

$$y_1 = y_0 + k_2 = \left( e \frac{1}{1/e} \right) + k_2 = \left( 6.5691810854, 0.2145963407 \right)$$

The true values at $t = 1.5$ are 6.722533605 and 0.223130160.

In later steps, we get

$$k_1 = hf(t_1, y_1) = \left( 5.6145463957, -0.1141694817 \right)$$

$$k_2 = hf(t_1 + \frac{1}{2}h, y_1 + \frac{1}{2}k_1) = \left( 7.8625965252, -0.0933188574 \right)$$

$$y_2 = y_1 + k_2 = \left( 14.4317776107, 0.1212774833 \right)$$

$$y_3 = \left( 30.2538910932, 0.0653104260 \right)$$

$$y_4 = \left( 62.2742345985, 0.0322934446 \right)$$

Higher order RK methods are derived in a similar way, except that the details can get quite tricky.

The current record, according to the Guinness book of world records, is held by A.R. Curtis, for an 18 stage formula of order 10 in 1975. (Thanks to Roger Alexander for this information).

Make sure you don’t confuse “order” with “number of stages”. A RK method of order $n$ is one in which the local error is of order $O(h^{n+1})$. An $n$-stage method is one that uses $k_1$ through $k_n$. The two happen to be the same for $n = 2, n = 3$ and $n = 4$, but not in general.

The most popular method is probably the four-stage fourth-order RK with constants $\alpha_2 = \alpha_3 = 1/2$, $\alpha_4 = 1$, $\beta_{21} = \beta_{32} = 1/2$, $\beta_{31} = \beta_{41} = \beta_{42} = 0$, $\beta_{43} = 1$, $\gamma_1 = \gamma_4 = 1/6$, $\gamma_2 = \gamma_3 = 1/3$. Written out, this gives you

$$k_1 = hf(t_n, y_n)$$

$$k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$$

$$k_3 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)$$

$$k_4 = hf(t_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

This method is popular probably because it is nice and symmetric, and several $\beta$ are zero. When somebody talks about “the” Runge-Kutta method, they probably mean this one.

Example: Again, our example

$$y'(t) = -2ty^2(t)$$

$$y(0) = 1$$
We again use $h = 0.5$, $t_0 = 0$, $y_0 = 1$. The first complete step is

\[
\begin{align*}
k_1 &= 0 \\
k_2 &= -0.25 \\
k_3 &= -0.19140625 \\
k_4 &= -0.3269119263 \\
y_1 &= 0.7983792623
\end{align*}
\]

and the following values

\[
\begin{align*}
y_2 &= 0.4997015229 \\
y_3 &= 0.3081669121 \\
y_4 &= 0.2004056722
\end{align*}
\]

so the global error is $E_4 = 0.0004056722$.

If we repeat the process with $h = 0.25$, we get

\[
\begin{align*}
y_0 &= 1 \\
y_1 &= 0.941154013 \\
\cdots \\
y_4 &= 0.5000135525 \\
\cdots \\
y_8 &= 0.2000271443
\end{align*}
\]

with a global error $E_8 = 0.0000271443$.

The ratio of errors is $14.9 \approx 16$, confirming that this is indeed a fourth order method. \qed

We will leave even simple examples of this method for systems to the computers.

### 8.8. Implicit Methods

An explicit method calculates the numerical solution at $t_{n+1}$ from values at $t_n$ or earlier. An implicit method uses also values at $t_{n+1}$. Implicit methods are may require that you solve a nonlinear equation at every step, but they are more accurate and more stable.

To derive a couple of simple implicit methods, rewrite

\[
y' = f(t, y(t))
\]

in the form

\[
y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t)) \, dt,
\]

and replace the integral by a simple quadrature formula. With the notation

\[
f_n = f(t_n, y_n)
\]

we get the following methods:

Approximating the integral by a rectangle, using the value at the left endpoint, we get

\[
y_{n+1} = y_n + hf_n,
\]

which is Euler’s method.

Using the right endpoint,

\[
y_{n+1} = y_n + hf_{n+1},
\]

which is called the backward Euler method.

Using the trapezoidal rule,

\[
y_{n+1} = y_n + \frac{h}{2} \left[ f_n + f_{n+1} \right],
\]

which is called the trapezoidal method.
The local error is the error in the integration formula, so backward Euler has local error $O(h^2)$ and is a first-order method, and the trapezoidal rule has local error $O(h^3)$ and is a second-order method.

How do we actually take a step in an implicit method? If the equation is linear, we can solve for $y_{n+1}$ by hand (see example below). Any method from chapter 7 (for a single equation) or chapter 9 (for systems of equations) can be used, but that is only necessary if the equation is very badly behaved. Most of the time, we use the predictor-corrector technique outlined in the next section.

**Example:** Consider the test equation

$$y' = -y$$

$$y(0) = 1$$

Using backward Euler, we get

$$y_{n+1} = y_n - hy_{n+1},$$

or

$$y_{n+1} = \frac{1}{1 + h} y_n$$

Obviously, this is stable for any $h$, since $y_n \to 0$ as $n \to \infty$. 

### 8.9. Multi-step Methods

My section 8.9 will cover the material in sections 8.9 through 8.11 in the book. Read the book sections on your own.

A $k$-step method is defined as a method which uses information from the last $k$ values of $y$ and $f$ to find the next value. All methods discussed so far have been one-step methods.

The general form of an explicit $k$-step method is

$$y_{n+1} = \alpha_1 y_n + \alpha_2 y_{n-1} + \cdots + \alpha_k y_{n-k+1} + h [\beta_1 f_n + \beta_2 f_{n-1} + \cdots + \beta_k f_{n-k+1}]$$

The general form of an implicit $k$-step method is

$$y_{n+1} = \alpha_0 y_{n+1} + \alpha_1 y_n + \cdots + \alpha_k y_{n-k+1} + h [\beta_0 f_{n+1} + \beta_1 f_n + \cdots + \beta_k f_{n-k+1}]$$

### 8.9.1. The Predictor-Corrector Scheme

As mentioned above, implicit methods are usually both more accurate and more stable than explicit methods, but they are harder to apply. To understand the so-called predictor-corrector approach, we need to take a little detour.

Suppose we want to solve an equation of the form

$$x = g(x),$$

where $g(x)$ is some function. A solution of this equation is called a fixed point of $g$. We could try to set up an iteration of the form

$$x_0 = \text{initial guess}$$

$$x_1 = g(x_0)$$

$$x_2 = g(x_1)$$

$$\ldots$$

and so on, and hope that this converges to a fixed point. This method is called fixed point iteration. Sometimes it works, sometimes it doesn’t.

From the material in chapter 7 it follows that for small enough $h$, fixed point iteration will work for solving the equations in implicit $k$-step methods.

The setup for a predictor-corrector method is this:

- We use an explicit method to get a good guess for $y_{n+1}$. This is called the predictor step.
- We calculate $f_{n+1} = f(t_{n+1}, y_{n+1})$. This is the evaluation step.
- We use an implicit method with these guesses for $y_{n+1}$ and $f_{n+1}$ on the right-hand side. This is the corrector step (one step of fixed point iteration, really). This gives us a new and improved $y_{n+1}$.
- We calculate a new and improved $f_{n+1}$ with our new and improved $y_{n+1}$ from the corrector step.
We could repeat the last two steps over and over again until we get convergence. We know that this will happen for small enough $h$. In practice, this is not efficient. Even if we found the exact $y_{n+1}$, this would still not be the true $y(t_{n+1})$. There are two components of error: one from replacing the ODE with a numerical method, and one from the fixed point iteration. The best place to stop is when both are about equal. One corrector step is usually sufficient for that. If you need more accuracy, it is more efficient to choose a smaller stepsize.

The method outlined above is referred to as a PECE scheme (predict-evaluate-correct-evaluate). Other schemes are possible, like $PEC$, $P(EC)^2E$, and so on. PECE is the most commonly used.

**Example:** Apply the predictor-corrector scheme to our standard example

$$y'(t) = -2ty^2(t)$$
$$y(0) = 1$$

using Euler’s method as a predictor, the trapezoidal method as a corrector. For stepsize $h = 0.5$, we find

$t_0 = 0, \quad y_0 = 1$
$f_0 = -2t_0y_0^2 = 0$

$t_1 = 0.5, \quad y_1 = y_0 + hf_0 = 1$ (P, Euler’s method)
$f_1 = -2t_1y_1^2 = -1$ (E)
$y_1 = y_0 + \frac{h}{2}[f_0 + f_1] = 0.75$ (C, trapezoidal method)
$f_1 = -2t_1y_1^2 = -0.5625$ (E)

$t_2 = 1, \quad y_2 = y_1 + hf_1 = 0.46875$
$f_2 = -2t_2y_2^2 = -0.439453125$
$y_2 = y_1 + \frac{h}{2}[f_1 + f_2] = 0.499511719$
$f_2 = -2t_2y_2^2 = -0.499023914$

and so on. □

### 8.9.2. The Adams-Bashforth-Moulton Methods

The most commonly used multistep methods are *Adams-Bashforth* (predictor) and *Adams-Moulton* (corrector). Both are based on the same idea as the methods in section 8.8.

Replace the ODE

$$y'(t) = f(t, y(t))$$

by

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t, y(t)) \, dt,$$

replace $f$ by an interpolating polynomial and integrate.

**Example:** Let us derive the two-step Adams-Bashforth predictor method in detail. Since $k = 2$, we use the last 2 values of $f$, namely $f_{n-1}$ and $f_n$, and put a polynomial through the points $(t_{n-1}, f_{n-1})$ and $(t_n, f_n)$. In this case, the polynomial is a straight line. We get

$$f(t, y(t)) \approx \frac{t - t_n}{t_{n-1} - t_n}f_{n-1} + \frac{t - t_{n-1}}{t_n - t_{n-1}}f_n$$

$$\int_{t_n}^{t_{n+1}} f(t, y(t)) \, dt \approx \int_{t_n}^{t_{n+1}} \left[ \frac{t - t_n}{t_{n-1} - t_n}f_{n-1} + \frac{t - t_{n-1}}{t_n - t_{n-1}}f_n \right] \, dt$$

$$= \left[ \frac{(t - t_n)^2}{2}f_{n-1} + \frac{(t - t_{n-1})^2}{2}f_n \right]_{t_n}^{t_{n+1}}$$

$$= \frac{3h}{2}f_n - \frac{h}{2}f_{n-1}.$$

The method is therefore

$$y_{n+1} = y_n + h \left[ \frac{3}{2}f_n - \frac{1}{2}f_{n-1} \right]$$
The most commonly used methods are 4-step Adams-Bashforth as a predictor
\[ y_{n+1} = y_n + \frac{h}{24} [55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}] \]
and 3-step Adams-Moulton as a corrector
\[ y_{n+1} = y_n + \frac{h}{24} [9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}] \]
Both of them are of order 4.

In general, the \( k \)-step Adams-Bashforth method is of order \( k \), the \( k \)-step Adams-Moulton method is of order \( k + 1 \). You can convince yourself of that as follows: The \( k \)-step Adams-Bashforth method uses interpolation at \( k \) points. The interpolation error is \( O(h^k) \). When we integrate, we pick up another factor of \( h \) (from the interval width). Thus, the local error is \( O(h^{k+1}) \), and the global error is \( O(h^k) \). The \( k \)-step Adams-Moulton method uses \( (k + 1) \) points. By the same reasoning, it is of order \( (k + 1) \).

**Example:** We use our standard example again,
\[ y'(t) = -2ty^2(t) \]
\[ y(0) = 1 \]
We use \( h = 0.25 \), the fourth order Adams-Bashforth-Moulton method (4-step predictor, 3-step corrector).

startup. To get going, we need values for the first 4 points. I used the Runge-Kutta method to get those.
\[
\begin{align*}
y_0 &= 1, & f_0 &= 0 \\
y_1 &= 0.9411540130, & f_1 &= -0.442885438 \\
y_2 &= 0.7999481032, & f_2 &= -0.639916968 \\
y_3 &= 0.6399738841, & f_3 &= -0.614349858 
\end{align*}
\]
Then we get the predictor step
\[ y_4 = y_3 + \frac{h}{24} [55f_3 - 59f_2 + 37f_1 - 9f_0] = 0.5105894849, \]
the evaluation step
\[ f_4 = -0.521403244, \]
the corrector step
\[ y_4 = y_3 + \frac{h}{24} [9f_4 + 19f_3 - 5f_2 + f_1] = 0.498217826 \]
and finally another evaluation step
\[ f_4 = -0.496442097 \]
The next values are
\[
\begin{align*}
y_5 &= 0.3862529501 \quad \text{(predictor)} \\
&= 0.390328576 \quad \text{(corrector)} \\
y_6 &= 0.3090162813 \quad \text{(corrector)} \\
y_7 &= 0.2472511935 \quad \text{(corrector)} \\
y_8 &= 0.2007863546 \quad \text{(corrector)}
\end{align*}
\]
The global error at the end is \( E_8 \approx 0.000786 \).
8.9.3. **Stability.** Recall the general form of a $k$-step method

$$y_{n+1} = \alpha_0 y_{n+1} + \alpha_1 y_n + \cdots + \alpha_k y_{n-k+1} + h \left[ \beta_0 f_{n+1} + \beta_1 f_n + \cdots + \beta_k f_{n-k+1} \right].$$

If $\alpha_0 = \beta_0 = 0$, the method is explicit, otherwise implicit.

If we apply this method to the test problem

$$y' = \lambda y,$$

we get

$$y_{n+1} = (\alpha_0 + h\lambda \beta_0) y_{n+1} + (\alpha_1 + h\lambda \beta_1) y_n + (\alpha_2 + h\lambda \beta_2) y_{n-1} + \cdots + (\alpha_k + h\lambda \beta_k) y_{n-k+1}.$$

This is a so-called **difference equation**. Difference equations are like discrete versions of differential equations, in many respects, and have a well-developed theory. The solutions of difference equations are infinite sequences $\{y_j\}$.

Define the **characteristic equation** of the difference equation as

$$r^k = (\alpha_0 + h\lambda \beta_0)r^k + (\alpha_1 + h\lambda \beta_1)r^{k-1} + (\alpha_2 + h\lambda \beta_2)r^{k-2} + \cdots + (\alpha_k + h\lambda \beta_k)r^0$$

and the **reduced characteristic equation** as the characteristic equation with $h\lambda = 0$

$$r^k = \alpha_0 r^k + \alpha_1 r^{k-1} + \alpha_2 + r^{k-2} + \cdots + \alpha_k r^0.$$

The theory of difference equations says that a $k$th order difference equation has $k$ linearly independent solutions $y_j$, $j = 1, \ldots, k$ of the form

$$(y_j)_n = r_j^n$$

where $r_j$ is a root of the characteristic equation. The general solution is

$$y = c_1 y_1 + \cdots + c_k y_k,$$

where the $c_j$ are arbitrary constants. This is very similar to the theory of constant coefficient ODEs.

**Example:** Euler’s method is

$$y_{n+1} = y_n + h f(t_n, y_n).$$

For the sample problem, this becomes

$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n.$$

The characteristic equation is

$$r = 1 + h\lambda,$$

the reduced characteristic equation is

$$r = 1.$$

**Example:** The trapezoidal method

$$y_{n+1} = y_n + \frac{h}{2} [y_n + y_{n+1}]$$

has characteristic equation

$$r = 1 + \frac{h\lambda}{2}(1 + r)$$

and reduced characteristic equation

$$r = 1.$$

**Example:** The second order Runge-Kutta method with $\alpha = 0.5$ is given by

$$k_1 = h f(t_n, y_n)$$
$$k_2 = h f(t_n + h/2, y_n + k_1/2)$$
$$y_{n+1} = y_n + k_2$$
For the sample problem, this becomes

\[ \begin{align*}
  k_1 &= h\lambda y_n \\
  k_2 &= h\lambda (y_n + k_1/2) \\
  &= h\lambda y_n + (h\lambda)^2 y_n/2 \\
  y_{n+1} &= y_n + k_2 = y_n + h\lambda y_n + (h\lambda)^2 y_n/2
\end{align*} \]

The characteristic equation is

\[ r = 1 + h\lambda + (h\lambda)^2/2, \]

the reduced characteristic equation is

\[ r = 1. \]

\[ \square \]

**Example:** For the 4-step Adams-Bashforth method

\[ y_{n+1} = y_n + \frac{h}{24} [55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}] \]

we have for the sample problem

\[ y_{n+1} = y_n + \frac{h\lambda}{24} [55y_n - 59y_{n-1} + 37y_{n-2} - 9y_{n-3}] \]

The characteristic equation is

\[ r^4 = r^3 + \frac{h\lambda}{24} [55r^3 - 59r^2 + 37r - 9], \]

the reduced characteristic equation is

\[ r^4 = r^3. \]

\[ \square \]

Theory now tells us the following facts:

- The method is stable at stepsize \( h \) for a given \( \lambda \) if all solutions of the characteristic equation have absolute value less than or equal to 1, and all multiple solutions have absolute value less than 1. The method is unstable if one or more of the solutions have absolute value greater than 1. Note that we allow complex \( \lambda \), so this is the complex absolute value.
- The reduced characteristic equation always has the solution \( r = 1 \). The method is stable for small values of \( h\lambda \) if all the other solutions have absolute value less than 1.

Again, the region of stability of a method is defined as the set of all points \((h\lambda)\) in the complex plane for which the method is stable.

Let us consider our examples again:

**Example:** Euler’s method has reduced characteristic equation

\[ r = 1 \]

This equation has the solution \( r = 1 \), as predicted, and no others. Euler’s method is therefore stable for small values of \( h\lambda \) (we already knew that).

The full characteristic equation is

\[ r = 1 + h\lambda \]

It has exactly one solution, namely \( r = 1 + h\lambda \), so Euler’s method is stable whenever

\[ |1 + h\lambda| < 1. \]

We knew that, too. The region of stability is a circle of radius 1 around the point (-1). The real part of it is the interval \([-2, 0]\).

**Example:** The trapezoidal method has the reduced characteristic equation

\[ r = 1, \]
so it is stable for small values of $h\lambda$. The full characteristic equation

$$r = 1 + \frac{h\lambda}{2} (1 + r)$$

has solution

$$r = \frac{1 + h\lambda/2}{1 - h\lambda/2}$$

It is stable whenever

$$\left| \frac{1 + h\lambda/2}{1 - h\lambda/2} \right| < 1.$$  

It can be shown that this is exactly the left half plane.  

**Example:** The second order RK method has the same reduced characteristic equation

$$r = 1$$

and is also stable for small $h\lambda$. (Remark: any one-step method has reduced characteristic equation $r = 1$. Think about it).

The full characteristic equation is

$$r = 1 + h\lambda + (h\lambda)^2/2$$

The method is stable whenever

$$|1 + h\lambda + (h\lambda)/2| < 1.$$  

The region of stability is harder to find here, but looks similar to the circle we got for Euler’s method. The real part of is again the interval $[-2, 0]$ (see picture).

**Example:** 4-step Adams-Bashforth has the reduced characteristic equation

$$r^4 = r^3.$$  

It has one solution $r = 1$, as predicted, and the triple solution $r = 0$. All solutions other than $r = 1$ have absolute values less than 1, so this method is again stable for small $h\lambda$.

The characteristic equation itself is a fourth degree polynomial

$$r^4 = r^3 + \frac{h\lambda}{24} [55r^3 - 59r^2 + 37r - 9].$$

For each complex $h\lambda$, we need to find the four (complex) solutions, take the absolute value of each and decide whether they are all less than or equal to 1 or not. Then we can decide whether this particular $h\lambda$ belongs in the region of stability or not.

Fortunately, somebody has done this before (see picture).

Some regions of stability are pictured below. Notice that sometimes the region sticks out into the right half plane a little. What that means in practice is that the method is a little too stable. The true solution is exponentially growing, while the numerical solution goes to zero.

Also notice that all of these methods have fairly small regions of stability, only up to -2 or -3.

**8.10. Order and Error of a Multi-step Method**

This section in the book is covered in my section 8.9.

**8.11. Stability for Multi-step Methods**

This section in the book is covered in my section 8.9.

**8.7. Subroutine SDRIV2**

Read this section for yourself.
CHAPTER 9

Optimization and Nonlinear Least Squares

9.1. Introduction

We will cover the material in sections 9.2 and 9.4. Read sections 9.3 and 9.5 by yourself.

As already outlined in chapter 7, the goal of the present chapter is to find a local minimum of a function $F(x)$ (a single function of several variables). This problem has many connections to the problem of finding a zero of a nonlinear function.

9.1.1. Accuracy. One “feature” of minimum-finding subroutines is that they can only find the solution to lower precision. Consider this in one dimension:

$$f(x) \approx f(\bar{x}) + f'(\bar{x})(x - \bar{x}) + \frac{1}{2}f''(\bar{x})(x - \bar{x})^2 + \ldots$$

If $\bar{x}$ is a local minimum, then $f'(\bar{x}) = 0$, so

$$f(x) - f(\bar{x}) \approx C(x - \bar{x})^2$$

In order to locate the minimum, the relative change between $f(x)$ and $f(\bar{x})$ must be at least the machine $\epsilon$. But then the relative change between $x$ and $\bar{x}$ can only be $\sqrt{\epsilon}$, assuming $c \approx 1$. In single precision arithmetic, $\epsilon \approx 10^{-7}$, so $\sqrt{\epsilon} \approx 10^{-3}$ or $10^{-4}$. This means we can only find a minimum to about 3 or 4 decimal places.

This becomes apparent when you solve the same problem in several ways. The answers you get only agree to about 3 decimals. Watch for this in the examples below.

9.2. One-Dimensional Optimization

We want to find a local minimum of a function $F(x)$.

9.2.1. Golden Section Search. The method outlined in this section is similar to the bisection method in spirit: we start with an interval we think contains a minimum, and keep making the interval smaller. It is slow but reliable, and it requires no derivatives.

This method only works reliably under the assumption that we have an interval where the function is unimodal. This means that there is exactly one minimum $\bar{x}$ in the interval $[a, b]$ we are currently considering, and $F$ is decreasing in $[a, \bar{x}]$ and increasing in $(\bar{x}, b]$. (See figure 9.4 in the book).

We start with a given interval $[a, b]$. Now we pick two points $a < x_1 < x_2 < b$ inside the interval. Assume $F(x_1) > F(x_2)$. Then $F$ is decreasing at least part of the way between $x_1$ and $x_2$. Since $F$ is always increasing to the right of $\bar{x}$, $\bar{x}$ cannot be any smaller than $x_1$. Thus, the minimum $\bar{x}$ is in $[x_1, b]$. Likewise, if $F(x_1) < F(x_2)$, then $\bar{x} < x_2$, and the minimum must be in $[a, x_2]$.

What is the optimal way to choose the subdivision points? If you know how many steps you want to take, the best method is called Fibonacci search. If you don’t know, the best method is the golden section search. Look at the book for more details on the derivation.

In golden section search, we put one subdivision point at $(\sqrt{5} - 1)/2 \approx 0.6180$ of the interval width. This number is called the golden ratio. The other point goes symmetrically, at about 0.3820 of the interval width. With this placement, we can always re-use one of the points at the next step. Remember that the goal is to minimize the number of function evaluations necessary.
At every step, the interval becomes smaller by a factor of 0.618, so this method is linearly convergent.

**Example:** Find a local minimum of the function

\[ F(x) = e^x + \frac{1}{x} \]

With the initial interval \([0.1, 1]\), we get successive intervals

\[
\begin{align*}
[0.1000000, 1.000000] \\
[0.4437694, 1.000000] \\
[0.4437694, 0.7875388] \\
[0.5750777, 0.7875388] \\
[0.6562306, 0.7875388] \\
... \\
[0.7037065, 0.7037098] \\
[0.7037065, 0.7037086] \\
[0.7037073, 0.7037086] \\
[0.7037078, 0.7037086]
\end{align*}
\]

with a best guess of \(x = 0.7037081\), \(F(x) = 3.442277\).

The true answer to 10 decimals is \(x = 0.703467422\), \(F(x) = 3.44227294\). Notice how \(F(x)\) is accurate to 7 decimals, while \(x\) is only accurate to 3 decimals. \(\square\)

9.2.2. **Newton’s Method.** We convert the minimization problem to a zero-finding problem. Thus, we want to solve

\[ F'(x) = 0. \]

If we use Newton’s method (for finding zeros of a function) for that, we get the iteration

\[ x_{n+1} = x_n - \frac{F'(x_n)}{F''(x_n)}. \]

This is also called Newton’s method (for optimization). We now have Newton’s method for finding zeros and Newton’s method for optimization, which is based on the first one. Try not to confuse the two.

**Remark:** Of course, there is also a Newton’s method for polynomial interpolation, which has nothing to do with the present discussion at all.

**Example:** Minimize the function

\[ F(x) = e^x + \frac{1}{x} \]

We find with initial guess \(x_0 = 1\),

\[
\begin{align*}
x_0 &= 1.0 & F(x_0) &= 3.718282 \\
x_1 &= 0.6358247 & F(x_1) &= 3.461339 \\
x_2 &= 0.6963254 & F(x_2) &= 3.442477 \\
x_3 &= 0.7033929 & F(x_3) &= 3.442277 \\
x_4 &= 0.7034674 & F(x_4) &= 3.442277 \\
\end{align*}
\]

and no further changes. This time, the answer is correct all the way, but that is because this is a zero-finding problem now, not a minimization problem any more. I can only get higher accuracy because I know the exact derivative of \(F\).

Note that \(F(x_3)\) already equals \(F(x_4)\), even though \(x_3\) and \(x_4\) are different. \(\square\)

Newton’s method for optimization has the same advantages and disadvantages as Newton’s method for finding zeros:

- It converges very fast (quadratically).
• It may run into problems (infinite loops, division by zero, etc.)
• Its main disadvantage is that it requires derivatives. Two derivatives now, even.

9.2.3. Successive Parabolic Interpolation. There is another way to derive Newton’s method which suggests a new method.

Recall that Newton’s method for finding zeros can be interpreted as follows: $x_0$ is an initial guess. We compute the tangent to the graph of $f(x)$ at the point $(x_0, f(x_0))$ and find the zero of the tangent. That is our $x_1$. Then we put a tangent at the point $(x_1, f(x_1))$, and so on. The equation of the tangent at $(x_i, f(x_i))$ is a truncated Taylor series at $x_i$

$$t(x) = f(x_i) + f'(x_i)(x - x_i)$$

If we want to avoid derivatives, we can replace the tangent by a secant through $x_{i-1}$ and $x_i$, and that leads to the secant method. The secant method converges a little more slowly than Newton’s method, but does not require derivatives.

A similar interpretation is possible for Newton’s method for optimization. Here, we need to go one step further in the Taylor series: We replace $F(x)$ by the tangent parabola at $(x_i, F(x_i))$

$$p(x) = F(x_i) + F'(x_i)(x - x_i) + \frac{F''(x_i)}{2}(x - x_i)^2$$

and calculate the minimum of the parabola. The result is the Newton’s method for minimization

$$x_{n+1} = x_n - \frac{F'(x_n)}{F''(x_n)}.$$

To avoid derivatives, we could put a parabola through the last three points $x_j$ and find the minimum of the parabola. Essentially, we are replacing the derivatives in Newton’s formula by difference quotients. The resulting method is called **successive parabolic interpolation**.

It has properties similar to the secant method: it does not converge with second order, like Newton’s method, but it converges with better than first order.

The formulas are sort of messy, so we will not go into the details.

9.2.4. Summary of One-Dimensional Methods. We have considered three methods: Golden section search, Newton’s method and Successive parabolic interpolation. They correspond more or less to Bisection, Newton’s method and Secant method for the problem of finding zeros.

That means

• Golden Section Search, like Bisection, is slow but reliable and will converge in a certain number of steps which can be calculated in advance.
• Newton’s method is fast and is the method of choice if you have a sufficient number of derivatives available.
• Successive Parabolic Interpolation is a form of Newton’s method with derivatives replaced by difference quotients. It is slower than Newton’s method, but still converges quite fast.

The best general purpose method is a combination of Golden Section and Successive Parabolic Interpolation: Use Parabolic Interpolation most of the time. If it gives crazy answers or tries to divide by zero, use Golden Section for a step or two. This is what routine `FMIN` does.

9.3. Subroutine `FMIN`

Read this section on your own.

9.4. Optimization in Many Dimensions

Most higher-dimensional methods follow the same scheme:

• Pick an initial guess $x_0$ and an initial search direction $u_0$. 

• Let $x_1$ be the minimum of $F$ in direction $u_0$, starting at $x_0$. Mathematically speaking, we define

$$h(\lambda) = F(x_0 + \lambda u_0)$$

so $h$ is a function of the single variable $\lambda$. Think of it as a one-dimensional slice through the surface $F$. We find the $\lambda$ for which this function is a minimum, and then

$$x_1 = x_0 + \lambda u_0.$$ 

This step is called the line search. Here is where we need one of the one-dimensional minimizers, like the golden section search.

• Pick a new search direction $u_1$.
• Let $x_2$ be the minimum of $F$ in direction $u_1$, starting at $x_1$.

and so on.

Collectively, these methods are known as direction set methods. Different methods are distinguished mainly by how the search directions $u_i$ are picked. The choice of one-dimensional minimizer makes a minor difference, and there are other tricks you can use (for example, you may overshoot the one-dimensional minimum on purpose at every step). We will ignore all that and just concentrate on the choice of $u_i$. That is where the biggest difference between the methods lies.

First we need to recall a couple more things from calculus.

**9.4.1. Surfaces and Level Sets.** We are given a function $F(x)$, where $x$ is an $n$-vector. We can think of the graph of $F$ as a surface in $(n+1)$-dimensional space. A level surface of $F$ is the set of all points

$$\{x : F(x) = c\}$$

where $c$ is some constant. For every $c$, we get a different level curve. (The isoclines on a survey map are level curves, for example). Near a local minimum, the level surfaces always look like nested ellipsoids.

The gradient of $F$ is the vector

$$\nabla F = \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{pmatrix}$$

The gradient indicates the direction in which $F$ changes the most. Along the level surfaces, $F$ does not change at all. The gradient is always perpendicular to the level surfaces.

When we are doing our line searches, $F$ is increasing or decreasing as long as we intersect the level surfaces at an angle. $F$ has a stationary point when we are tangent to a level surface. The place where the line minimum occurs is always a stationary point. Thus

$$u_{i-1} \cdot \nabla F(x_i) = 0.$$ 

We will need this formula later.

**9.4.2. Coordinate Search.** I don’t know if this method even has a name, so I made one up. A very simple-minded approach would be to use the coordinate directions as search directions:

$$u_0 = e_1$$

$$u_1 = e_2$$

$$\ldots$$

$$u_{n-1} = e_n$$

$$u_n = e_1$$

$$\ldots$$,

where

$$e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$
and so on. In other words: we first find a minimum in the $e_1$ direction, while keeping all the other coordinates fixed. Then we find a minimum in the $e_2$ direction, and so on.

This will work, but will be slow.

**Example:** The function

$$F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4$$

has a local minimum at $(1, 1)$. Using $x_0 = (1.25, 1.25)$ and coordinate search, we find

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1.2500000, 1.2500000)</td>
<td>0.0039062500</td>
</tr>
<tr>
<td>$(1.2469850, 1.2500000)</td>
<td>0.0038120960</td>
</tr>
<tr>
<td>$(1.2469850, 1.2469890)</td>
<td>0.0037212200</td>
</tr>
<tr>
<td>$(1.2440800, 1.2469890)</td>
<td>0.0036337960</td>
</tr>
<tr>
<td>$(1.2440800, 1.2440790)</td>
<td>0.0035491690</td>
</tr>
<tr>
<td>$(1.2412700, 1.2440790)</td>
<td>0.0034674370</td>
</tr>
<tr>
<td>$(1.2385530, 1.2412690)</td>
<td>0.0033885690</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$(1.1005000, 1.1004990)</td>
<td>0.0001020135</td>
</tr>
<tr>
<td>$(1.1002980, 1.1004990)</td>
<td>0.0001016024</td>
</tr>
<tr>
<td>$(1.1002980, 1.1002980)</td>
<td>0.0001011975</td>
</tr>
<tr>
<td>$(1.1000970, 1.1000970)</td>
<td>0.0001007932</td>
</tr>
<tr>
<td>$(1.0998970, 1.1000970)</td>
<td>0.0001003888</td>
</tr>
<tr>
<td>$(1.0998970, 1.1000970)</td>
<td>0.0000999874</td>
</tr>
</tbody>
</table>

The gap represents about 400 iterations. As you can see, it works, but the method is awfully slow. □

**9.4.3. Steepest Descent.** A very natural idea is to go in the direction in which $F$ decreases the fastest, which is the negative gradient:

$$u_i = -\nabla F(x_i)$$

**Example:** With the same function

$$F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4$$

and initial guess $x_0 = (1.25, 1.25)$, we get

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1.2500000, 1.2500000)</td>
<td>0.0039062500</td>
<td>$(-0.0062500, 0.0000000)$</td>
</tr>
<tr>
<td>$(1.2469840, 1.2500000)</td>
<td>0.0038120960</td>
<td>$(0.0000498, -0.0603151)$</td>
</tr>
<tr>
<td>$(1.2469870, 1.2469900)</td>
<td>0.0037212990</td>
<td>$(-0.0602052, -0.0000620)$</td>
</tr>
<tr>
<td>$(1.2440780, 1.2469870)</td>
<td>0.0036336850</td>
<td>$(0.0000185, -0.0581813)$</td>
</tr>
<tr>
<td>$(1.2412700, 1.2440790)</td>
<td>0.0035491140</td>
<td>$(-0.0581253, -0.0000381)$</td>
</tr>
<tr>
<td>$(1.2412620, 1.2440790)</td>
<td>0.0034674370</td>
<td>$(0.0001655, -0.0563383)$</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

and so on, for hundreds of iterations. (It will eventually converge).
On the other hand, if we use the initial guess $x_0 = (1.25, 1.5)$, we get

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.5000000)</td>
<td>0.6289063000</td>
<td>(4.9375000, -5.0000000)</td>
</tr>
<tr>
<td>(1.3716800, 1.3767800)</td>
<td>0.0193443900</td>
<td>(-0.1033715, -0.1020122)</td>
</tr>
<tr>
<td>(1.0836200, 1.0925090)</td>
<td>0.0008389865</td>
<td>(0.1754356, -0.1777744)</td>
</tr>
<tr>
<td>(1.0880010, 1.0880700)</td>
<td>0.0000600194</td>
<td>(-0.0013527, -0.0013733)</td>
</tr>
<tr>
<td>(1.0375130, 1.0368130)</td>
<td>0.0000068819</td>
<td>(-0.0412135, 0.0410023)</td>
</tr>
<tr>
<td>(1.0371580, 1.0371630)</td>
<td>0.0000019065</td>
<td>(-0.0001051, -0.0001001)</td>
</tr>
<tr>
<td>(1.0337760, 1.0339400)</td>
<td>0.0000015701</td>
<td>(0.0031241, -0.0032738)</td>
</tr>
<tr>
<td>(1.0338540, 1.0338580)</td>
<td>0.0000013136</td>
<td>(-0.0000789, -0.0000763)</td>
</tr>
</tbody>
</table>

... ... ...

and we get much closer to the true solution before things bog down. □

What is happening here is the following: At every step, the new search direction is perpendicular to the previous one. (Remember, the old direction is tangent to the level surface, the new direction is perpendicular to the level surface).

If the level surfaces are almost spherical near the minimum, steepest descent works quite well. If the level curves are elliptical, the method typically does a lot of zig-zagging. The more elongated the level curves are, the worse it gets.

Imaging you are standing in a long and narrow canyon, partly up the side slope. (This is Iowa, but you have probably seen canyons on TV). You want to reach the bottom of the valley, so you should be following the long direction of the valley. Instead, if you choose the method of steepest descent, you have to go sideways first, down to the valley floor, since the slope is steeper in that direction. After that, you have to switch back and forth at 90° angles, so you slowly zigzag down the valley.

If you are lucky, you happen to be on the valley floor at the start, so your initial direction is almost correct. This is what happened with the second set of initial data in the example.

This particular example is constructed to have level curves which are approximately ellipses with the longer diameter ten times as long as the shorter diameter. In general, the method of steepest descent will do very poorly here.

For a general problem, you have no clue what the level surfaces look like. Especially in higher dimensions, it is more likely than not that steepest descent will not work very well. Appropriate scaling may improve things.

### 9.4.4. Conjugate Gradient.

Let us review the general approach: We choose an initial $x_0$, then find $x_1$ by minimizing in the $u_0$ direction, then $x_2$ by minimizing in the $u_1$ direction, and so on.

$x_1$ is a local minimum in the $u_0$-direction, $x_2$ is a local minimum in the $u_1$-direction, by construction. However, we cannot assume that $x_2$ is still a minimum in the $u_0$ direction. The question arises whether there is a way to choose $u_1$ so that $x_2$ is still a minimum in the $u_1$ direction.

In general, the answer is no. However, in the case when $F$ is a quadratic function, we can do this. For general $F$ then we replace $F$ by its second order Taylor approximation, and achieve this goal approximately.

Thus,

\begin{equation}
F(x) \approx F(x_1) + DF(x_1)(x - x_1) + \frac{1}{2} (x - x_1)^T [D^2 F(x_1)] (x - x_1)
\end{equation}

We know

\begin{align*}
 u_0 \cdot \nabla F(x_1) &= 0, \\
 u_1 \cdot \nabla F(x_2) &= 0.
\end{align*}

This just expresses the fact that $x_1$ is a local minimum in the $u_0$ direction and $x_2$ is a local minimum in the $u_1$ direction. We want $x_2$ to be also a local minimum in the $u_0$ direction, so we demand

\begin{equation}
 u_0 \cdot \nabla F(x_2) = 0.
\end{equation}
Putting equations (2.4.4) together, we get

\[ u_0 \cdot [\nabla F(x_2) - \nabla F(x_1)] = 0. \tag{9.3} \]

By differentiating (2.1), we get

\[ \nabla F(x) \approx \nabla F(x_1) + [D^2 F(x_1)](x - x_1), \tag{9.4} \]

so

\[ \nabla F(x_2) - \nabla F(x_1) \approx [D^2 F(x_1)](x_2 - x_1) = \lambda_2 [D^2 F(x_1)] u_1. \]

(Remember, \( x_2 = x_1 + \lambda u_1 \) for some \( \lambda \)). Substitute that into (2.2) and cancel the \( \lambda_2 \) to get

\[ u_0^T [D^2 F(x_1)] u_1 = 0. \]

If \( A \) is any symmetric, positive definite matrix (whatever that means), we say that \( u \) and \( v \) are conjugate vectors (with respect to \( A \)) if

\[ u^T A v = 0. \]

Being conjugate is something like being perpendicular. In fact, if \( A \) is the identity matrix, conjugate vectors are just perpendicular vectors.

We just found that \( u_1 \) has the desired property (of making \( x_2 \) a local minimum in the \( u_0 \)-direction) if \( u_0 \) and \( u_1 \) are conjugate with respect to the second derivative matrix of \( F \) (which happens to be symmetric and positive definite).

Likewise, \( x_3 \) will be a minimum in the \( u_0 \), \( u_1 \) and \( u_2 \)-directions if \( u_2 \) is conjugate to both \( u_0 \) and \( u_1 \).

Our new algorithm is now this:

- \( x_0 \) is an initial guess.
- \( u_0 \) is the first search direction. Usually we use \( u_0 = -\nabla F(x_0) \).
- \( x_1 \) is the local minimum in direction \( u_0 \).
- \( u_1 \) is any direction conjugate to \( u_0 \) with respect to \( D^2 F(x_1) \) (the second derivative matrix of \( F \) at \( x_1 \)).
- \( x_2 \) is the local minimum in direction \( u_1 \).
- \( u_2 \) is any direction conjugate to \( u_0 \) and \( u_1 \) with respect to \( D^2 F \).

and so on.

Think of the analogy with orthogonality again.

You start with some \( u_0 \). There are many possible directions orthogonal to \( u_0 \), so \( u_1 \) is not uniquely determined. At the next step, you need to find a direction orthogonal to both \( u_0 \) and \( u_1 \), so the choices are more limited, but there is still no unique answer. Eventually, at the \( n \)th step, there is a unique answer. After that, you have to throw out some of the older vectors, otherwise there is no solution at all.

In practice, it has proved useful to throw away all the \( u_i \) periodically and start over again.

If \( F \) was a true quadratic function (that is, if (2.1) was exact), then \( x_2 \) would be a minimum in both the \( u_0 \) and \( u_1 \) directions, \( x_3 \) would be a minimum in directions \( u_0 \), \( u_1 \), and \( u_2 \), and so on. \( x_n \) would be the exact answer, where \( n \) is the dimension of the space. In practice, \( x_2 \) is only approximately a minimum in the \( u_0 \) direction, and \( x_n \) is only an approximate minimum, so we have to keep on going.

A geometric interpretation of the conjugate gradient method is that we are trying to find the main axes of the ellipsoid level surfaces. Once we know those, we can zero in on the minimum much faster. Think of the analogy with the long and steep canyon again. After getting oriented, we want to figure out what the main direction of the valley is.

The algorithm, as described, has a serious flaw: we need to calculate the matrix of second derivatives, and then solve a matrix equation at every step to find the new direction.
There is an ingenious algorithm due to Polak and Ribière which avoids that and still produces conjugate directions. We cannot go over the derivation, but the search directions are produced as follows:

\[ u_0 = g_0 = -\nabla F(x_0) \]
\[ g_1 = -\nabla F(x_1) \]
\[ \gamma_1 = \frac{(g_1 - g_0) \cdot g_1}{g_0 \cdot g_0} \]
\[ u_1 = g_1 + \gamma_1 \cdot u_0 \]
\[ g_2 = -\nabla F(x_2) \]
\[ \gamma_2 = \frac{(g_2 - g_1) \cdot g_1}{g_1 \cdot g_1} \]
\[ u_2 = g_2 + \gamma_2 \cdot u_1 \]

and so on. This will produce conjugate directions. Each new search direction is the negative gradient plus a correction term.

**Example:** With the same function

\[ F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4 \]

and initial guess \( x_0 = (1.25, 1.25) \), we get

\[
\begin{array}{ccc}
| x_i | F(x_i) | u_{i+1} = -\nabla F(x_i) |
\hline
(1.2500000, 1.2500000) | 0.0039062500 | (-0.0625000, 0.0000000) \\
(1.2469840, 1.2500000) | 0.0038120960 | (-0.0582067, -0.0603151) \\
(1.0814370, 1.0784570) | 0.0001328376 | (-0.5609487, -0.4576368) \\
(1.0637270, 1.0640080) | 0.0000172822 | (-0.0000172822, 0.0000000) \\
\end{array}
\]

With initial guess \( x_0 = (1.25, 1.5) \), we get

\[
\begin{array}{ccc}
| x_i | F(x_i) | u_{i+1} = -\nabla F(x_i) |
\hline
(1.2500000, 1.5000000) | 0.6289063000 | (4.9375000, -5.0000000) \\
(1.3716800, 1.3767800) | 0.0193443900 | (-0.1012289, -0.1041820) \\
(1.0793160, 1.0758880) | 0.0001571043 | (-0.2005831, -0.0652514) \\
(1.0740560, 1.0741770) | 0.0000302231 | \\
\end{array}
\]

9.4.5. **Newton and Quasi-Newton Methods.** The Newton method for optimization, which we discussed in the one-dimensional case earlier, generalizes to higher dimensions. We can derive the formula as follows.

Look at formula (2.4.4) again, this time around the minimum point \( \overline{x} \)

\[
\nabla F(x) \approx \nabla F(\overline{x}) + [D^2 F(\overline{x})](x - \overline{x}) = [D^2 F(\overline{x})](x - \overline{x}),
\]

since the gradient vanishes at the minimum. If we are at the point \( x \), we need to go in direction \((\overline{x} - x)\) to get to \( \overline{x} \). From the last equation we find

\[
\overline{x} - x \approx -\left[D^2 F(\overline{x})\right]^{-1} \nabla F(x) \approx -\left[D^2 F(x)\right]^{-1} \nabla F(x),
\]

if we assume that \( D^2 F \) does not change very much. The resulting method is an exact analogue of the one-dimensional method

\[
x_{n+1} = x_n - \left[D^2 F(x_n)\right]^{-1} \nabla F(x_n)
\]

There are actually two different methods here: Either, we can use the formula the way we just wrote it. In this case, we need not do any one-dimensional minimizations. Or, we could use \(-\left[D^2 F(x_n)\right]^{-1} \nabla F(x_n)\) as our new search direction \( u_{n+1} \) and do a one-dimensional search.
I think that one method is known as Newton-Raphson, the other one is just Newton’s method. Unfortunately, I can never remember which is which, and the books I have handy don’t say. Most people use “Newton” and “Newton-Raphson” interchangeably, anyway.

**Example:** With the same function

\[ F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4 \]

and initial guess \( x_0 = (1.25, 1.25) \), we get

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( F(x_i) )</th>
<th>( u_{i+1} = -\nabla F(x_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.2500000)</td>
<td>0.0039062500</td>
<td>(-0.0033333, -0.0033333)</td>
</tr>
<tr>
<td>(1.0000190, 1.0000190)</td>
<td>0.0000000000</td>
<td></td>
</tr>
</tbody>
</table>

With initial guess \( x_0 = (1.25, 1.5) \) we get

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( F(x_i) )</th>
<th>( u_{i+1} = -\nabla F(x_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.5000000)</td>
<td>0.6289063000</td>
<td>(-0.0033333, -0.2533333)</td>
</tr>
<tr>
<td>(1.2466660, 1.2466260)</td>
<td>0.0037020270</td>
<td>(-0.0032189, -0.0031790)</td>
</tr>
<tr>
<td>(1.0524790, 1.0548480)</td>
<td>0.0000637142</td>
<td></td>
</tr>
</tbody>
</table>

In both cases, I have done line minimizations. □

Instead of trying to find out the main axes of the ellipsoid level surfaces, this method tries to go to the center in a single step. In this example, it pretty much succeeded.

The main problem (as usual) with this method is the need for a full matrix of second derivatives, plus the need to invert the matrix at every step.

A whole class of methods, known as quasi-Newton methods, deals with this problem. The idea is to replace the exact inverse second derivative matrix by some approximation to it which can be found with relatively little work. Usually, we assume that the matrix changes very little from step to step, so we just update the last matrix based on information we gathered at the last step. The details go far beyond the scope of this course.

**9.4.6. Summary of Higher Dimensional Optimization.** You have probably gathered by now that optimization in higher dimensions is fairly complicated and (computer) time consuming. The higher the dimension is, the more work you need to do per step, and simultaneously you need more steps.

We have considered a few direction set methods, namely

- Steepest Descent, where we search in the direction of the negative gradient;
- Conjugate Gradient, where we search in a direction conjugate to the earlier directions;
- Newton’s method;
- Quasi-Newton methods.

Steepest Descent and Conjugate Gradients may or may not work well. Newton’s method, as usual, is the best, except that in higher dimensions the amount of extra work becomes enormous.

A look at some subroutine libraries reveals that quasi-Newton methods are the methods of choice. Every single one I looked at had some form of quasi-Newton method in it. This includes the subroutine UNCMIN provided with NMS.

**9.5. Subroutine UNCMIN**

Read this section on your own.
APPENDIX A

Computer Info

A.1. Available Computers

You may use any computer of your choice for doing the programming assignments. The Computation Center (CC) provides free access to three types of machines: the Vax, Wybur (an IBM mainframe clone), and Project Vincent (PV), a network of DEC workstations. I will refer to these three as the “CC machines”. Any other machine that you have access to at work or at home is also acceptable.

I recommend that you do the programming assignments in MATLAB rather than Fortran. Of the CC machines, only PV has MATLAB, as far as I know. Both PV and the Vax have the necessary subroutines to do the assignments in Fortran.

If you want help from me, I recommend that you use PV. If you want to use your own machine, or one of the other CC machines, you are on your own. You will need a copy of MATLAB, or a Fortran compiler with a library manager.

Public PV workstations are in Durham 139, in Durham 248 (when they are not used for a class), 116 East Lagomarcino, 131K Coover Hall, and other places. There are six public machines in the Math Department, in rooms Carver 451 through 457. You can use four of them (not the DEC 5000s). Unfortunately, the rooms are always locked nowadays, after somebody walked off with one of our a Macintosh computers in broad daylight. Unless you can find somebody to let you in, you can only use them remotely.

You can dial into the CC machines remotely, either from another machine on campus or through a phone line (see below for details). However, most of the public PV stations are restricted to the user sitting in front of it. The only public machines I know that allow remote login are

vincent1.iastate.edu
vincent2.iastate.edu
pv3437.vincent.iastate.edu
pv343d.vincent.iastate.edu
pv3440.vincent.iastate.edu
pv3455.vincent.iastate.edu

Vincent1 and Vincent2 are DEC 5000s in the basement of Durham Center, and are usually pretty crowded; the other four are the public machines in the Math Department. There is also a machine class1 that you may have access to. It is supposed to be restricted to people taking certain CS courses, but I don’t know if that is being enforced.

A.2. Getting Started

You should use the first couple of weeks of the term to familiarize yourself with the computer you are going to use. Here is a brief outline of the things you need to learn. If you are familiar with your system of choice already, you may be able to skip some of the steps. I will provide the details for PV only.

The Computation Center offers introductory courses to PV for free. You even get free course notes with that. I recommend this highly for new users.
A hypertext introduction to PV is available online. You need to type

```
% add abc
% lesson vincent
```

Manuals and other help for CC machines are available in the following places:

- **On-line help.** On PV, the command is `man` (short for “manual”). You can do `man man` or `help help` for starters. If you are sitting in front of a PV machine, you can use the window version `xman` instead of `man`.
- **CC consultants** are on duty Monday through Friday 8am to 5pm and Sunday through Thursday 6pm to 9pm. Go to Durham 138 or call them at 294-1314.
- **Manuals** are available for consultation in Durham 138 (the help room) or Durham 195 (where you get accounts). Manuals are for sale in Durham 195.
- **PV has an on-line consulting facility olc**, where you can ask questions. Type `olc` and take it from there. You will get a personal answer within a few hours. They will answer questions like “How do I get `matlab` started?”, but not “How do I do this in `matlab`?”. Before you ask a question, type `answers` from inside `olc` and look at the answers to frequently asked questions. Maybe your question is already answered there.
- **There is a local newsgroup `isu.math.473` for this class which can be accessed from PV, Vax, and Wylbur. See below for details.**
- **Ask a friend.**
- **Send e-mail to `keinert@iastate.edu`.**

If your problem has to do with computer use in general, try the first six options first, before you ask me. I will be glad to answer questions about your assignments, but I am here to teach mathematics, not computer use.

Now here are the things you should do or familiarize yourself with:

- **Get an account.**
  You can sign up for a PV account by going to any unoccupied PV station and clicking on the `Register` box, or by logging in remotely as user `register`. After registering, it will take several hours until you can actually log in. There are fliers outside Durham 195 with more details about the registration procedure.

- **Customize your account.** Most machines let you create a file that gets executed every time you log in. Here is where you tell the system what kind of terminal you have, what prompt you want to use, etc. On PV, you are supposed to split this stuff into several files, but for beginners it is easier to stick with just one or two of them.

  PV is intended to be run as a window system from a workstation, but most commands also work on a text-only terminal. All the commands you only want executed when you are in window mode go in a file `.startup.x11`. If you use the `save state` feature when you log out you probably don’t need this file. All the commands you want executed in both text and window mode go into a file called `.startup.tty`.

  For this class, `.startup.tty` should contain at least the following lines:

  ```
  add mathclasses
  add matlab
  setenv MATLABPATH /home/mathclasses/matlab/473
  ```

  The first time you log in, you can type those in by hand, or type `source .startup.tty` after creating the file.

- **Find the files used in this class.**
  All the files mentioned here are subdirectories of `/home/mathclasses`.

  There is an IBM PC floppy disk in the back of your textbook. This is only needed if you want to use Fortran. The contents of the disk have already been installed on Vax and PV. Fortran code is in `/src/nms`, double precision version in `/src/dnms`, and the data files in `/src/nms/examples`. 
You should also look around `keinert`, especially in subdirectories 473 and 473notes. Matlab routines and data for this course are in `matlab/473`. There is some `matlab` documentation in `doc/matlab`.

- **Learn to use an editor**, for typing in your programs and other text.
  
  On PV, the main choices are `vi` and `emacs`. `vi` stands for “visual editor” and is the basic editor on all Unix systems. The emphasis is on “basic”: easy to learn, but limited. You start the editor with `vi filename` and exit with `<Esc> :wq <Return>`. (By the way, the Escape key is labeled F11 on DEC workstations).

  `Emacs` takes a little longer to learn, but has a lot more features and is highly customizable. You start it with `emacs filename` and exit with `Ctrl-X Ctrl-S Ctrl-X Ctrl-C`. `Emacs` has a built-in tutorial which can be invoked with `Ctrl-H t` from inside `emacs`. Whenever you edit a file with `emacs`, it saves the original in a file with the same name, but a tilde at the end. It also saves the file in a temporary file every 300 keystrokes, in a file with the same name surrounded by # signs. Thus, if you are editing `prog.f`, the file `prog.f~` is the old version, and file `#prog.f#` is the save file. The save file will disappear by itself, unless `emacs` crashes.

  There are a number of other editors around. I don’t know much about most of them. Some of them are
  - `tpu` is similar to the Vax editor of that name, I think.
  - `joe` (Joe’s Own Editor or something) bears a striking resemblance to the old WordStar for PCs.
  - `jo` is some modified version of `joe` in the `aeem` locker.
  - `xedit`, a very basic X window based editor.
  - `xcoral` is another X windows based editor in the `aeem` locker.
  - `vim`, an extension of `vi`, in the `aeem` locker.
  - `axe`, Another X windows Editor in the `axe` locker.
  - `epoch`, an extension of `emacs`, in the `math` locker.

  By the way, when I say “in the `aeem` locker”, that means you have to type `add aeem` before you can call up the editor.

  Whatever you do, be sure you are using a **text editor** and not a **word processor**. A text editor creates plain files (“ASCII files”) with only text in them, the kind that compilers like to see. A word processor is a program which lets you use underline, boldface, etc. It puts internal codes in the file that compilers don’t like. Most word processors have a way to read and write plain text files, but you have to ask for special treatment for the file every time you do something. This tends to get on your nerves (on mine, at least).

- **Learn how to run `matlab` or the Fortran compiler**.
  
  To use `matlab`, your `.startup.tty` file must contain the line `add matlab`, or you have to execute it by hand. Then, simply type `matlab`. You should also add the line

  `setenv MATLABPATH /home/mathclasses/matlab/473`

  so that `matlab` knows where to find the files I will provide for you.

  You should print out the file `/home/mathclasses/doc/matlab/primer35.ps` on any PostScript capable printer (not a dot matrix printer). This, together with the built-in `help` function in `matlab`, should be all you need to learn enough `matlab` for this course. If necessary, you can use the manuals in Durham for specific questions.

  To use Fortran, take a simple test program, like the old “hello, world” program:

  ```fortran
  program hello
  print*, 'hello, world'
  end
  ```

  Put this in a file called `hello.f`.

  The standard Fortran compiler on PV is called `f77`, but you might as well get used to the batch file `f473` we use for this course. Using `f473` instead of `f77` will make sure that your program can find the subroutines it needs. It also turns on some forms of error checking, and it will force you to
explicitly declare all variables. You may not like that, but it is good for you. Trust me. You compile and run the program with the commands

```c
% f473 hello
% hello
```

If you keep main program and subroutines in different files, you can still compile them together with f473. Just make sure that the main program is the first file you list.

If you want to use Fortran on a personal machine, come talk to me. It is possible to use languages other than Fortran to link to the subroutines in the book, but it is a little tricky. Again, come talk to me if you plan to do that.

• **Learn how to print a file.**
  The print command on PV is called `lpr`. Look in the PV User’s Guide for details.

  Two words of caution:
  First, each PV workstation has a default printer where the output will appear. The `lpinfo` command will tell you the name and location of that printer. If you use any machine not supported by the CC, like the Math department machines, your output may go someplace unexpected, or disappear altogether. The `-P` parameter in `lpr` is used to redirect output.

  Second, not all printers are capable of printing all files. For some of the public machines in Durham 139, the default printer is an old dot matrix printer which cannot handle PostScript files. *On any printer, don’t ever try to print a `.dvi` file directly. You will get lots of expensive scratch paper. Run it through the `dvips` program first.*

• **Learn how to do basic plotting.**
  “Basic plotting” means you should be able to graph one or more curves given as a sequence of $x$ and $y$ values, both on the screen and on paper. That is all we need for this course.

  If you are using `matlab`, use the built-in `plot` function to display a plot on the screen. You can print the plot with the `print` command. If that doesn’t work for some reason, you can save the plot in a file with the `meta` command, convert it to PostScript (at the operating system level) with the `gpp` command, and print it with `lpr`.

  If you are using Fortran or anything else, you can use one of the two programs `ploty` and `plotxy` that I wrote. They do the following steps:

  - Scan the input file for columns of numbers; ignore everything that does not look like a number. `Plotxy` assumes that the first column contains the $x$-values, the other columns contain the $y$-values of curves. `Ploty` assumes the user wants equally spaced $x$, and treats all columns as $y$-values.

  - Produce a plot on the screen.

  - Leave the plot in PostScript form in a file, ready for printing. The file name is the input file name with `.ps` appended.

  For example, suppose your file `prog1.out` looks like this:

<table>
<thead>
<tr>
<th>$x$</th>
<th>true values</th>
<th>numerical values</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.000</td>
<td>1.000</td>
<td>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>1.100</td>
<td>1.080</td>
<td>0.02</td>
</tr>
</tbody>
</table>

  To plot it, find yourself a PV station and log in. If you are logged in remotely, you can still produce a plot to be printed, but you can’t look at it on the screen.

  ```bash
  % plotxy prog1.out
  found 3 curves with 20 points each
  %
  ```

  After a short while, a black rectangle will show up on your screen. Click on it with the mouse, and
a plot with three curves will appear: the $x$ axis has units 0.0, 0.1 and so on, the $y$ values come from the other columns. Click again, and the plot will disappear. The program uses different colors for different curves: black for the axes, red for the first curve, green for the second, then blue, purple, gray, and then it starts over again with red.

Since your input file was `prog1.out`, there will now be a file `prog1.out.ps` which contains the plot in PostScript form, ready for printing. Most PV printers speak PostScript, but not the dot matrix printers.

There is an intermediate file called `gmeta` which can be erased afterwards. Look at the manual page for `ploty` and `plotxy` for more detail. Basically, I wrote these programs so that you can take the printed output from your program and plot it, without having to do a lot of reformatting.

- **Learn how to use electronic mail.**
  You can start by sending a message to yourself. PV mail is a little different from other flavors of Unix. The basic Unix `mail` command still works for sending mail, but there are better ways. Read your manual or the on-line help.

- **Learn how to access the Usenet news groups.**
  A newsgroup is a place where people with a common interest can read and post messages. Thousands of subjects range from computer-related topics to cooking or movie reviews.
  You should get used to reading at least the standard campus groups (like `isu.cc.general`) and the group for this class, which is `isu.math.473`. You are responsible for knowing whatever I post in `isu.math.473`.

You are encouraged to post messages in this group yourself to ask for clarifications to the homework and things like that. The advantage of a newsgroup is that I only have to answer the question once, and everybody else can see the answer.

There are several newsreaders on PV. I know of `nn`, `rn`, `trn`, and the window version `xrn`. I personally find `trn` the easiest to use since it groups items by topic.

- **Learn how to Access These Notes on PV**
  I am writing these lecture notes in TeX, a mathematical typesetting system. On PV, go to directory `/home/mathclasses/keinert/473notes`. There will be a million files there, but you can ignore most of them. The important ones are
  
  - The original text is in `chapter0.tex`, `chapter1.tex`, ..., `appendixa.tex`, ... These files are only interesting to you if you want to learn TeX.
  
  - The files ending in `.dvi`, like `chapter0.dvi`, can be viewed on the screen. Use the `xdvi` program. Look at the manual page for `xdvi` for more information. You have to do `add tex` before you can use it.
  
  - The files ending in `.ps`, like `chapter0.ps`, can be printed. Use the regular `lpr` command. Do not attempt to print the `.dvi` files directly. You will be sorry. Unless the Computation Center staff has fixed this recently, you will get many, many pages of garbage that way (at 5 cents per page).

  The `.ps` files can also be previewed, either with the command `dxpsview` or with GhostScript (add `ghost`, followed by `gs filename.ps`). The display quality is worse, though, than with `xdvi`, and you cannot go backwards to the previous page.

  Any other files you see you can ignore.

**A.3. Remote Login**

From most machines hooked up to the campus ethernet, whether VAX, IBM PC compatible, Macintosh, workstation, you can use the `telnet` program. Type `telnet vincent1.iastate.edu`, and you get connected to `vincent1`, for example. Look at the manual page for `telnet` for more info.

If you are dialing in through a modem, you need to do the following: call the campus modem pool (294-1200 for 1200 baud modems, 294-2400 for 2400 baud modems, and so on. I think there is also 294-ENET, but only for the really fast 14,400 baud modems). To get to `pv343d`, the connection would work as follows (minor details of this procedure change from time to time):

```<you call in, hit <return> a few times>```
DIAL: ethernet (or enet)
RINGING
ANSWERED

< I find it useful to wait a few seconds at this point; the system tends to hang forever if I don't >

< hit <return> again>

*** Welcome to ISU Telecommunications Terminal Server ***

1. Connect to Telnet host
2. Connect to LAT host
3. Vincent1
4. Vincent2
5. Class1

10. Logout

Enter number of selection> 1

Connect to Telnet host...

< type in the full name of the machine, like pv343d.iastate.edu>

Xyplex -010- Session 1 to PV343D.IASTATE.EDU established

Project Vincent / ULTRIX V4.2 (pv343d.iastate.edu)
login:

A.4. File Transfers

From most machines hooked up to the campus ethernet you can use the ftp program. To transfer files to or from vincent1, you would type ftp vincent1.iastate.edu. Look at the manual page for ftp for more info. This is very fast, even for large files.

To get access to /home/mathclasses on PV for ftp, first issue the command

quote SMNT mathclasses

from inside ftp (SMNT must be in upper case, followed by exactly one space). This should work from most machines. You can then get or put the files you need.

A special PV machine called pvio.vincent.iastate.edu in Durham 139 has every device commonly used on workstations attached to it, including a floppy drive. This floppy drive normally uses a Unix file system, not an MS-DOS or Mac file system. However, there is a bunch of public domain programs called mtools for reading/writing MS-DOS floppies. Type add mtools, and then man mtools for help.

If you are logged in through a modem, you can transfer files through your terminal emulator (Kermit, ProComm, Zmodem, or whatever). Ask the debug room (294-1314) for details. This is likely to be quite slow, and is only recommended for small files.
APPENDIX B

Fortran Hints

This appendix contains hints on how to find errors in your programs, and how to avoid some common Fortran errors in the first place.

The basics on how to invoke the Fortran compiler are described in appendix A.

B.1. Some Common Fortran Errors

Most students have a pretty good grasp of basic Fortran, which is really all you need for this course. However, I have noticed that some students don’t know what the common and external statements are for. You will need both of these at some point. If you don’t know offhand what these things are, go find a book on Fortran and read up on them.

The following are some common errors I have come across in the past. They probably account for 90% of all errors at this level, and becoming aware of them might save you hours of searching. Most of these errors I have made myself at least once. Try and guess what the problem is and how to fix it before you look up the solution.

1. Consider the program fragment

   ...
   x = 3 / 5
   y = 10 ** (-4)
   ...

   $x$ and $y$ both turn out to be zero. Why is that?

2. Suppose you write

   double precision pi
   pi = 3.14159265358979323846
   ...

   and then print out $pi$ to 15 digits. Only the first 7 will be correct. Why?

3. Consider the code

   ...
   real x, x0, h
   ...
   x0 = 1.
   h = 0.1
   do 10 i = 0, 10
        x = x0 + i * h
   ...

B-1
Obviously, you want to perform some calculations with $x = 1, x = 1.1, x = 1.2, \ldots$ Why does this not work?

(4) What is wrong here?

```fortran
double precision x, y, z
...  
z = sum(x,y)
call summ(x,y,z)
...

double precision function sum(x,y)
double precision x, y
sum = x + y
end

subroutine summ(x,y,z)
z = x + y
end
```

Both the function and subroutine calls are supposed to assign the sum of $x$ and $y$ to $z$. Neither one works right (actually, they may work on a particular machine, but don’t count on it).

(5) Here is an easy one:

```fortran
real a(100), x
...  
do 10 i = 1, 200
   10 a(i) = x
...
```

(6) What is wrong here?

```fortran
...  
read (*) n, x(0), x(n)
...
```

(7) What is wrong here?

```fortran
h = 0.3
x = 0.
1 <some statements>
x = x + h
if (x .le. 3.) goto 1
...
```

The loop only gets executed up to $x = 2.7$.

(8) Guess the value of $x$ after this statement:

```fortran
x = 10. / 5. * 2.
```

(9) What is the difference between the statements
do 10 i = 1, 10
   do 10 i = 1, 10

(10) What is wrong with the following statement?

   x = y**2.0

And here are the answers:

(1) If you perform any operation on two integers, the result will be an integer. If the result is assigned to a real number, it gets rounded to an integer first, then made into a real number again.

   3 / 5 = 0
   3.0 / 5.0 = 0.6
   10 ** (-4) = 0
   10.0 ** (-4) = 0.0001

   In the last case, it is vastly better to write 1.0e-4.

(2) 3.14... is a real constant. It first gets truncated to single precision and then extended again to double precision. This may seem crazy, but that is what the Fortran syntax demands. The correct way is to use a double precision constant (with d0 at the end).

   double precision pi
   pi = 3.14159265358979323846d0

   or in a more portable version

   double precision pi
   pi = 4. * atan(1.d0)

   This will give you full precision on ANY machine.

   Mistakes of this type are pretty hard to find, because the answer will be correct to 7 digits, just not to 15 digits, as you had expected.

   Remember: If you work in double precision, all floating point constants should carry a d0 at the end. Otherwise your results may only be single precision.

(3) One of the main problems of Fortran is that it does not require you to declare variables. If you have a variable x0 (x zero) somewhere, and you misspell it to xO (x capital Oh), you will probably search for this mistake for hours.

   Another common consequence of not declaring variables is this: You have a double precision variable in the main program and pass it to a subroutine as a parameter. In the subroutine, you forget to declare it, and it ends up as a single precision real. The result: you get garbage from your subroutine. Or maybe it works on one machine, but not on another. (See example 4).

   A good compiler should notice this last mistake, at least if main program and subroutine are compiled together, but I have seen many that won’t. The VAX Fortran compiler is one of the bad ones. I recommend highly that you declare everything explicitly and even ask Fortran to complain about nondeclared variables. I have set up the standard Fortran compilers on both Vax and PV to do that.

(4) Check every subroutine and function call and make sure that the arguments have the correct number, order and type (see also number 3). Many Fortran compilers will not notice mistakes, and your subroutines may produce garbage.

   sum is a double precision subroutine, but the main program thinks it is single precision. (sum has not been declared, so it is automatically single precision).

   The arguments of summ are double precision in the main program, single precision inside summ.
These things may actually work correctly on some machines, but certainly not on all machines. This brings up another point: Even if you knew that your favorite compiler handles something like mismatched types in subroutine calls correctly, you should still not use it. You are asking for trouble if you want to upgrade your compiler or move the program to a different machine. Another common problem like this is initialization. Most compilers set all variables to 0 initially, but don’t count on that. I had a summer job one time converting a big commercial program package from PDP 11 Fortran to IBM/370 Fortran. After weeks of tracing through about 500 subroutines, it turned out that the only problem was that the IBM Fortran did not initialize variables to zero. Not only that, it did not even have a compiler option to do that. I spent the rest of the summer hunting down all the variables and setting them to zero at the beginning of every subroutine.

(5) Make sure your arrays are big enough. If you declare an array of size 100 and write something into element 120, you will be destroying some other variable (whatever happens to be at that location). I have set up the standard Fortran compilers on both Vax and PV to check array bounds.

(6) The problem is that the storage location of x(n) is computed before the statement starts. n at this point is undefined, so automatically has value 0 (or garbage). The program reads n and x(0) correctly, but then stores the third number again into x(0). Thus, neither x(0) nor x(n) are correct. The problem is corrected by splitting the read statement into two.

```fortran
read (*) n
read (*) x(0), x(n)
```

Be careful, though: now the number n must be on a separate line. If you read a partial line in Fortran, it throws away the rest of the data on that line. With most compilers, the x(0) and x(n) can be on the same line or on two different lines.

(7) The problem is round-off error. The number 0.3 is represented internally as 0.3000001 or something similar, so the last comparison is with x = 3.000001 or something, and fails. The solution: Use only integers for loop counters. If you must use a real number, compare x to 3. + epsilon instead of to 3., where epsilon is a small number near the machine epsilon.

(8) x = 4, of course. Fortran interprets this as (10./5.)*2., not as 10./(5.*2.). Multiplication and division have equal precedence and are evaluated left to right. The division first, then the multiplication.

(9) The first statement is the beginning of a do loop. The second assigns the value 1.1 to the variable do10i. Another good reason to declare all variables. Fortran ignores all blanks inside a statement. The variable XMIN and X MIN N are exactly the same. Somebody on Usenet mentioned a similar problem that actually happened to him. It went about like this:

```fortran
if (n .gt. 10) goto 1
n = 10
```

The second line had slipped left one space by accident. The n was now in column 6 and counted as a continuation character. The effect: variable goto1 was assigned a value of 10. The moral of the story: declare every single variable, and make the compiler enforce that.

(10) The problem is that a power with non-integer exponent gets evaluated via logarithms and exponentials. For example, x**3.14 is calculated by taking the log of x, multiplying by 3.14 and exponentiating again. Powers with integer exponents are evaluated by repeated multiplication. In the harmless case, y**2.0 takes unnecessary computer time to evaluate. However, if y is negative, then y**2.0 fails, while y**2 would work just fine.
B.2. How to Avoid or Find These Errors

Here is how I would recommend you go about writing a program:

1. Write the program and type it in. Before you compile it, ask yourself a few simple questions:
   - Is everything explicitly declared, and will the compiler complain about undeclared variables?
   - Are all arrays big enough, and will the compiler catch references to undefined array elements?
   - Are the parameters for all subroutine calls in the correct order and of the correct type?
   - If you are using double precision, do all constants carry a d0 at the end?
   - Are all variables initialized? Don’t rely on the compiler to set them to zero automatically.

   That should get rid of the most common mistakes right away. I have set up the standard compilers
   on both Vax and PV so that they try to catch as many of these errors as possible.

2. Compile the program and fix the errors the compiler complains about.

3. On both Vax and PV you will find a program called ftncheck. It will scan your program and flag
   suspicious looking statements, similar to what the lint program does for the C language.

   Run ftncheck on your program and look at the messages you get. Usually, most warnings are
   harmless and can be ignored, but you may catch some mistakes this way.

4. Now, let’s assume the compiler is happy with your program, and ftncheck is also happy. Before
   you let your program loose on the data from the homework assignment, try it on some really stupid
   data or on an example from class, where you know the answer.

   If your program is a matrix solver, try the identity matrix or a little two by two matrix. If it is a
   numerical integration program, integrate the constant 1 or the function $y = x$. You get the drift.

   You will be surprised how many minor mistakes you can find that way. Things like a missing factor
   of 2 show up much more clearly this way than if you use a full set of actual data right away.

   If all goes well, put in the real data, and you are done.

5. What if the program dies with some weird error message?

   Whatever the message is, write it down exactly if it does not make sense immediately. It happens
   frequently that students have problems with their account that are not reproducible while I am
   watching. (Don’t you hate that?). Then go and find somebody who can tell you what the error
   message means.

6. What if I find the words Infinite, NaN or Indefinite in my output?

   This means a floating point error has occoured. See section B.3.

7. Now, let’s assume you have done all these things, and the program runs, but the answers are not
   correct. You have looked at the code and cannot find the mistake. Now what?

   I would use a debugger at this point. A debugger is a program that lets you single-step through
   your program and look at variables along the way. The standard debugger on PV is called dbx; on
   Vax it is called debug. Neither one is very easy to use, but they pay off in the long run. Look at the
   on-line documentation.

   The best debugger I have ever used is called codeview and comes with Microsoft Fortran for
   IBM/PC. If you have access to that, the debugger alone would almost make it worthwhile to use
   that system.

B.3. Floating Point Errors

There are two types of floating point errors. Severe errors are overflow, division by zero, and illegal
operations like taking the square root of (-1). Whenever you run into a severe error, there is a mistake in
your program. Find out where it is, and fix it.

Underflow is a minor error. Figure out why it occured, and ignore it if it does not seem to be serious. If
the underflow occurs inside one of the canned subroutines, you can assume it is harmless. I know that this
happens sometimes in the blas, and maybe in other routines. If it occurs in the part of the program that
you wrote, it may be a programming error.

On Vax, I have set up the compiler so that all floating point errors are reported. The severe errors
terminate the program after the message. Execution continues after an underflow error, with the result set
to zero. These options can be changed, if they get to be too much of a nuisance.

On PV, underflow gets rounded to zero without message, and the program dies on more severe errors. If
you find some results labeled infinite or NaN or indefinite in your output, a severe floating point error
has occurred, and the program continued anyway for some reason.

To find the place where the error is, run the program from inside the debugger until it crashes, and ask the debugger where you are. Here is an example:

```fortran
program test
  real x

  ! x = 0.
  call inverse(x)
  print*, x
end

subroutine inverse(x)
  real x

  ! x = 1. / x
end
```

Compile and run this:

```
% f473 test
compiling test.f into test
% test
forrtl: error: floating divide by zero
Illegal instruction
```

Ok, there is a division by 0, but where is it in the program?

```
% dbx test
dbx version 2.0
Type 'help' for help.
reading symbolic information ...
[using test.test]
test: 4 x = 0.
(dbx) run
Floating point exception [inverse:12 +0xc,0x40026c]
  x = 1. / x
(dbx) where
> 0 inverse(X = 0.0) ["test.f":12, 0x40026c]
  1 test.test() ["test.f":5, 0x40020c]
  2 main(0x7fffb84, 0x0, 0x0, 0x0, 0x0) ["for_main.c":169, 0x4002a0]
(dbx) quit
```

After run, the debugger already tells you where the error is: in line 12 of the program, inside subroutine inverse. To find out how you got there, type where.

This tells you that you are in subroutine inverse, which was called with a parameter of x = 0.0, and your position corresponds to line 12 in file test.f. This routine was called from subroutine test.test, which is the name the compiler gave to your main program, line 5 in file test.f. Your main program in turn was called from a routine main, from line 169 in file for_main.c. This last stuff has to do with the internal workings of the compiler and is of no interest to us.
This appendix is addressed to faculty members and graduate students teaching Math 473. Most of it should apply to other Math courses that use computers, with suitable changes. You should also look at the introductory remarks for students I wrote up (appendix A).

If you find any errors or unclear passages in this document, please notify Fritz Keinert (phone: 294-5223, e-mail keinert@iastate.edu).

Students are highly encouraged to use Project Vincent (PV) machines. I left instructions for setting up the Vax in this write-up, since I wrote them long ago, but I would not start any new projects on that machine.

C.1. Project Vincent

C.1.1. Finding the Files. We have a special area set aside for math classes. To get to it, you need to type add mathclasses first. This command should go in your .startup.tty file, so you don’t have to type it every time.

A directory /home/mathclasses will show up, which is organized as follows:

src: Source code for various packages we use.
lib/dec: Compiled libraries for linking.
bin/dec: Programs and shell scripts to be used by classes.
keinert: and similar directories for other instructors contain files of interest to students of this instructor. I put my general interest files into /home/mathclasses/keinert, and files of interest to only one class into subdirectories /home/mathclasses/keinert/473 etc. You can do whatever you want with your own directory.

If you keep files with sensitive information (like test scores), you are responsible for protecting your directories adequately (see below).

C.1.2. The Andrew File System. The following section was written at a time when most of the information existed in Martin Calsyn’s brain, but was not written down anywhere. The current PV User’s Guide has a better and more up-to-date description than this section, but maybe you’ll learn something from this, anyway.

If this section looks confusing to you, have Mike Fletcher set up your directories for you. It only takes a couple of minutes.

There are two types of remote file systems on PV: the Network File System (NFS), also used by the Suns, and the Andrew File System (AFS). Faculty and graduate student home directories, and much of PV system software, are on NFS; undergraduate student home directories are on AFS.

AFS offers much improved file protection, so we are using that for the class directories. It also offers file caching, which is supposed to speed up file access and cut down on network traffic. There may be a few machines that don’t have access to AFS yet. Talk to Mike Fletcher if you have any problems.

In regular Unix and in NFS, there are three types of access permission for files and directories: read, write and execute. Each one of them can be specified independently for three groups of people: yourself, your group, and everybody.
On AFS, you can set seven protections for directories. Three of them affect access to existing files in the directory:

- **r (read)**: Permission to read files in the directory.
- **w (write)**: Permission to edit files.
- **k (lock)**: Permission to lock files, i.e. take access to a file away from other people while you are using it. This is for applications where several people need to modify the same file and want to avoid chaos. It is not needed in our setting.

The other four affect the directory itself:

- **d (delete)**: Permission to delete files.
- **i (insert)**: Permission to create new files.
- **a (administer)**: Permission to change permissions.
- **l (lookup)**: Permission to list the contents of the directory.

The files inside a directory can be separately protected by the old-style Unix settings in the user field. The old-style permissions for group and world are ignored.

Standard read-only access is `rl`. This is what you want your students to have.

You can attach up to 20 sets of permissions to a directory, each with a list of people who have that type of permission. Those lists are called Access Control Lists (ACLs). Permissions can be positive or negative (i.e. you can take away permissions for selected people).

Any Moira list (like `math`, `mathfac`, etc.) can be an ACL by putting `system:` in front of its name, but not the other way around (I think). Any user has permission to create up to 20 ACLs himself or herself.

Permissions are managed with the `fs` command, found in `/usr/bin/afsws/bin`. Read the man page for full information.

Let’s look at an example.

```bash
g:~% fs la /home/mathclasses
Access list for /home/mathclasses is
Normal rights:
  system:mathfac rlidwka
  system:mathclasses rl
  system:math rlidwka
  system:administrators rlidwka
  system:authuser rl
```

This means: the members of list `mathfac` have full permission `rlidwka`. This list contains all Math faculty. Likewise, lists `math` and `administrators` have full permission. List `math` contains the Math superusers. I don’t know about list `administrators`; that must be the PV system gurus.

Then, there is a list `mathclasses` which has read-only permission. At the moment, `mathclasses` contains `LIST:everybody`, but we could easily restrict that to students currently taking Math courses later.

List `authuser` also has read-only access. This must be present for some technical reason, otherwise access for `LIST:everybody` does not work. (Don’t ask me why; I didn’t design the system).

If I am teaching Math 473, for example, I could give permission to look at `/home/mathclasses/keinert` to everybody, but restrict the 473 subdirectory to students in this class only.

Obviously, you can carry this to the point where you spend your days fine-tuning the access permissions. Then again, it takes only a few minutes at the beginning of the term to set up the protections on a couple of directories, which is really all we need.

### C.1.3. Getting Set Up.

I am using the following setup myself. You can use a similar one or make up your own.

- Create `/home/mathclasses/keinert`, with access for everybody. This is for general information, like office hour information and help files.
- Create `/home/mathclasses/keinert/473` with access for everybody. I could restrict access to students in the class only, if I ever consider that necessary. This is for handouts for this class, homework
assignments, homework and test scores (without names, of course, just to let students see where they stand in the class), data to be used for assignments.

- Put utility programs and shell scripts you expect your students to use in /home/mathclasses/bin/dec. For example, I have a shell script f473 there, which will compile the students’ programs with the compiler settings I like, and link them with the necessary subroutines automatically. I also have a couple of simple plot programs there.

- Put documentation in both /home/mathclasses/man/mancc and /home/mathclasses/man/catc, so the man program can find them.

- Put the source code of any routines you expect your students to use in a subdirectory of /home/mathclasses/src.

- Put compiled subroutine libraries for your students in /home/mathclasses/lib/dec.

C.2. The Vax

Personally, I am phasing out the Vax. Students can use it as long as it works, but I won’t be spending any more time updating stuff or installing new things. The following is an old write-up which should still be accurate.

C.2.1. Finding the Files. Several scientific subroutine packages regularly used in numerical analysis courses are stored on the Vax. We keep both the source listings and the compiled versions there. The files are kept in classlib:[math] and its subdirectories. You can get to these files with the list and get commands described below. If you ever need to make any changes: The files are stored under user id simth, which belongs to the department, in directory classlib:[000000.math]. The password is known (at least) to Fritz Keinert, Roger Alexander and Jim Cornette.

I assume you know about the class: directory and its uses. Get the info from the Computation Center if you don’t.

C.2.2. Getting Set Up. On a class account, you have two different login files. One is login.com in your own directory, which gets executed first. The other one is glogin.com in directory class:. The things in login.com only affect you, the contents of glogin.com also affect your students. This section discusses what I think should go into glogin.com, and why.

In order to save disk space, the source code for various packages is stored in compressed form (arc format, short for archive). I have written two command files to extract the source code, called list and get. List gives you a listing of available files, and get actually deposits a copy in your directory. If you just type list or get without parameters, the commands will give you a short description of themselves. When you use one of these commands, you get a warning message at the end. That is a bug in the arc program or the operating system and is harmless.

In order to use these commands, glogin.com has to contain the following lines:

```bash
$ arc := $classlib:[microstuff]arc
$ list := @classlib:[math]list
$ get := @classlib:[math]get
```

Tell the students that they are not supposed to add the source code to their programs. Instead, they should link their main programs with the already compiled versions. This saves both storage space and computer time. The source code is available for reading or for downloading to micros, if they do their assignments on one. If students ask you about downloading, tell them that

- the same source code is sitting on Project Vincent, and is a lot easier to get from there, and

- they will have to make changes to machcon.for if they move the programs to a different machine.

An easy way to link with the compiled libraries is to add one or more of the following lines to glogin.com

```bash
$ nms := classlib:[math.nms]nms/library
$ dnms := classlib:[math.nms]dnms/library
$ fmm := classlib:[math.forsythe]fmm/library
$ depac := classlib:[math.depac]depac/library
```
This will define a short abbreviation for the corresponding libraries. The students can then link their programs with the \texttt{nms} library, for example, by saying

\begin{verbatim}
$ link program,'nms'
\end{verbatim}

The last apostrophe is optional. If students have a problem with this, they are usually either forgetting the apostrophe or using the accent grave (backwards apostrophe).

I have installed a Fortran checker called \texttt{ftnchek} on both PV and Vax. This program, similar to \texttt{lint} for the C language, attempts to find programming errors in Fortran programs. It checks for things like variables that are used before they are defined, inconsistent subroutine parameters, and the like.

To use it on Vax, the following lines must be in \texttt{glogin.com}.

\begin{verbatim}
$ ftnchek :== $classlib:[math]ftnchek
$ define hlp$library classlib:[math]ftnchek.hlb
\end{verbatim}

I would also suggest adding the following lines to \texttt{glogin.com}

\begin{verbatim}
$ f*ortran :== fortran/list/check=(bounds,overflow,underflow)-
/extend_source/warn=(general,declarations)
\end{verbatim}

This tells the Fortran compiler to always behave as follows:

- generate a program listing;
- generate run-time error messages if the program exceeds array bounds (a common error) or a calculation overflows or underflows (see remark 1);
- allow lines longer than 72 characters;
- force students to explicitly declare every single variable they use (see remark 2).

If you like other defaults, you can of course change this line to suit your preferences. Two remarks about this setup:

(1) In my opinion, Fortran’s habit of accepting undeclared variables is responsible for a considerable portion of the students’ programming errors. The option \texttt{warn=declarations} will force them to declare everything. They may not like it, but it will often save them time in the end. Make sure you explain to them that they need to declare everything.

(2) Sometimes a program will generate lots of harmless underflows, which gets to be annoying after a while. To turn off underflow messages (after you are convinced that they are harmless), change the option \texttt{check=underflow} to \texttt{check=nounderflow}.

In my experience, most harmless underflow messages come from the \texttt{blas} routines. The Basic Linear Algebra Subroutines are for calculating the length of a vector, taking dot products of vectors, and stuff like that. They are used extensively by other routines for vector and matrix calculations. These \texttt{blas} routines are carefully written to insure that underflows are harmless. I don’t know why they were not carefully written to avoid underflow in the first place, but that is a different matter.

Finally, I find it useful to have a logon message for my students to remind them of deadlines, give further hints on programming assignments and things like that. An easy way to do this is to put the messages in a file and have this file listed every time a student logs on. I like to call this file \texttt{motd}. (don’t forget the period at the end). This comes from the UNIX operating system and stands for \textit{message of the day}.

If the file is not there, this causes an error, and the rest of the login file is skipped. Therefore, this \texttt{type} command should be the last thing in the login file, just in case you forget to create the message file. (This is called \textit{defensive programming}).

Now let us put all these things together. A suggested \texttt{glogin.com} file looks like this:

\begin{verbatim}
$ arc :== $classlib:[microstuff]arc
$ list :== @classlib:[math]list
\end{verbatim}
This file is stored on the Vax as classlib: [math]glogin.com. You can simply copy it to class:glogin.com and edit it to your particular preferences.

C.3. Online Discussion Groups

Mike Fletcher is our local Usenet administrator. Usenet distributes about 1,500 newsgroups on topics ranging from computer subjects to recipes, square dancing, movie reviews, etc. There are several newsreaders available to access these groups from PV, Vax or Wylibur. On PV, there are (at least) rn, xrn, nn and trn.

Mike can arrange to have local newsgroups for the math department created. You can create a group for a class and encourage students to ask questions and trade information there. The advantage of a newsgroup over e-mail or personal questions is that when you answer somebody’s question there, everybody else can see the question and the answer. This may cut down on questions, or maybe they will even answer each other’s questions. The group for Math 473 is called isu.math.473 and already exists, so you don’t have to have it created.

Students and faculty members may need some encouragement to start using this facility, but I believe that this is the trend of the future.