Principles of Scientific Computing

Jonathan Goodman

last revised April 20, 2006

Preface

This book grew out of a one semester first course in Scientific Computing for graduate students at New York University. It covers the basics that anyone doing scientific computing needs to know. In my view, these are: some mathematics, the most basic algorithms, a bit about the workings of a computer, and an idea how to build and use software for scientific computing applications. Students who have taken Scientific Computing are prepared for more specialized classes such as Computational Fluid Dynamics or Computational Statistics.

My original goal was to write a book that could be covered in an intensive one semester class, but the present book is a little bigger than that. I made many painful choices of material to leave out. Topics such as finite element analysis, constrained optimization, algorithms for finding eigenvalues, etc. are barely mentioned. In each case, I found myself unable to say enough about the topic to be helpful without crowding out other topics that I think everyone doing scientific computing should be familiar with.

The book requires a facility with the mathematics that is common to most quantitative modeling: multivariate calculus, linear algebra, basics of differential equations, and elementary probability. There is some review material here and suggestions for reference books, but nothing that would substitute for classes in the background material.

The student also will need to know or learn C or C++, and Matlab. In teaching this class I routinely have some students who learn programming as they go. The web site http://www.math.nyu.edu/faculty/goodman/ScientificComputing/ has materials to help the beginner get started with C/C++ or Matlab. It is possible to do the programming in Fortran, but students are discouraged from using a programming language, such as Java, Basic, or Matlab, not designed for efficient large scale scientific computing.

The book does not ask the student to use a specific programming environment. Most students use a personal computer (laptop) running Linux, OSX, or Windows. Many use the gnu C/C++ compiler while others use commercial compilers from, say, Borland or Microsoft. I discourage using Microsoft compilers because they are incompatible with the IEEE floating point standard. Most students use the student edition of Matlab, which runs fine on any of the platforms they are likely to use. There are shareware visualization tools that could be used in place of Matlab, including gnuplot. I discourage students from using Excel for graphics because it is designed for commercial rather than scientific visualization.

Many of my views on scientific computing were formed during my association with the remarkable group of faculty and graduate students at Serra House, the numerical analysis group of the Computer Science Department of Stanford University, in the early 1980's. I mention in particularly Marsha Berger, Petter Björstad, Bill Coughran, Gene Golub, Bill Gropp, Eric Grosse, Bob Higdon, Randy LeVeque, Steve Nash, Joe Oliger, Michael Overton, Robert Schreiber, Nick Trefethen, and Margaret Wright. Colleagues at the Courant Institute who have influenced this book include Leslie Greengard, Gene Isaacson, Peter Lax, Charlie Peskin, Luis Reyna, Mike Shelley, and Olof Widlund. I also acknowledge the lovely book *Numerical Methods* by Germund Dahlquist and Åke Björk. From

an organizational standpoint, this book has more in common with *Numerical Methods and Software* by Forsythe and Moler.

Contents

Chapter 1

Introduction

Most problem solving in science and engineering uses scientific computing. A scientist might devise a system of differential equations to model a physical system then use a computer to calculate their solutions. An engineer might develop a formula to predict cost as a function of several variables then use a computer to find the combination of variables that minimizes cost. A scientist or engineer not only needs to know the principles that allow him or her to model systems of interest, but also the computational methods needed to find out what the models predict.

Scientific computing is interesting and challenging partly because it is multidisciplinary. Direct physical reasoning may lead to a computational method that "works", in the sense that it produces something like the right answer if you wait long enough. But each discipline offers ways to make computations dramatically better. A well engineered computer code can be more *robust*, more likely to find an approximation to the answer in challenging cases. Mathematical analysis can quantify sources of error and find ways to reduce them. Clever algorithms may be faster than a naive "implementation" of a formula. The *performance* of a piece of software refers to the amount of real time it takes to execute a given sequence of computations. This often depends on details of the software and hardware that the programmer should be aware of. There are some tricks and habits of scientific programming that make it easier to avoid and detect mistakes – principles of software engineering specifically for scientific computing.

Modern software tools have a big impact on scientific computing practice. Interactive window based debuggers make basic programming much easier than it used to be. Specialized systems such as Matlab make certain computations, particularly those involving linear algebra and basic data manipulation, quick and convenient. Advanced scientific visualization systems make it possible to understand, check, and present the results of a computation. Performance tools help the developer find "bottlenecks" in a large code and suggest ways to improve them.

This book weaves together all these aspects of scientific computing. Most of the text is mathematics and algorithms, but each chapter has a *Software* section that discusses some of the "softer" aspects of scientific computing. The computational exercises ask the student not only to understand the mathematics and algorithms, but to construct small pieces of high quality scientific software. They nurture programming habits that are as important to overall success of scientific computing projects as basic mathematics.

This book gives only the briefest introduction to most parts of computational technique. Please do not think the things left out are not important. For example, anyone solving ordinary differential equations must know the stability theory of Dalhquist and others, which can be found in any serious book on numerical solution of ordinary differential equations. There are many variants of the FFT that are faster than the simple one in Chapter 7, more sophisticated kinds of spline interpolation, etc. The same applies to things like software engineering and scientific visualization.

Chapter 2

Sources of Error

In scientific computing, we never expect to get the exact answer. Inexactness is practically the definition of scientific computing. Getting the exact answer, generally with integers or rational numbers, is *symbolic computing*, an interesting but distinct subject. Suppose we are trying to compute the number A. The computer will produce an approximation, which we call $\widehat{A}$. This $\widehat{A}$ may agree with A to 16 decimal places, but the identity $A = \widehat{A}$ (almost) never is true in the mathematical sense, if only because the computer does not have an exact representation for A. For example, if we need to find x that satisfies the equation $x^2 - 175 = 0$, we might get 13 or 13.22876, depending on the computational method, but $\sqrt{175}$ cannot be represented exactly as a floating point number.

Four primary sources of error are: (i) *roundoff error*, (ii) *truncation error*, (iii) *termination of iterations*, and (iv) *statistical error* in Monte Carlo. We will estimate the sizes of these errors, either *a priori* from what we know in advance about the solution, or *a posteriori* from the computed (approximate) solutions themselves. Software development requires distinguishing these errors from those caused by outright bugs. In fact, the bug may not be that a formula is wrong in a mathematical sense, but that an approximation is not accurate enough. This chapter discuss floating point computer arithmetic and the IEEE floating point standard. The others are treated later.

Scientific computing is shaped by the fact that nothing is exact. A mathematical formula that would give the exact answer with exact inputs might not be robust enough to give an approximate answer with (inevitably) approximate inputs. Individual errors that were small at the source might combine and grow in the steps of a long computation. Such a method is *unstable*. A problem is *ill conditioned* if any computational method for it is unstable. *Stability* theory, which is modeling and analysis of error growth, is an important part of scientific computing.

2.1 Relative error, absolute error, and cancellation

The *absolute error* in approximating A by $\widehat{A}$ is $e = \widehat{A} - A$. The *relative error*, which is $\epsilon = e/A$, is usually more meaningful. These definitions may be restated as

$$\widehat{A} = A + e \quad \text{(absolute error)} \ , \qquad \widehat{A} = A \cdot (1 + \epsilon) \quad \text{(relative error)}. \tag{2.1}$$

For example, the absolute error in approximating $A = \sqrt{175}$ by $\widehat{A} = 13$ is $e = 13.22876 \cdots - 13 \approx .23$. The corresponding relative error is $e/A \approx .23/13.2 \approx .017 < 2\%$. Saying that the error is less than 2% is probably more informative than saying that the error is less than $.25 = 1/4$.

Relative error is a *dimensionless* measure of error. In practical situations, the desired A probably has units, such as seconds, meters, etc. If A is a length measured in meters, knowing $e \approx .23$ does not tell you whether e is large or small. If the correct length is half a meter, then .23 is a large error. If the correct length in meters is $13.22876\cdots$, then $\widehat{A}$ is off by less than 2%. If we switch to centimeters the error becomes 22.9. This may seem larger, but it still is less than 2% of the exact length, $1,322.876\cdots$ (in centimeters).

We often describe the accuracy of an approximation or measurement by saying how many decimal digits are correct. For example, Avogadro's number (the number of molecules in one mole) with two digits of accuracy is $N_0 \approx 6.0 \times 10^{23}$. We write 6.0 instead of just 6 to indicate that we believe the 0 is correct, that the true Avogadro's number is closer to 6×10^{23} than to 6.1×10^{23} or 5.9×10^{23}. With three digits the number is $N_0 \approx 6.02 \times 10^{23}$. In an absolute sense, the difference between $N_0 \approx 6 \times 10^{23}$ and $N_0 \approx 6.02 \times 10^{23}$ is 2×10^{21} molecules per mole, which may seem like a lot, but the relative error is about a third of one percent.

While relative error is more useful than absolute error, it also is more problematic. Relative error can grow through *cancellation*. For example, suppose we know $A = B - C$ and we have evaluated B and C to three decimal digits of accuracy. If the first two digits of B and C agree, then they *cancel* in the subtraction, leaving only one correct digit in A. If, say, $B \approx \widehat{B} = 2.38 \times 10^5$ and $C \approx \widehat{C} = 2.33 \times 10^5$, then $A \approx \widehat{A} = 5 \times 10^3$. This $\widehat{A}$ is probably off by more than 10% even though $\widehat{B}$ and $\widehat{C}$ had relative error less than 1%. *Catastrophic cancellation* is losing many digits in one subtraction. More subtle and more common is an accumulation of less dramatic cancellations over a series of steps.

2.2 Computer arithmetic

Error from inexact computer floating point arithmetic is called *roundoff error*. Roundoff error occurs in most floating point operations. Some computations involve no other approximations. For example, solving systems of linear equations using Gaussian elimination would give the exact answer in *exact arithmetic* (all computations performed exactly). Even these computations can be unstable and give wrong answers. Being exactly right in exact arithmetic does not imply being approximately right in floating point arithmetic.

Floating point arithmetic on modern computers is governed by the *IEEE floating point standard*. Following the standard, a floating point operation normally has relative error less than the *machine precision*, but of the same order of magnitude. The machine precision is $\epsilon_{\text{mach}} \approx 6 \cdot 10^{-8}$ for *single precision* (data type `float` in C), and $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$ for *double precision* (data type `double` in C). Let $A = B \bigcirc C$, with $\bigcirc$ standing for one of the arithmetic operations: addition ($A = B+C$), subtraction, multiplication, or division. With the same B and C, the computer will produce $\widehat{A}$ with relative error (2.1) that normally satisfies $|\epsilon| \leq \epsilon_{\text{mach}}$.

2.2.1 Introducing the standard

The *IEEE floating point standard* is a set of conventions for computer representation and processing of floating point numbers. Modern computers follow these standards for the most part. The standard has four main goals:

1. To make floating point arithmetic as accurate as possible.
2. To produce sensible outcomes in exceptional situations.
3. To standardize floating point operations across computers.
4. To give the programmer control over exception handling.

The standard specifies exactly how numbers are represented in hardware. The most basic unit of information that a computer stores is a *bit*, a variable whose value may be either 0 or 1. Bits are organized into 32 bit or 64 bit *words*, or *bit strings*. The number of 32 bit words is[1] $2^{32} = 2^2 \cdot 2^{30} \approx 4 \times (10^3)^3 = 4$ billion. A typical computer should take well under a minute to list all of them. A computer running at 1GHz in theory can perform one billion operations per second, though that may not be achieved in practice. The number of 64 bit words is about $1.6 \cdot 10^{19}$, which is too many to be listed in a year. A 32 bit floating point number is called *single precision* and has data type `float` in C/++. A 64 bit floating point number is called *double precision* and has data type `double`.

C/C++ also has data types `int` (for 32 bits) and `longint` (for 64 bits) that represent integers. Integer, or *fixed point* arithmetic, is very simple. With 32 bit integers, the $2^{32} \approx 4 \cdot 10^9$ distinct words represent that many consecutive integers, filling the range from about $-2 \cdot 10^9$ to about $2 \cdot 10^9$. Addition, subtraction, and multiplication are done exactly whenever the answer is within this range. The result is unpredictable when the answer is out of range (*overflow*). Results of integer division are rounded down to the nearest integer below the answer.

2.2.2 Representation of numbers, arithmetic operations

For scientific computing, integer arithmetic has two drawbacks. One is that there is no representation for numbers that are not integers. Also important is the small range of values. The number of dollars in the US national debt, several trillion (10^{12}), cannot be represented as a 32 bit integer but is easy to approximate in 32 bit floating point.

The standard assigns a real number value to each single precision or double precision bit string. On a calculator display, the expression:

$$-.2491\mathrm{E}-5$$

means $-2.491 \cdot 10^{-6}$. This expression consists of a sign bit, $s = -$, a mantissa, $m = 2491$ and an exponent, $e = -5$. The expression $s.m\mathrm{E}e$ corresponds to the number $s \cdot .m \cdot 10^e$. Scientists like to put the first digit of the mantissa on the left of the decimal point ($-2.491 \cdot 10^{-6}$) while calculators put the whole thing on the

[1]We use the approximation $2^{10} = 1024 \approx 10^3$.

right $(-.2491 \cdot 10^{-5})$. In base 2 (binary) arithmetic, the scientists' convention saves a bit, see below.

When the standard interprets a 32 bit word, the first bit is the *sign bit*, $s = \pm$. The next 8 bits form the *exponent*[2], e, and the remaining 23 bits determine the form the *fraction*, f. There are two possible signs, $2^8 = 256$ possible values of e (ranging from 0 to 255), and $2^{23} \approx 8$ million possible fractions. Normally a floating point number has the value

$$A = \pm 2^{e-127} \cdot (1.f)_2 \quad , \tag{2.2}$$

where f is base 2 and the notation $(1.f)_2$ means that the expression $1.f$ is interpreted in base 2. Note that the mantissa is $1.f$ rather than just the fractional part, f. Any number (except 0) can be normalized so that its base 2 mantissa has the form $1.f$. There is no need to store the "1." explicitly, which saves one bit.

For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot \left(1 + 2^{-2} + 2^{-4} + 2^{-5}\right) \\ &= 2^{11} \cdot (1 + (.01)_2 + (.0001)_2 + (.00001)_2) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Altogether, we have, using $11 = (1011)_2$,

$$2752 = +(1.01011)_2^{(1011)_2} \; .$$

Thus, we have sign $s = +$. The exponent is $e - 127 = 11$ so that $e = 138 = (10001010)_2$. The fraction is $f = (01011000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_{s}\underbrace{10001010}_{e}\underbrace{01011000000000000000000}_{f} \quad .$$

For arithmetic operations, the standard mandates the rule: *the exact answer, correctly rounded*. For example, suppose `x`, `y`, and `z` are computer variables of type `float`, and the computer executes the statement `x = y / z;`. Let B and C be the numbers that correspond to the 32 bit strings `y` and `z` using the standard (2.2). A number that can be represented exactly in form (2.2) using 32 bits is a (32 bit) *floating point number*. Clearly B and C are floating point numbers, but the exact quotient, $A = B/C$, probably is not. *Correct rounding* means finding the floating point number $\widehat{A}$ closest[3] to A. The computer is supposed to set the bit string `x` equal to the bit string representing $\widehat{A}$. For exceptions to this rule, see below.

[2]This a slight misnomer; the actual exponent is $e - 127$ (in single precision) exponent.

[3]Ties can happen. The accuracy of IEEE floating point arithmetic does not depend on how ties are resolved.

The exact answer correctly rounded rule implies that the only error in floating point arithmetic comes from rounding the exact answer, A, to the nearest floating point number, $\widehat{A}$. This rounding error is determined by the distance between floating point numbers. The greatest rounding is when A is half way between neighboring floating point numbers, B_- and B_+. For a floating point number of the form $B_- = (1.f_-)_2 \cdot 2^p$, the next larger floating point number is usually $B_+ = (1.f_+)_2 \cdot 2^p$, where we get f_+ from f_- by adding the smallest possible fraction, which is 2^{-23} for 23 bit single precision fractions. The *relative* size of the gap between B_- and B_+ is, after some algebra,

$$\gamma = \frac{B_+ - B_-}{B_-} = \frac{(1.f_+)_2 - (1.f_-)_2}{(1.f_-)_2} = \frac{2^{-23}}{(1.f_-)_2} .$$

The largest γ is given by the smallest denominator, which is $(1.0\cdots 0)_2 = 1$, which gives $\gamma_{max} = 2^{-23}$. The largest rounding error is half the gap size, which gives the single precision machine precision $\epsilon_{mach} = 2^{-24}$ stated above.

The 64 bit double precision floating point format allocates one bit for the sign, 11 bits for the exponent, and the remaining 52 bits for the fraction. Therefore its floating point precision is given by $\epsilon_{\text{mach}} = 2^{-53}$. Double precision arithmetic gives roughly 16 decimal digits of accuracy instead of 7 for single precision. There are 2^{11} possible exponents in double precision, ranging from 1023 to -1022. The largest double precision number is of the order of $2^{1023} \approx 10^{307}$. The largest single precision number is about $2^{126} \approx 10^{38}$. Not only is double precision arithmetic more accurate than single precision, but the range of numbers is far greater.

2.2.3 Exceptions

The extreme exponents, $e = 0$ and $e = 255$ in single precision ($e = 0$ and $e = 2^{11} - 1 = 2047$ in double), are not interpreted using (2.2). Instead, they have carefully engineered interpretations that make the IEEE standard distinctive. Numbers with $e = 0$ are *denormalized* and have the value

$$A = \pm 0.f \cdot 2^{-126} \text{ (single precision)}, \qquad A = \pm 0.f \cdot 2^{-1022} \text{ (double)}.$$

This feature is called *gradual underflow*. *Underflow* is the situation in which the result of an operation is not zero but is closer to zero than any normalized floating point number. In single precision, the smallest normalized positive floating point number is $A = (1.0\cdots 0)_2 \cdot 2^{-126}$. The nearest floating point number in the positive direction is $B_+ = (1.0\cdots 01)_2 \cdot 2^{-126}$. The nearest floating point number in the negative direction is the denormalized number $B_- = (0.1\cdots 11)_2 \cdot 2^{-126}$. The gap between A and B_+ and the gap between B_- and A both are $(0.0\cdots 01)_2 \cdot 2^{-126} = 2^{-126-23} = 2^{-149}$. Without denormalized numbers, A would have a gap of size 2^{-149} on the right and 2^{-126} (the space between 0 and A) on the left: the left gap would be $2^{23} \approx 4$ billion times larger than the gap on the right. Gradual underflow also has the consequence that two floating point numbers are equal, $x = y$, if and only if subtracting one from the other gives exactly zero.

The other extreme case, $e = 255$ in single precision, has two subcases, `inf` (for *infinity*) if $f = 0$ and `NaN` (for *Not a Number*) if $f \neq 0$. The C++ statement `cout << x;` produces[4] "`inf`" and "`NaN`" respectively. An arithmetic operation produces `inf` if the exact answer is larger than the largest floating point number, as does `1/x` if $x = \pm 0$. (Actually $1/+0 = +$`inf` and $1/-0 =$ `-inf`). Invalid operations such as `sqrt(-1.)`, `log(-4.)`, produce `NaN`. Any operation involving a `NaN` produces another `NaN`. It is planned that f will contain information about how or where in the program the `NaN` was created but this is not standardized yet. Operations with `inf` are common sense: $\texttt{inf} + finite = \texttt{inf}$, $\texttt{inf}/\texttt{inf} = \texttt{NaN}$, $finite/\texttt{inf} = 0$, $\texttt{inf} + \texttt{inf} = \texttt{inf}$, $\texttt{inf} - \texttt{inf} = \texttt{NaN}$.

A floating point arithmetic operation is an *exception* if the result is not a normalized floating point number. The standard mandates that a hardware *flag* (a binary bit of memory in the processor) should be *set* (given the value 1) when an exception occurs. There should be a separate flag for the underflow, `inf`, and `NaN` exceptions. The programmer should be able to specify what happens when an exception flag is set. Either the program execution continues without interruption or an *exception handler* procedure is called. The programmer should be able to write procedures that interface with the exception handler to find out what happened and take appropriate action. Only the most advanced and determined programmer will be able to do this. The rest of us have the worst of both: the exception handler is called, which slows the program execution but does nothing useful.

Many features of IEEE arithmetic are illustrated in Figure 2.1. Note that e^{204} gives `inf` in single precision but not in double precision because the range of values is larger in double precision. We see that `inf` and `NaN` work as promised. The main rule, "exact answer correctly rounded", explains why adding pairs of floating point numbers is commutative: the mathematical sums are equal so they round to the same floating point number. This does not force addition to be associative, which it is not. Multiplication also is commutative but not associative. The division operator gives integer or floating point division depending on the types of the operands. Integer arithmetic *truncates* the result to the next lower integer rather than rounding it to the nearest integer.

```
// A program that explores floating point arithmetic in the IEEE
// floating point standard.  The source code is SourcesOfError.C.

#include <iostream.h>
#include <math.h>

int main() {
```

[4] Microsoft, in keeping with its pattern of maximizing incompatibility, gives something different.

```
float  xs, ys, zs, ws;  // Some single precision variables.
double yd;              // A double precision variable.


xs = 204.;         // Take an exponential that is out of range.
ys = exp(xs);
cout << "The single precision exponential of " << xs <<
        " is " << ys << endl;
yd = exp ( xs );   // In double precision, it is in range.
cout << "The double precision exponential of " << xs <<
        " is " << yd << endl;


zs = xs / ys;      // Divide a normal number by infinity.
cout << xs << " divided by " << ys  <<
                   " gives " << zs << endl;

ws = ys;           // Divide infinity by infinity.
zs = ws / ys;
cout << ws << " divided by " << ys << " gives " << zs << endl;

zs = sqrt( -1.) ;  // sqrt(-1) should be NaN.
cout << "sqrt(-1.) is " << zs << endl;

ws = xs + zs;      // Add NaN to a normal number.
cout << xs << " + " << zs << " gives " << ws << endl;

xs =        sin(1.); // Some generic single precision numbers.
ys = 100.  *sin(2.);
zs = 10000.*sin(3.);
float xsPys, ysPxs, xsPzs, zsPxs; // xsPzx holds xs + zs, etc.
xsPys = xs + ys;
ysPxs = ys + xs;   // Try commuting pairs.
xsPzs = xs + zs;
zsPxs = zs + xs;
if ( ( xsPys == ysPxs ) && ( xsPzs == zsPxs ) )
   cout << "Adding " << xs << "  " << ys << " and "<< zs <<
           " in pairs commutes." << endl;
  else
    cout << "Adding " << xs << "  " << ys << " and "<< zs <<
            " in pairs does not commute." << endl;

float xsPysPzs, ysPzsPxs;     // Test for associativity.
xsPysPzs = ( xs + ys ) + zs;
ysPzsPxs = ( ys + zs ) + xs;
if ( xsPysPzs == ysPzsPxs )
```

```
     cout << "Adding " << xs << "  " << ys << " and "<< zs <<
             " is associative." << endl;
   else
      cout << "Adding " << xs << "  " << ys << " and "<< zs <<
              " is not associative." << endl;

int xi, yi; // Some integer variables.
xi =  9;    // Compute the quotient using integer
yi = 10;    // and floating point arithmetic.
zs = xi/yi;
ws = ( (float) xi ) / ( (float) yi ); // Convert, then divide.
cout << "Integer division of " << xi << " by " << yi <<
    " gives " << zs << ".  " <<
    " Floating point gives " << ws << endl;

return(0);

}
```

Figure 2.1: *A program that illustrates some of the features of arithmetic using the IEEE floating point standard.*

2.3 Truncation error

Truncation error is the error in analytical approximations such as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \ . \tag{2.3}$$

This is not an exact formula, but it can be a useful approximation. We often think of truncation error as arising from truncating a Taylor series. In this case, the Taylor series formula,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \cdots \ ,$$

is truncated by neglecting all the terms after the first two on the right. This leaves the approximation

$$f(x+h) \approx f(x) + hf'(x) \ ,$$

which can be rearranged to give (2.3). Truncation usually is the main source of error in numerical integration or solution of differential equations. The analysis of truncation error using Taylor series will occupy the next two chapters.

h	.3	.01	10^{-5}	10^{-8}	10^{-10}
$\widehat{f'}$	6.84	5.48	5.4366	5.436564	5.436562
e_{tot}	1.40	$4.10 \cdot 10^{-2}$	$4.08 \cdot 10^{-5}$	$-5.76 \cdot 10^{-8}$	$-1.35 \cdot 10^{-6}$

Figure 2.2: Estimates of $f'(x)$ using (2.3). The error is e_{tot}, which results from truncation and roundoff error. Roundoff error is apparent only in the last two columns.

As an example, we take $f(x) = xe^x$, $x = 1$, and several h values. The truncation error is

$$e_{tr} = \frac{f(x+h) - f(x)}{h} - f'(x) \; .$$

In Chapter 3 we will see that (in exact arithmetic) e_{tr} roughly is proportional to h for small h. The numbers in Figure 2.3 were computed in double precision floating point arithmetic. The total error, e_{tot}, is a combination of truncation and roundoff error. Roundoff error is significant for the smallest h values: for $h = 10^{-8}$ the error is no longer proportional to h; by $h = 10^{-10}$ the error has increased. Such small h values are rare in a practical calculation.

2.4 Iterative Methods

Suppose we want to find A by solving an equation. Often it is impossible to find a formula for the solution. Instead, *iterative methods* construct a sequence of approximate solutions, A_n, for $n = 1, 2, \ldots$. Hopefully, the approximations *converge* to the right answer: $A_n \to A$ as $n \to \infty$. In practice, we must stop the iteration process for some large but finite n and accept A_n as the approximate answer.

For example, suppose we have a $y > 0$ and we want to find x with $xe^x = y$. There is not a formula for x, but we can write a program to carry out the iteration: $x_1 = 1$, $x_{n+1} = \ln(y) - \ln(x_n)$. The numbers x_n are *iterates*. The limit $x = \lim_{n\to\infty} x_n$ (if it exists), is a *fixed point* of the iteration, i.e. $x = \ln(y) - \ln(x)$, which implies $xe^x = y$. Figure 2.4 demonstrates the convergence of the iterates in this case with $y = 10$. The *initial guess* is $x_1 = 1$. After 20 iterations, we have $x_{20} \approx 1.74$. The error is $e_{20} \approx 2.3 \cdot 10^{-5}$, which might be small enough, depending on the application.

After 67 iterations the relative error is $(x_{67} - x)/x \approx 2.2 \cdot 10^{-16}/1.75 \approx 1.2 \cdot 10^{-16}$, which is only slightly larger than double precision machine precision $\epsilon_{mach} \approx 1.1 \cdot 10^{-16}$. This shows that supposedly approximate iterative methods can be as accurate as *direct* methods that would be exact in exact arithmetic. It would be a surprising fluke for even a direct method to achieve better than machine precision because even they are subject to roundoff error.

n	1	3	6	10	20	67
x_n	1	1.46	1.80	1.751	1.74555	1.745528
e_n	$-.745$	$-.277$	$5.5 \cdot 10^{-2}$	$5.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-5}$	$2.2 \cdot 10^{-16}$

Figure 2.3: Iterates of $x_{n+1} = \ln(y) - \ln(x_n)$ illustrating convergence to a limit that satisfies the equation $xe^x = y$. The error is $e_n = x_n - x$. Here, $y = 10$.

n	10	100	10^4	10^6	10^6	10^6
$\widehat{A}$	.603	.518	.511	.5004	.4996	.4991
error	.103	$1.8 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$4.4 \cdot 10^{-4}$	$-4.0 \cdot 10^{-4}$	$-8.7 \cdot 10^{-4}$

Figure 2.4: Statistical errors in a demonstration Monte Carlo computation.

2.5 Statistical error in Monte Carlo

Monte Carlo means using random numbers as a computational tool. For example, suppose[5] $A = E[X]$, where X is a random variable with some known distribution. *Sampling* X means using the computer random number generator to create independent random variables X_1, X_2, ..., each with the distribution of X. The simple Monte Carlo method would be to generate n such samples and calculate the *sample mean*:

$$A \approx \widehat{A} = \frac{1}{n} \sum_{k=1}^{n} X_k \; .$$

The difference between $\widehat{A}$ and A is *statistical error*. A theorem in probability, the *law of large numbers*, implies that $\widehat{A} \to A$ as $n \to \infty$. Monte Carlo statistical errors typically are larger than roundoff or truncation errors. This makes Monte Carlo a method of last resort, to be used only when other methods are not practical.

Figure 2.5 illustrates the behavior of this Monte Carlo method for the random variable $X = \frac{3}{2}U^2$ with U uniformly distributed in the interval $[0, 1]$. The exact answer is $A = E[X] = \frac{3}{2}E[U^2] = .5$. The value $n = 10^6$ is repeated to illustrate the fact that statistical error is random (see Chapter 9 for a clarification of this). The errors even with a million samples are much larger than those in the right columns of Figures 2.3 and 2.4.

2.6 Error propagation and amplification

Errors can be amplified as they *propagate* through a computation. For example, suppose the divided difference (2.3) is part of a long calculation:

```
   .    .
f1 = . . . ;  \\ approx of f(x)
```

[5] $E[X]$ is the *expected value* of X. Chapter 9 has some review of probability.

```
f2 = . . . ;  \\ approx of f(x+h)
    .   .
fPrimeHat = ( f2 - f1 ) / h ;  \\ approx of derivative
```

It is unlikely that $f_1 = \widehat{f(x)} \approx f(x)$ is exact. Many factors may contribute to the errors $e_1 = f_1 - f(x)$ and $e_2 = f_2 - f(x+h)$. There are three contributions to the final error in f':

$$\widehat{f}' - f' = e_r + e_{tr} + e_{pr} \ . \tag{2.4}$$

One is the roundoff error in evaluating `( f2 - f1 ) / h` in floating point

$$\widehat{f}' = \frac{f_2 - f_1}{h} + e_r \ . \tag{2.5}$$

The truncation error in the difference quotient approximation is

$$\frac{f(x+h) - f(x)}{h} - f' = e_{tr} \ . \tag{2.6}$$

The *propagated error* comes from using inexact values of $f(x+h)$ and $f(x)$:

$$\frac{f_2 - f_1}{h} - \frac{f(x+h) - f(x)}{h} = \frac{e_2 - e_1}{h} = e_{pr} \ . \tag{2.7}$$

If we add (2.5), (2.6), and (2.7), and simplify, we get the formula (2.4).

A stage of a long calculation creates some errors and propagates errors from earlier stages, possibly with amplification. In this example, the difference quotient evaluation introduces truncation and roundoff error. Also, e_1 and e_2 represent errors generated at earlier stages when $f(x)$ and $f(x+h)$ were evaluated. These errors, in turn, could have several sources, including inaccurate x values and roundoff in the code evaluating $f(x)$. According to (2.7), the difference quotient propagates these and amplifies them by a factor of $1/h$. A typical value $h = .01$ could amplify incoming errors e_1 and e_2 by a factor of 100.

This increase in error by a large factor in one step is an example of *catastrophic cancellation*. If the numbers $f(x)$ and $f(x_h)$ are nearly equal, the difference can have much less relative accuracy than the numbers themselves. More common and more subtle is *gradual error growth* over a long sequence of computational steps. Exercise 2.12 has an example in which the error roughly doubles at each stage. Starting from double precision roundoff level, the error after 30 steps is negligible but the error after 60 steps is larger than the answer.

An algorithm is *unstable* if its error mainly comes from amplification. This *numerical instability* can be hard to discover by standard debugging techniques that look for the first place something goes wrong, particularly if there is gradual error growth.

Mathematical *stability theory* in scientific computing is the search for gradual error growth in computational algorithms. It focuses on propagated error only, ignoring the original sources of error. For example, Exercise 8 involves the backward recurrence $f_{k-1} = f_{k+1} - f_k$. In a stability analysis, we would assume that the subtraction is performed exactly and that the error in f_{k-1} is entirely due to errors in f_k and f_{k+1}. That is, if $\widehat{f_k} = f_k + e_k$ is the computer approximation, then the e_k satisfy the mathematical relation $e_{k-1} = e_{k+1} - e_k$, which is the *error propagation equation*. We then would use the theory of recurrence relations to see whether the e_k can grow relative to the f_k as k decreases. If this error growth is possible, it will happen in practically any computation.

2.7 Condition number and ill conditioned problems

The *condition number* of a computational problem measures the sensitivity of the answer to small changes in the data. If κ is the condition number, then we expect error at least $\kappa \cdot \epsilon_{mach}$, regardless of the computational algorithm. A problem with large condition number is *ill conditioned*. For example, if $\kappa > 10^7$, then there probably is no algorithm that gives anything like the right answer in single precision arithmetic. Condition numbers as large as 10^7 or 10^{16} can and do occur in practice.

The definition of κ is simplest when the answer is a single number that depends on a single scalar variable, x: $A = A(x)$. A change in x causes a change in A: $\Delta A = A(x + \Delta x) - A(x)$. The condition number measures the relative change in A caused by a small relative change of x:

$$\left| \frac{\Delta A}{A} \right| \approx \kappa \left| \frac{\Delta x}{x} \right| \quad . \tag{2.8}$$

Any algorithm that computes $A(x)$ must round x to the nearest floating point number, $\widehat{x}$. This creates a relative error (assuming x is within the range of normalized floating point numbers) of $|\Delta x/x| = |(\widehat{x} - x)/x| \sim \epsilon_{mach}$. If the rest of the computation were done exactly, the computed answer would be $\widehat{A}(x) = A(\widehat{x})$ and the relative error would be (using (2.8))

$$\left| \frac{\widehat{A}(x) - A(x)}{A(x)} \right| = \left| \frac{A(\widehat{x}) - A(x)}{A(x)} \right| \approx \kappa \left| \frac{\Delta x}{x} \right| \sim \kappa \epsilon_{mach} \, . \tag{2.9}$$

If A is a differentiable function of x with derivative $A'(x)$, then, for small Δx, $\Delta A \approx A'(x)\Delta x$. With a little algebra, this gives (2.8) with

$$\kappa = \left| A'(x) \cdot \frac{x}{A(x)} \right| \quad . \tag{2.10}$$

This analysis argues that any computational algorithm for an ill conditioned problem must be unstable. Even if $A(x)$ is evaluated exactly, relative errors in the input of size ϵ are amplified by a factor of κ. The formulas (2.9) and (2.10) represent an absolute lower bound for the accuracy of any computational algorithm. An ill conditioned problem is not going to be solved accurately, period.

The formula (2.10) gives a dimensionless κ because it measures relative sensitivity. The extra factor $x/A(x)$ removes the units of x and A. Absolute sensitivity is is just $A'(x)$. Note that both sides of our starting point (2.8) are dimensionless with dimensionless κ.

As an example consider the problem of evaluating $A(x) = R\sin(x)$. The condition number formula (2.10) gives

$$\kappa(x) = \left|\cos(x) \cdot \frac{x}{\sin(x)}\right| .$$

Note that the problem remains well conditioned (κ is not large) as $x \to 0$, even though $A(x)$ is small when x is small. For extremely small x, the calculation could suffer from underflow. But the condition number blows up as $x \to \pi$, because small relative changes in x lead to much larger relative changes in A. This illustrates quirk of the condition number definition: typical values of A have the order of magnitude R and we can evaluate A with error much smaller than this, but certain individual values of A may not be computed to high relative precision. In most applications that would not be a problem.

There is no perfect definition of condition number for problems with more than one input or output. Suppose at first that the single output $A(x)$ depends on n inputs $x = (x_1, \ldots, x_n)$. Of course A may have different sensitivities to different components of x. For example, $\Delta x_1/x_1 = 1\%$ may change A much more than $\Delta x_2/x_2 = 1\%$. If we view (2.8) as saying that $|\Delta A/A| \approx \kappa\epsilon$ for $|\Delta x/x| = \epsilon$, a worst case multicomponent generalization could be

$$\kappa = \frac{1}{\epsilon} \max \left|\frac{\Delta A}{A}\right| \quad , \quad \text{where} \quad \left|\frac{\Delta x_k}{x_k}\right| \leq \epsilon \;\text{ for all } k.$$

We seek the worst case[6] Δx. For small ϵ we write

$$\Delta A \approx \sum_{k=1}^{n} \frac{\partial A}{\partial x_k} \Delta x_k ,$$

then maximize subject to the constraint $|\Delta x_k| \leq \epsilon\,|x_k|$ for all k. The maximum occurs at $\Delta x_k = \pm\epsilon x_k$, which (with some algebra) leads to one possible generalization of (2.10):

$$\kappa = \sum_{k=1}^{n} \left|\frac{\partial A}{\partial x_k} \cdot \frac{x_k}{A}\right| . \tag{2.11}$$

This formula is useful if the inputs are known to similar relative accuracy, which could happen even when the x_k have different orders of magnitude or different units. Condition number for multivariate problems is discussed using matrix norms in Section 4.3. The analogue of (2.10) is (4.29).

[6]As with rounding, typical errors tend to be order of the worst case error.

2.8 Software

Each chapter of this book has a *Software* section. Taken together they form a mini course in software for scientific computing. The material ranges from simple tips to longer discussions of bigger issues. The programming exercises illustrate the chapter's software principles as well as the mathematical material from earlier sections.

Scientific computing projects fail because of bad software as often as they fail because of bad algorithms. The principles of scientific software are less precise than the mathematics of scientific computing, but are just as important. There is a set of principles for scientific programming that goes beyond those for general programming (modular design, commenting, etc.). Projects are handsomely rewarded for extra efforts and care taken to do the software "right".

2.8.1 Floating point numbers are (almost) never equal

Because of inexact floating point arithmetic, two numbers that should be equal in exact arithmetic often are not equal in the computer. In general, an equality test between two variables of type `float` or `double` is a mistake. A striking illustration of this can happen with Intel processor chips, where variables of type `double` are stored on the chip in 80 bit registers but in memory with the standard 64. Moving a variable from the register to memory loses the extra bits. Thus, a program can execute the instruction `y1 = y2;` and then do not reassign either `y1` or `y2`, but later `( y1 == y2 )` may evaluate to `false` because `y1` but not `y2` was copied from register to memory.

A common mistake in this regard is to use floating point comparisons to regulate a loop. Figure 2.5 illustrates this. In exact arithmetic this would give the desired n iterations. Because of floating point arithmetic, after the n^{th} iteration, the variable `t` may be equal to `tFinal` but is much more likely to be above or below due to roundoff error. It is impossible to predict which way the roundoff error will go. We do not know whether this code will execute the `while` loop body n or $n+1$ times. Figure 2.6 uses exact integer arithmetic to guarantee n executions of the `for` loop body.

2.8.2 Plotting

Careful visualization is a key step in understanding any data. Pictures can be more informative than tables of numbers. Explore and understand the data by plotting it in various ways. There are many ways to visualize data, simple graphs, surface plots, contour and color plots, movies, etc. We discuss only simple graphs here. Here are some things to keep in mind.

Learn your system and your options. Find out what visualization tools are available or easy to get on your system. Choose a package designed for scientific visualization, such as Matlab or Gnuplot, rather than one designed for commercial presentations such as Excel. Learn the options such as line style (dashes, thickness, color, symbols), labeling, etc.

```
double tStart, tFinal, t, dt;
int    n;
tStart = . . .  ;   // Some code that determines the start
tFinal = . . .  ;   // and ending time and the number of
n      = . . .  ;   // equal size steps.
dt     = ( tFinal - tStart ) / n;  // The size of each step.
for ( t = tStart, t < tFinal, t+= dt )
   { . . . }        // Body of the loop does not assign t.
```

Figure 2.5: *A code fragment illustrating a pitfall of using a floating point variable to regulate a* `while` *loop.*

```
double tStart, tFinal, t, dt;
int    n,    , i;
tStart = . . .  ;   // Some code that determines the start
tFinal = . . .  ;   // and ending time and the number of
n      = . . .  ;   // equal size steps.
dt     = ( tFinal - tStart ) / n;  // The size of each step.
for ( i = 0, i < n, i++ )
   { t = tStart + i*dt;  // In case the value of t is needed
          . . . }        // in the loop body.
```

Figure 2.6: *A code fragment using an integer variable to regulate the loop of Figure 2.5.*

Figure 2.7: *Plots of the first n Fibonacci numbers, linear scale on the left, log scale on the right*

Use scripting and other forms of automation. You will become frustrated typing several commands each time you adjust one detail of the plot. Instead, assemble the sequence of plot commands into a script.

Frame the plot. The Matlab `plot` function with values in the range from 1.2 to 1.22 will use a vertical scale from 0 to 2 and plot the data as a nearly horizontal line, unless you tell it otherwise. Figure 2.7, presents the first 70 Fibonacci numbers. The Fibonacci numbers, f_i, are defined by $f_0 = f_1 = 1$, and $f_{i+1} = f_i + f_{i-1}$, for $i \geq 1$. On the linear scale, f_1 through f_{57} sit on the horizontal axis, indistinguishable *to plotting accuracy* from zero. The log plot shows how big each of the 70 numbers is. It also makes it clear that $\log(f_i)$ is nearly proportional to i, which implies (if $\log(f_i) \approx a + bi$, then $f_i \approx cd^i$) that the f_i are approximately exponential. If we are interested in the linear scale plot, we can edit out the useless left part of the graph by plotting only from $n = 55$ to $n = 70$.

Combine curves you want to compare into a single figure. Stacks of graphs are as frustrating as arrays of numbers. You may have to scale different curves differently to bring out the relationship they have to each other. If the curves are supposed to have a certain slope, include a line with that slope. If a certain x or y value is important, draw a horizontal or vertical line to mark it in the figure. Use a variety of line styles to distinguish the curves. Exercise 9 illustrates some of these points.

Make plots self–documenting. Figure 2.7 illustrates mechanisms in Matlab for doing this. The horizontal and vertical axes are labeled with values and text. In the third plot, the simpler command `plot(f(iStart:n))` would have labeled the horizontal axis from 1 to 15 (very misleading) instead of 55 to 70. Parameters from the run, in this case just n, are embedded in the title.

The Matlab script that made the plots of Figure 2.7 is in Figure 2.8. The only real parameters are n, the largest i value, and whether the plot is on a linear or log scale. Both of those are recorded in the plot. Note the convenience and clarity of not hard wiring $n = 70$. It would take just a moment to make plots up to $n = 100$.

2.9 Further reading

The idea for starting a book on computing with a discussion of sources of error comes from the book *Numerical Methods and Software* by David Kahaner, Cleve Moler, and Steve Nash. Another interesting version is in *Scientific Computing* by Michael Heath. My colleague, Michael Overton, has written a nice short book *IEEE Floating Point Arithmetic.*

```
  % Matlab code to generate and plot Fibonacci numbers.

clear f     % If you decrease the value of n, it still works.
n    = 70;  % The number of Fibonacci numbers to compute.
fi   = 1;   % Start with f0 = f1 = 1, as usual.
fim1 = 1;
f(1) = fi;  % Record f(1) = f1.
for i = 2:n
   fip1 = fi + fim1;  % f(i+1) = f(i) + f(i-1) is the recurrence
   fim1 = fi;         % relation that defines . . .
   fi   = fip1;       % the Fibonacci numbers.
   f(i) = fi;         % Record f(i) for plotting.
end

plot(f)
xlabel('i')           % The horizontal and vertical axes are
ylabel('f')           % i and f respectively.
topTitle = sprintf('Fibonacci up to n = %d',n);
                                       % Put n into the title.
title(topTitle)
text(n/10, .9*f(n), 'Linear scale');
grid                % Make it easier to read values in the plot.
set ( gcf, 'PaperPosition', [.25 2.5 3.2 2.5]);
                    % Print a tiny image of the plot for the book.
print -dps FibLinear_se
```

Figure 2.8: *Matlab code to calculate and plot Fibonacci numbers.*

2.10 Exercises

1. It is common to think of $\pi^2 = 9.87$ as approximately ten. What are the absolute and relative errors in this approximation?

2. If `x` and `y` have type `double`, and `( ( x - y ) >= 10 )` evaluates to `TRUE`, does that mean that `y` is not a good approximation to `x` in the sense of relative error?

3. Show that $f_{jk} = \sin(x_0 + (j-k)\pi/3)$ satisfies the recurrence relation
$$f_{j,k+1} = f_{j,k} - f_{j+1,k} \; . \tag{2.12}$$
We view this as a formula that computes the f values on level $k+1$ from the f values on level k. Let $\widehat{f}_{jk}$ for $k \geq 0$ be the floating point numbers that come from implementing $f_{j0} = \sin(x_0 + j\pi/3)$ and (2.12) (for $k > 0$) in double precision floating point. If $\left|\widehat{f}_{jk} - f_{jk}\right| \leq \epsilon$ for all j, show that $\left|\widehat{f}_{j,k+1} - f_{j,k+1}\right| \leq 2\epsilon$ for all j. Thus, if the level k values are very accurate, then the level $k+1$ values still are pretty good.

   Write a program (C/C++ or Matlab) that computes $e_k = \widehat{f}_{0k} - f_{0k}$ for $1 \leq k \leq 60$ and $x_0 = 1$. Note that f_{0n}, a single number on level n, depends on $f_{0,n-1}$ and $f_{1,n-1}$, two numbers on level $n-1$, and so on down to n numbers on level 0. Print the e_k and see whether they grow monotonically. Plot the e_k on a linear scale and see that the numbers seem to go bad suddenly at around $k = 50$. Plot the e_k on a log scale. For comparison, include a straight line that would represent the error if it were exactly to double each time.

4. What are the possible values of `k` after the `for` loop is finished?

```
float  x = 100*rand() + 2;
int    n = 20, k = 0;
float dy = x/n;
for ( float y = 0; y < x; y += dy; ) {
   k++;  /* body does not change x, y, or dy  */ }
```

5. We wish to evaluate the function $f(x)$ for x values around 10^{-3}. We expect f to be about 10^5 and f' to be about 10^{10}. Is the problem too ill conditioned for single precision? For double precision?

6. Show that in the IEEE floating point standard with any number of fraction bits, ϵ_{mach} essentially is the largest floating point number, ϵ, so that $1+\epsilon$ gives 1 in floating point arithmetic. Whether this is exactly equivalent to the definition in the text depends on how ties are broken in rounding, but the difference between the two definitions is irrelevant (show this).

7. Starting with the declarations

```
      float x, y, z, w;
const float oneThird = 1/ (float) 3;
const float oneHalf  = 1/ (float) 2;
                  // const means these never are reassigned
```

we do lots of arithmetic on the variables `x, y, z, w`. In each case below, determine whether the two arithmetic expressions result in the same floating point number (down to the last bit) as long as no `NaN` or `inf` values or denormalized numbers are produced.

(a)

```
( x * y ) + ( z - w )
( z - w ) + ( y * x )
```

(b)

```
( x +   y ) + z
  x + ( y   + z )
```

(c)

```
  x * oneHalf + y * oneHalf
( x + y ) * oneHalf
```

(d)

```
  x * oneThird + y * oneThird
( x + y ) * oneThird
```

8. The *fibonacci numbers*, f_k, are defined by $f_0 = 1$, $f_1 = 1$, and

$$f_{k+1} = f_k + f_{k-1} \tag{2.13}$$

for any integer $k > 1$. A small perturbation of them, the *pib numbers* ("p" instead of "f" to indicate a perturbation), p_k, are defined by $p_0 = 1$, $p_1 = 1$, and

$$p_{k+1} = c \cdot p_k + p_{k-1}$$

for any integer $k > 1$, where $c = 1 + \sqrt{3}/100$.

(a) Plot the f_n and p_n in one together on a log scale plot. On the plot, mark $1/\epsilon_{mach}$ for single and double precision arithmetic. This can be useful in answering the questions below.

(b) Rewrite (2.13) to express f_{k-1} in terms of f_k and f_{k+1}. Use the computed f_n and f_{n-1} to recompute f_k for $k = n-2, n-3, \ldots, 0$. Make a plot of the difference between the original $f_0 = 1$ and the recomputed $\widehat{f}_0$ as a function of n. What n values result in no accuracy for the recomputed f_0? How do the results in single and double precision differ?

(c) Repeat b. for the pib numbers. Comment on the striking difference in the way precision is lost in these two cases. Which is more typical? *Extra credit*: predict the order of magnitude of the error in recomputing p_0 using what you may know about recurrence relations and what you should know about computer arithmetic.

9. The binomial coefficients, $a_{n,k}$, are defined by

$$a_{n,k} = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

To compute the $a_{n,k}$, for a given n, start with $a_{n,0} = 1$ and then use the recurrence relation $a_{n,k+1} = \frac{n-k}{k+1} a_{n,k}$.

(a) For a range of n values, compute the $a_{n,k}$ this way, noting the largest $a_{n,k}$ and the accuracy with which $a_{n,n} = 1$ is computed. Do this in single and double precision. Why is roundoff not a problem here as it was in problem 8? Find n values for which $\widehat{a}_{n,n} \approx 1$ in double precision but not in single precision. How is this possible, given that roundoff is not a problem?

(b) Use the algorithm of part (a) to compute

$$E(k) = \frac{1}{2^n} \sum_{k=0}^{n} k a_{n,k} = \frac{n}{2} \quad . \tag{2.14}$$

Write a program without any safeguards against overflow or zero divide (this time only!)[7]. Show (both in single and double precision) that the computed answer has high accuracy as long as the intermediate results are within the range of floating point numbers. As with (a), explain how the computer gets an accurate, small, answer when the intermediate numbers have such a wide range of values. Why is cancellation not a problem? Note the advantage of a wider range of values: we can compute $E(k)$ for much larger n in double precision. Print $E(k)$ as computed by (2.14) and $M_n = \max_k a_{n,k}$. For large n, one should be `inf` and the other `NaN`. Why?

(c) For fairly large n, plot $a_{n,k}/M_n$ as a function of k for a range of k chosen to illuminate the interesting "bell shaped" behavior of the $a_{n,k}$ near $k = n/2$. Combine the curves for $n = 10$, $n = 20$, and $n = 50$ in a single plot. Choose the three k ranges so that the curves are close to each other. Choose different line styles for the three curves.

[7] One of the purposes of the IEEE floating point standard was to allow a program with overflow or zero divide to run and print results.

Chapter 3

Local Analysis

Among the most common computational tasks are differentiation, interpolation, and integration. The simplest methods used for these operations are *finite difference* approximations for derivatives, *low order polynomial* interpolation, and *panel method* integration. Finite difference formulas, integration rules, and interpolation form the core of most scientific computing projects that involve solving differential or integral equations.

The finite difference formulas (3.14) range from simple *low order* approximations (3.14a) – (3.14c) to not terribly complicated *high order* methods such as (3.14e). Figure 3.2 illustrates that high order methods can be far more accurate than low order ones. This can make the difference between getting useful answers and not in serious large scale applications. The methods here will enable the reader to design professional quality highly accurate methods rather than relying on simple but often inefficient low order ones.

Many methods for these problems involve a *step size*, h. For each h there is an approximation[1] $\widehat{A}(h) \approx A$. We say $\widehat{A}$ is *consistent* if $\widehat{A}(h) \to A$ as $h \to 0$. For example, we might estimate $A = f'(x)$ using the finite difference formula (3.14a): $\widehat{A}(h) = (f(x+h) - f(x))/h$. This is consistent, as $\lim_{h\to 0} \widehat{A}(h)$ is the definition of $f'(x)$. The accuracy of the approximation depends on f, but the *order of accuracy* does not.[2] The approximation is *first order accurate* if the error is nearly proportional to h for small enough h. It is *second order* if the error goes like h^2. When h is small, $h^2 \ll h$, so approximations with a higher order of accuracy can be much more accurate.

The design of difference formulas and integration rules is based on *local analysis*, approximations to a function f about a *base point* x. These approximations consist of the first few terms of the *Taylor series* expansion of f about x. The *first order* approximation is

$$f(x+h) \approx f(x) + hf'(x) \; . \tag{3.1}$$

The *second order* approximation is more complicated and more accurate:

$$f(x+h) \approx f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 \; . \tag{3.2}$$

Figure 3.1 illustrates the first and second order approximations. *Truncation error* is the difference between $f(x+h)$ and one of these approximations. For example, the truncation error for the first order approximation is

$$f(x) + f'(x)h - f(x+h) \; .$$

To see how Taylor series are used, substitute the approximation (3.1) into the finite difference formula (3.14a). We find

$$\widehat{A}(h) \approx f'(x) = A \; . \tag{3.3}$$

[1] In the notation of Chapter 2, $\widehat{A}$ is an estimate of the desired answer, A.

[2] The nominal order of accuracy may be achieved if f is smooth enough, a point that is important in many applications.

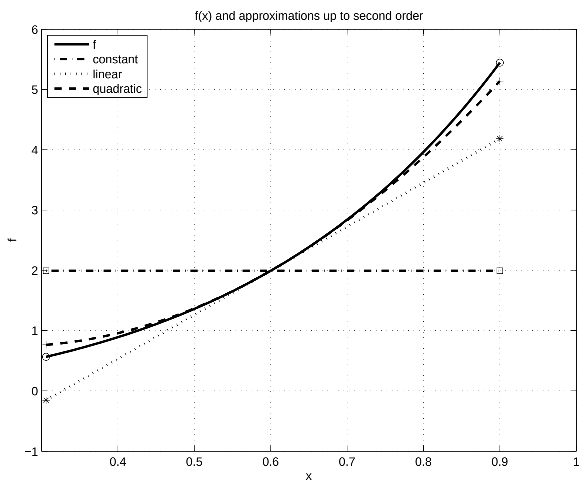

Figure 3.1: *Plot of* $f(x) = xe^{2x}$ *together with Taylor series approximations of order zero, one, and two. The base point is* $x = .6$ *and* h *ranges from* $-.3$ *to* $.3$. *The symbols at the ends of the curves illustrate convergence at fixed* h *as the order increases. Also, the higher order curves make closer contact with* $f(x)$ *as* $h \to 0$.

The more accurate Taylor approximation (3.2) allows us to estimate the error in (3.14a). Substituting (3.2) into (3.14a) gives

$$\widehat{A}(h) \approx A + A_1 h \ , \quad A_1 = \frac{1}{2} f''(x) \ . \tag{3.4}$$

This *asymptotic error expansion* is an estimate of $\widehat{A} - A$ for a given f and h. It shows that the error is roughly proportional to h for small h. This understanding of truncation error leads to more sophisticated computational strategies. *Richardson extrapolation* combines $\widehat{A}(h)$ and $\widehat{A}(2h)$ to create higher order estimates with much less error. *Adaptive methods* take a desired accuracy, e, and attempt (without knowing A) to find an h with $\left|\widehat{A}(h) - A\right| \leq e$. Error expansions like (3.4) are the basis of many adaptive methods.

This chapter focuses on truncation error and mostly ignores roundoff. In most practical computations that have truncation error, including numerical solution of differential equations or integral equations, the truncation error is much larger than roundoff. Referring to Figure 2.3, the practical range of h from .01 to 10^{-5}, the computed error is roughly $4.1 \cdot h$, as (3.4) suggests is should be. This starts to break down for the impractically small $h = 10^{-8}$. More sophisticated high order approximations reach roundoff sooner, which can be an issue in code testing, but rarely in large scale production runs.

3.1 Taylor series and asymptotic expansions

The Taylor series expansion of a function f about the point x is

$$f(x+h) = \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(x) h^n \ . \tag{3.5}$$

The notation $f^{(n)}(x)$ refers to the n^{th} derivative of f evaluated at x. The *partial sum* of order p is a degree p polynomial in h:

$$F_p(x,h) = \sum_{n=0}^{p} \frac{1}{n!} f^{(n)}(x) h^n \ . \tag{3.6}$$

The partial sum F_p is the Taylor approximation to $f(x+h)$ of order p. It is a polynomial of order p in the variable h. Increasing p makes the approximation more complicated and more accurate. The order $p = 0$ partial sum is simply $F_0(x,h) = f(x)$. The first and second order approximations are (3.1) and (3.2) respectively.

The Taylor series sum *converges* if the partial sums converge to f:

$$\lim_{p \to \infty} F_p(x,h) = f(x+h) \ .$$

If there is a positive h_0 so that the series converges whenever $|h| < h_0$, then f is *analytic* at x. A function probably is analytic at most points if there is a formula for it, or it is the solution of a differential equation. Figure 3.1 plots a function $f(x) = xe^{2x}$ together with the Taylor approximations of order zero, one, and two. The symbols at the ends of the curves illustrate the convergence of this Taylor series when $h = \pm.3$. When $h = .3$, the series converges monotonically: $F_0 < F_1 < F_2 < \cdots \to f(x+h)$. When $h = -.3$, there are approximants on both sides of the answer: $F_0 > F_2 > f(x+h) > F_1$.

It often is more useful to view the Taylor series as an *asymptotic expansion* of $f(x+h)$ valid as $h \to 0$. We explain what this means by giving an analogy between *order of magnitude* (and powers of ten) and *order of approximation* (powers of h). If B_1 and B_2 are positive numbers, then B_2 is an order of magnitude smaller than B_1 roughly if $B_2 \leq .1 \cdot B_1$. If $E_1(h)$ and $E_2(h)$ are functions of h defined for $|h| \leq h_0$, and if there is a C so that $|E_2(h)| \leq C \cdot h \cdot |E_1(h)|$, then we say that E_2 is an order (or an order of approximation) smaller than E_1 as $h \to 0$. This implies that for small enough h ($|h| < 1/C$) $E_2(h)$ is smaller than $E_1(h)$. Reducing h further makes E_2 much smaller than E_1. It is common to know that there is such a C without knowing what it is. Then we do not know how small h has to be before the asymptotic relation $E_2 \ll E_1$ starts to hold. Figure 3.2 and Figure 3.3 show that asymptotic relations can hold for practical values of h.

An asymptotic expansion actually is a sequence of approximations, like the $F_p(x,h)$, with increasing order of accuracy. One can view decimal expansions in this way, using order of magnitude instead of order of approximation. The expansion

$$\pi = 3.141592\cdots = 3 + 1 \cdot .1 + 4 \cdot (.1)^2 + 1 \cdot (.1)^3 + 5 \cdot (.1)^4 + \cdots \tag{3.7}$$

is a sequence of approximations

$$\begin{array}{lcl} \widehat{A}_0 & \approx & 3 \\ \widehat{A}_1 & \approx & 3 + 1 \cdot .1 \\ \widehat{A}_2 & \approx & 3 + 1 \cdot .1 + 4 \cdot (.1)^2 \\ & & \text{etc.} \end{array}$$

The approximation $\widehat{A}_p$ is an order of magnitude more accurate than $\widehat{A}_{p-1}$. The error $\widehat{A}_p - \pi$ is of the order of magnitude $(.1)^{p+1}$, which also is the order of magnitude of the first neglected term. The error in $\widehat{A}_3 = 3.141$ is $\widehat{A}_3 - \pi \approx 6 \cdot 10^{-4}$. This error is approximately the same as the next term, $5 \cdot (.1)^4$. Adding the next term gives an approximation whose error is an order of magnitude smaller:

$$\widehat{A}_3 + 5 \cdot (.1)^4 - \pi = \widehat{A}_4 - \pi \approx -9 \cdot 10^{-5} \ .$$

The *big O* (O for *order*) notation expresses asymptotic size relations. If $B_1(h) > 0$ for $h \neq 0$, the notation, $O(B_1(h))$ as $h \to 0$ refers to any function that is less than $C \cdot B_1(h)$ when $|h| \leq h_0$ (for some C). Thus, $B_2(h) = O(B_1(h))$ as $h \to 0$ if there is a C and an $h_0 > 0$ so that $B_1(h)$ and $B_2(h)$ are defined and and $B_2(h) \leq C \cdot B_1(h)$ for $|h| \leq h_0$. We also write $F(h) = G(h) + O(B_1(h))$ to mean that the function $B_2(h) = F(h) - G(h)$ satisfies $B_2(h) = O(B_1(h))$. For example, it is correct to write $\tan(h) = O(|h|)$ as $h \to 0$ because if $|h| \leq \pi/4$, $\tan(h) \leq 2|h|$. In this case the h_0 is necessary because $\tan(h) \to \infty$ as $h \to \pi/2$ (no C could work for $h = \pi/2$). Also, $C = 1$ does not work, even though $\tan(h) \approx h$ for small h, because $\tan(h) > h$.

There are two common misuses of the big O notation, and we indulge in both below. One is to forget that h could be negative and write $O(h)$ for $O(|h|)$. The other is to say $B = O(h^p)$ to mean that B and h^p are of the same order, that is both $B(h) = O(h^p)$ and $h^p = O(B(h))$. Technically, it is correct to say that $h^3 = O(h^2)$. But this can be misleading, as h^3 actually is an order smaller than h^2. It would be like saying you ran "less than ten miles" when you actually had run less than one mile, technically true but misleading.

We change notation when we view the Taylor series as an asymptotic expansion, writing

$$f(x+h) \sim f(x) + f'(x) \cdot h + \frac{1}{2} f''(x) h^2 + \cdots \; . \tag{3.8}$$

This means that the right side is an *asymptotic series* that may or may not converge. It represents $f(x+h)$ in the sense that the partial sums $F_p(x,h)$ are a family of approximations of increasing order of accuracy:

$$|F_p(x,h) - f(x+h)| = O(h^{p+1}) \; . \tag{3.9}$$

The asymptotic expansion (3.8) is much like the decimal expansion (3.7). The term in (3.8)of order $O(h^2)$ is $\frac{1}{2} f''(x) \cdot h^2$. The term in (3.7) of order of magnitude 10^{-2} is $4 \cdot (.1)^{-2}$. The error in a p term approximation is roughly the first neglected term, since all other neglected terms are at least one order smaller.

Figure 3.1 illustrates the asymptotic nature of the Taylor approximations. The lowest order approximation is $F_0(x,h) = f(x)$. The graph of F_0 touches the graph of f when $h = 0$ but otherwise has little in common. The graph of $F_1(x,h) = f(x) + f'(x)h$ not only touches the graph of f when $h = 0$, but the curves are tangent. The graph of $F_2(x,h)$ not only is tangent, but has the same curvature and is a better fit for small and not so small h.

3.1.1 Technical points

This subsection presents two technical points for mathematically minded readers. The first proves the basic fact that underlies most of the analysis in this chapter, that the Taylor series (3.5) is an asymptotic expansion. The second is two examples of asymptotic expansions that converge to the wrong answer or do not converge at all.

The asymptotic expansion property of Taylor series comes from the Taylor series *remainder theorem.*[3] If the derivatives of f up to order $p+1$ exist and are continuous in the interval $[x, x+h]$, then there is a $\xi \in [x, x+h]$ so that

$$f(x+h) - F_p(x,h) = \frac{1}{(p+1)!} f^{(p+1)}(\xi) h^{p+1} \; . \tag{3.10}$$

[3]See any good calculus book for a derivation and proof.

If we take
$$C = \frac{1}{(p+1)!} \max_{y \in [x,x+h]} \left| f^{(p+1)}(y) \right| ,$$
then we find that
$$|F_p(x,h) - f(x+h)| \leq C \cdot h^{p+1} .$$
This is the proof of (3.9), which states that the Taylor series is an asymptotic expansion.

The approximation $F_p(x,h)$ includes terms in the sum (3.8) up to order and including order p. The first neglected term is the term of order $p+1$, which is $\frac{1}{(p+1)} f^{(p+1)}(x)$. This also is the difference $F_{p+1} - F_p$. It differs from the right side of (3.10) only in ξ being replaced by x. Since $\xi \in [x, x+h]$, this is a small change if h is small. Therefore, the error in the F_p is nearly equal to the first neglected term.

An asymptotic expansion can converge to the wrong answer or not converge at all. We give an example of each. These are based on the fact that exponentials *beat* polynomials in the sense that, for any n,
$$t^n e^{-t} \to 0 \quad \text{as } \ t \to \infty .$$
If we take $t = a/|x|$ (because x may be positive or negative), this implies that
$$\frac{1}{x^n} e^{-a/x} \to 0 \quad \text{as } \ x \to 0 . \tag{3.11}$$

Consider the function $f(x) = e^{-1/|x|}$. This function is continuous at $x = 0$ if we define $f(0) = 0$. The derivative at zero is (using (3.11))
$$f'(0) = \lim_{h \to 0} \frac{f(h) - f(0)}{h} = \lim_{h \to 0} \frac{e^{-1/|h|}}{h} = 0 .$$
When $x \neq 0$, we calculate $f'(x) = \pm \frac{1}{|x|^2} e^{-1/|x|}$. The first derivative is continuous at $x = 0$ because (3.11) implies that $f'(x) \to 0 = f'(0)$ as $x \to 0$. Continuing in his way, one can see that each of the higher derivatives vanishes at $x = 0$ and is continuous. Therefore $F_p(0,h) = 0$ for any p, as $f^{(n)}(0) = 0$ for all n. Thus clearly $F_p(0,h) \to 0$ as $p \to \infty$. Simply put, the Taylor series converges to zero for any h because all the terms are zero. But this is the wrong answer, since, $f(h) = e^{1/h} > 0$ if $h \neq 0$. The Taylor series, while asymptotic, converges to the wrong answer.

What goes wrong here is that although the derivatives $f^{(p)}$ happen to take the value zero when $x = 0$, they are very large for x close to zero. The remainder theorem (**??**trf*lo) implies that $F_p(x,h) \to f(x+h)$ as $p \to \infty$ if
$$M_p = \frac{h^p}{p!} \max_{x \leq \xi \leq x+h} \left| f^{(p)}(\xi) \right| \to 0 \quad \text{as } p \to \infty.$$
Taking $x = 0$ and any $h > 0$, function $f(x) = e^{-1/|x|}$ has $M_p \to \infty$ as $p \to \infty$.

Here is an example of an asymptotic Taylor series that does not converge at all. Consider

$$f(h) = \int_0^{1/2} e^{-x/h} \frac{1}{1-x} dx \; . \tag{3.12}$$

The integrand goes to zero *exponentially* as $h \to 0$ for any fixed x. This suggests[4] that most of the integral comes from values of x near zero and that we can approximate the integral by approximating the integrand near $x = 0$. Therefore, we write $1/(1-x) = 1 + x + x^2 + \cdots$, which converges for all x in the range of integration. Integrating separately gives

$$f(h) = \int_0^{1/2} e^{-x/h} dx + \int_0^{1/2} e^{-x/h} x dx + \int_0^{1/2} e^{-x/h} x^2 dx + \cdots \; .$$

We get a simple formula for the integral of the general term $e^{-x/h}x^n$ if we change the upper limit from $1/2$ to ∞. For any fixed n, changing the upper limit of integration makes an exponentially small change in the integral, see problem (6). Therefore the n^{th} term is (for any $p > 0$)

$$\begin{aligned} \int_0^{1/2} e^{-x/h} x^n dx &= \int_0^{\infty} e^{-x/h} x^n dx + O(h^p) \\ &= n! h^{n+1} + O(h^p) \; . \end{aligned}$$

Assembling these gives

$$f(h) \sim h + h^2 + 2h^3 + \cdots + (n-1)! \cdot h^n + \cdots \tag{3.13}$$

This is an asymptotic expansion because the partial sums are asymptotic approximations:

$$\left| h + h^2 + 2h^3 + \cdots + (p-1)! \cdot h^p - f(h) \right| = O(h^{p+1}) \; .$$

But the infinite sum does not converge; for any $h > 0$ we have $n! \cdot h^{n+1} \to \infty$ as $n \to \infty$.

In these examples, the higher order approximations have smaller ranges of validity. For (**??**), the three term approximation $f(h) \approx h + h^2 + 2h^3$ is reasonably accurate when $h = .3$ but the six term approximation is less accurate, and the ten term "approximation" is 4.06 for an answer less than .5. The ten term approximation is very accurate when $h = .01$ but the fifty term "approximation" is astronomical.

3.2 Numerical Differentiation

One basic numerical task is estimating the derivative of a function from given function values. Suppose we have a smooth function, $f(x)$, of a single variable,

[4] A more precise version of this intuitive argument is in exercise 6.

x. The problem is to combine several values of f to estimate f'. These *finite difference* approximations are useful in themselves, and because they underlie methods for solving differential equations of all kinds. Several common finite difference approximations are

$$\left.\begin{aligned}
f'(x) &\approx \frac{f(x+h)-f(x)}{h} && (a) \\
f'(x) &\approx \frac{f(x)-f(x-h)}{h} && (b) \\
f'(x) &\approx \frac{f(x+h)-f(x-h)}{2h} && (c) \\
f'(x) &\approx \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} && (d) \\
f'(x) &\approx \frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x+2h)}{12h} && (e)
\end{aligned}\right\} \tag{3.14}$$

The first three have simple geometric interpretations as the slope of lines connecting nearby points on the graph of $f(x)$. A carefully drawn figure shows that (3.14c) is more accurate than (3.14a). We give an analytical explanation of this below. The last two are more technical. The formulas (3.14a), (3.14b), and (3.14d) are *one sided* because they use values only on one side of x. The formulas (3.14c) and (3.14e) are *centered* because they use points symmetrical about x and with opposite weights.

The Taylor series expansion (3.8) allows us to calculate the accuracy of each of these approximations. Let us start with the simplest (3.14a). Substituting (3.8) into the right side of (3.14a) gives

$$\frac{f(x+h)-f(x)}{h} \sim f'(x) + h\frac{f''(x)}{2} + h^2\frac{f'''(x)}{6} + \cdots \ . \tag{3.15}$$

This may be written:

$$\frac{f(x+h)-f(x)}{h} = f'(x) + E_a(h) \ ,$$

where

$$E_a(h) \sim \frac{1}{2}f''(x)\cdot h + \frac{1}{6}f'''(x)\cdot h^2 + \cdots \ . \tag{3.16}$$

In particular, this shows that $E_a(h) = O(h)$, which means that the one sided two point finite difference approximation is first order accurate. Moreover,

$$E_a(h) = \frac{1}{2}f''(x)\cdot h + O(h^2) \ , \tag{3.17}$$

which is to say that, to *leading order*, the error is proportional to h and given by $\frac{1}{2}f''(x)$.

Taylor series analysis applied to the two point centered difference approximation (3.14c) leads to

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + E_c(h)$$

where

$$\begin{aligned} E_c(h) &\sim \frac{1}{6} f'''(x) \cdot h^2 + \frac{1}{24} f^{(5)}(x) \cdot h^4 + \cdots \\ &= \frac{1}{6} f'''(x) \cdot h^2 + O(h^4) \end{aligned} \tag{3.18}$$

This centered approximation is second order accurate, $E_c(h) = O(h^2)$. This is one order more accurate than the one sided approximations (3.14a) and (3.14b). Any centered approximation such as (3.14c) or (3.14e) must be at least second order accurate because of the symmetry relation $\widehat{A}(-h) = \widehat{A}(h)$. Since $A = f'(x)$ is independent of h, this implies that $E(h) = \widehat{A}(h) - A$ is symmetric. If $E(h) = c \cdot h + O(h^2)$, then

$$E(-h) = -c \cdot h + O(h^2) = E(h) + O(h^2) \approx -E(h) \quad \text{for small } h,$$

which contradicts $E(-h) = E(h)$. The same kind of reasoning shows that the $O(h^3)$ term in (3.18) must be zero.

A Taylor series analysis shows that the three point one sided formula (3.14d) is second order accurate, while the four point centered approximation (3.14e) is fourth order. Sections 3.3.1 and 3.5 give two ways to find the coefficients 4, -3, and 8 achieve these higher orders of accuracy.

Figure 3.2 illustrates many of these features. The first is that the higher order formulas (3.14c), (3.14d), and (3.14e) actually are more accurate when h is small. For $h = .5$, the first order two point one sided difference formula is more accurate than the second order accurate three point formula, but their proper asymptotic ordering is established by $h = .01$. For $h \leq 10^{-5}$ with the fourth order centered difference formula and $h = 10^{-7}$ with the second order formula, double precision roundoff error makes the results significantly different from what they would be in exact arithmetic. The rows labeled $\widehat{E}$ give the leading order Taylor series estimate of the error. For the first order formula, (3.17) shows that this is $\widehat{E}(h) = \frac{1}{2} f''(x) \cdot h$. For the second order centered formula, (3.18) gives leading order error $\widehat{E}_c(h) = \frac{1}{6} f'''(x) \cdot h^2$. For the three point one sided formula, the coefficient of $f'''(x) \cdot h^2$ is $\frac{1}{3}$, twice the coefficient for the second order centered formula. For the fourth order formula, the coefficient of $f^{(5)}(x) \cdot h^4$ is $\frac{1}{30}$. The table shows that $\widehat{E}$ is a good predictor of E, if h is at all small, until roundoff gets in the way. The smallest error[5] in the table comes from the fourth order formula and $h = 10^{-5}$. It is impossible to have an error

[5]The error would have been $-3 \cdot 10^{-19}$ rather than $-6 \cdot 10^{-12}$, seven orders of magnitude smaller, in exact arithmetic. The best answer comes despite some catastrophic cancellation, but not completely catastrophic.

h	(3.14a)	(3.14c)	(3.14d)	(3.14e)
.5	3.793849	0.339528	7.172794	0.543374
E	2.38e+00	-1.08e+00	5.75e+00	-8.75e-01
$\widehat{E}$	5.99e+00	-1.48e+00	-2.95e+00	-1.85e+00
.01	2.533839	1.359949	1.670135	1.415443
E	1.12e+00	-5.84e-02	2.52e-01	-2.87e-03
$\widehat{E}$	1.20e+00	-5.91e-02	-1.18e-01	-2.95e-03
$5 \cdot 10^{-3}$	1.999796	1.403583	1.465752	1.418128
E	5.81e-01	-1.47e-02	4.74e-02	-1.83e-04
$\widehat{E}$	5.99e-01	-1.48e-02	-2.95e-02	-1.85e-04
10^{-3}	1.537561	1.417720	1.419642	1.418311
E	1.19e-01	-5.91e-04	1.33e-03	-2.95e-07
$\widehat{E}$	1.20e-01	-5.91e-04	-1.18e-03	-2.95e-07
10^{-5}	1.418431	1.418311	1.418311	1.418311
E	1.20e-04	-5.95e-10	1.16e-09	-6.05e-12
$\widehat{E}$	1.20e-04	-5.91e-10	-1.18e-09	-2.95e-19
10^{-7}	1.418312	1.418311	1.418311	1.418311
E	1.20e-06	2.76e-10	3.61e-09	8.31e-10
$\widehat{E}$	1.20e-06	-5.91e-14	-1.18e-13	$-2.95 \cdot 10^{-27}$

Figure 3.2: Estimates of $f'(x)$ with $f(x) = \sin(5x)$ and $x = 1$ using formulas (3.14a), (3.14c), (3.14d), and (3.14e). Each group of three rows corresponds to one h value. The top row gives the finite difference estimate of$f'(x)$, the middle row gives the error $E(h)$, and the third row is $\widehat{E}(h)$, the leading Taylor series term in the error formula. All calculations were done in double precision floating point arithmetic.

this small with a first or second order formula no matter what the step size. Note that the error in the (3.14e) column increased when h was reduced from 10^{-5} to 10^{-7} because of roundoff.

A difference approximation may not achieve its expected order of accuracy if the requisite derivatives are infinite or do not exist. As an example of this, let $f(x)$ be the function

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x^2 & \text{if } x \geq 0 \end{cases} .$$

If we want $f'(0)$, the formulas (1c) and (1e) are only first order accurate despite their higher accuracy for smoother functions. This f has a mild singularity, a discontinuity in its second derivative. Such a singularity is hard to spot on a graph, but may have a drastic effect on the numerical analysis of the function.

We can use finite differences to approximate higher derivatives such as

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + \frac{h^2}{12} f^{(4)} + O(h^4) \ ,$$

and to estimate partial derivatives of functions depending on several variables, such as

$$\frac{f(x+h,y)-f(x-h,y)}{2h} \sim \frac{\partial}{\partial x} f(x,y) + \frac{h^2}{3} \frac{\partial^3 f}{\partial x^3}(x,y) + \cdots \quad .$$

3.2.1 Mixed partial derivatives

There are several new features that arise only when evaluating mixed partial derivatives or sums of partial derivatives in different variables. For example, suppose we want to evaluate[6] $f_{xy} = \partial_x \partial_y f(x,y)$. Rather than using the same h for both[7] x and y, we use step size Δx for x and Δy for y. The first order one sided approximation for f_y is

$$f_y \approx \frac{f(x, y+\Delta y) - f}{\Delta y} \quad .$$

We might hope this, and

$$f_y(x+\Delta x, y) \approx \frac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y)}{\Delta y} \quad ,$$

are accurate enough so that

$$\begin{aligned}
\partial_x\big(\partial_y f\big) &\approx \frac{f_y(x+\Delta x, y) - f_y}{\Delta x} \\
&\approx \frac{\dfrac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y)}{\Delta y} - \dfrac{f(x, y+\Delta y) - f}{\Delta y}}{\Delta x} \\
f_{xy} &\approx \frac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y) - f(x, y+\Delta y) + f}{\Delta x \Delta y} \qquad (3.19)
\end{aligned}$$

is consistent[8].

To understand the error in (3.19), we need the Taylor series for functions of more than one variable. The rigorous remainder theorem is more complicated, but it suffices here to use all of the "first" neglected terms. The expansion is

$$\begin{aligned}
f(x+\Delta x, y+\Delta y) &\sim f + \Delta x f_x + \Delta y f_y \\
&\quad + \frac{1}{2}\Delta x^2 f_{xx} + \Delta x \Delta y f_{xy} + \frac{1}{2}\Delta y^2 f_{yy}
\end{aligned}$$

---

[6] We abbreviate formulas by denoting partial derivatives by subscripts, $\partial_x f = f_x$, etc., and by leaving out the arguments if they are (x,y), so $f(x+\Delta x, y) - f(x,y) = f(x+\Delta x, y) - f \approx \Delta x f_x(x,y) = \Delta x f_x$.

[7] The expression $f(x+h, y+h)$ does not even make sense if x and y have different physical units.

[8] The same calculation shows that the right side of (3.19) is an approximation of $\partial_y\left(\partial_x f\right)$. This is one proof that $\partial_y \partial_x f = \partial_y \partial_x f$.

$$
\begin{aligned}
&+\frac{1}{6}\Delta x^3 f_{xxx} + \frac{1}{2}\Delta x^2 \Delta y f_{xxy} + \frac{1}{2}\Delta x \Delta y^2 f_{xyy} + \frac{1}{6}\Delta y^3 f_{yyy} \\
&+ \ \cdots \\
&+\frac{1}{p!}\sum_{k=0}^{p}\binom{p}{k}\Delta x^{n-k}\Delta y^k \partial_x^{p-k}\partial_y^k f + \cdots \ .
\end{aligned}
$$

If we keep just the terms on the top row on the right, the second order terms on the second row are the first neglected terms, and (using the inequality $\Delta x \Delta y \leq \Delta x^2 + \Delta y^2$):

$$
f(x+\Delta x, y+\Delta y) = f + \Delta x f_x + \Delta y f_y + O\left(\Delta x^2 + \Delta y^2\right) \ .
$$

Similarly,

$$
\begin{aligned}
&f(x+\Delta x, y+\Delta y) \\
&\quad = f + \Delta x f_x + \Delta y f_y + \frac{1}{2}\Delta x^2 f_{xx} + \Delta x \Delta y f_{xy} + \frac{1}{2}\Delta y^2 f_{yy} \\
&\quad + O\left(\Delta x^3 + \Delta y^3\right) \ .
\end{aligned}
$$

Of course, the one variable Taylor series is

$$
f(x+\Delta x, y) = f + \Delta x f_x + \frac{1}{2}\Delta x^2 f_{xx} + O\left(\Delta x^3\right) \ , \text{ etc.}
$$

Using all these, and some algebra, gives

$$
\begin{aligned}
&\frac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y) - f(x, y+\Delta y) + f}{\Delta x \Delta y} \\
&\qquad\qquad\qquad = f_{xy} + O\left(\frac{\Delta x^3 + \Delta y^3}{\Delta x \Delta y}\right) \ . \qquad (3.20)
\end{aligned}
$$

This shows that the approximation (3.19) is first order, at least if Δx is roughly proportional to Δy. The fuller Taylor expansion above gives a quantitative estimate of the error:

$$
\begin{aligned}
&\frac{f(x+\Delta x, y+\Delta y) - f(x+\Delta x, y) - f(x, y+\Delta y) + f}{\Delta x \Delta y} - f_{xy} \\
&\qquad\qquad\qquad \approx \frac{1}{2}\left(\Delta x f_{xxy} + \Delta y f_{xyy}\right) \ . \qquad (3.21)
\end{aligned}
$$

This formula suggests (and it is true) that in exact arithmetic we could let $\Delta x \to 0$, with Δy fixed but small, and still have a reasonable approximation to f_{xy}. The less detailed version (3.20) suggests that might no be so.

A partial differential equation may involve a *differential operator* that is a sum of partial derivatives. One way to approximate a differential operator is to approximate each of the terms separately. for example, the *Laplace* operator (or *Laplacian*), which is $\triangle = \partial_x^2 + \partial_y^2$ in two dimensions, may be approximated by

$$\begin{aligned} \triangle f(x,y) &= \partial_x^2 f + \partial_y^2 f \\ &\approx \frac{f(x+\Delta x,y) - 2f + f(x-\Delta x,y)}{\Delta x^2} \\ &\quad + \frac{f(x,y+\Delta y) - 2f + f(x,y-2\Delta y)}{\Delta y^2} \ . \end{aligned}$$

If $\Delta x = \Delta y = h$ (x and y have the same units in the Laplace operator), then this becomes

$$\triangle f \approx \frac{1}{h^2}\big(f(x+h,y) + f(x-h,y) + f(x,y+h) + f(x,y-h) - 4f\big) \ . \tag{3.22}$$

This is the *standard* five point approximation (seven points in three dimensions). The leading error term is

$$\frac{h^2}{12}\big(\partial_x^4 f + \partial_y^4 f\big) \ . \tag{3.23}$$

The simplest *heat equation* (or *diffusion* equation) is $\partial_t f = \frac{1}{2}\partial_x^2 f$. The *space* variable, x, and the *time* variable, t have different units. We approximate the differential operator using a first order forward difference approximation in time and a second order centered approximation in space. This gives

$$\partial_t f - \frac{1}{2}\partial_x^2 f \approx \frac{f(x,t+\Delta t) - f}{\Delta t} - \frac{f(x+\Delta x,t) - 2f + f(x-\Delta x,t)}{2\Delta x^2} \ . \tag{3.24}$$

The leading order error is the sum of the leading errors from time differencing ($\frac{1}{2}\Delta t \partial_t^2 f$) and space differencing ($\frac{\Delta x^2}{24}\partial_x^4 f$), which is

$$\frac{1}{2}\Delta t \partial_t^2 f - \frac{\Delta x^2}{24}\partial_x^4 f \ . \tag{3.25}$$

For many reasons, people often take Δt proportional to Δx^2. In the simplest case of $\Delta t = \Delta x^2$, the leading error becomes

$$\Delta x^2 \left(\frac{1}{2}\partial_t^2 f - \frac{1}{24}\partial_x^4 f\right) \ .$$

This shows that the overall approximation (3.24) is second order accurate if we take the time step to be the square of the space step.

3.3 Error Expansions and Richardson Extrapolation

The error expansions (3.16) and (3.18) above are instances of a common situation that we now describe more systematically and abstractly. We are trying to compute A and there is an approximation with

$$\widehat{A}(h) \to A \ \text{ as } \ h \to 0 \ .$$

The error is $E(h) = \widehat{A}(h) - A$. A general *asymptotic error expansion* in powers of h has the form

$$\widehat{A}(h) \sim A + h^{p_1} A_1 + h^{p_2} A_2 + \cdots \quad , \tag{3.26}$$

or, equivalently,

$$E(h) \sim h^{p_1} A_1 + h^{p_2} A_2 + \cdots \quad .$$

As with Taylor series, the expression (3.26) does not imply that the series on the right converges to $\widehat{A}(h)$. Instead, the asymptotic relation (3.26) means that, as $h \to 0$,

$$\left.\begin{array}{ll} \widehat{A}(h) - (A + h^{p_1} A_1) = O(h^{p_2}) & (a) \\[2ex] \widehat{A}(h) - (A + h^{p_1} A_1 + h^{p_2} A_2) = O(h^{p_3}) & (b) \\[2ex] \textit{and so on.} & \end{array}\right\} \tag{3.27}$$

It goes without saying that $0 < p_1 < p_2 < \cdots$. The statement (3.27a) says not only that $A + A_1 h^{p_1}$ is a good approximation to $\widehat{A}(h)$, but that the error has the same order as the first neglected term, $A_2 h^{p_2}$. The statement (3.27b) says that including the $O(h^{p_2})$ term improves the approximation to $O(h^{p_3})$, and so on.

Many asymptotic error expansions arise from Taylor series manipulations. For example, the two point one sided difference formula error expansion (3.15) gives $p_1 = 1$, $A_1 = \frac{1}{2} f''(x)$, $p_2 = 2$, $A_2 = \frac{1}{6} f'''(x)$, etc. The error expansion (3.18) for the two point centered difference formula implies that $p_1 = 2$, $p_2 = 4$, $A_1 = \frac{1}{6} f'''(x)$, and $A_2 = \frac{1}{24} f^{(5)}(x)$. The three point one sided formula has $p_1 = 2$ because it is second order accurate, but $p_2 = 3$ instead of $p_2 = 4$. The fourth order formula has $p_1 = 4$ and $p_2 = 6$.

It is possible that an approximation is p^{th} order accurate in the big O sense, $|E(h)| \leq C \cdot h^p$, without having an asymptotic error expansion of the form (3.26). Figure 3.4 has an example showing that this can happen when the function $f(x)$ is not sufficiently smooth. Most of the extrapolation and debugging tricks described here do not apply in those cases.

We often work with asymptotic error expansions for which we know the powers p_k but not the coefficients, A_k. For example, in finite difference approximations, the A_k depend on the function f but the p_k do not. Two computational techniques that use this information are are *Richardson extrapolation* and *convergence analysis*. Richardson extrapolation combines $\widehat{A}(h)$ approximations for several values of h to produce a new approximation that has greater order of accuracy than $\widehat{A}(h)$. Convergence analysis is a debugging method that tests the order of accuracy of numbers produced by a computer code.

3.3.1 Richardson extrapolation

Richardson extrapolation increases the order of accuracy of an approximation provided that the approximation has an asymptotic error expansion of the form

(3.26) with known p_k. In its simplest form, we compute $\widehat{A}(h)$ and $\widehat{A}(2h)$ and then form a linear combination that eliminates the leading error term. Note that

$$\begin{aligned}\widehat{A}(2h) &= A + (2h)^{p_1} A_1 + (2h)^{p_2} A_2 + \cdots \\ &= A + 2^{p_1} h^{p_1} A_1 + 2^{p_2} h^{p_2} A_2 + \cdots \quad ,\end{aligned}$$

so

$$\frac{2^{p_1}\widehat{A}(h) - \widehat{A}(2h)}{2^{p_1}-1} = A + \frac{2^{p_1}-2^{p_2}}{2^{p_1}-1} h^{p_2} A_2 + \frac{2^{p_3}-2^{p_2}}{2^{p_1}-1} h^{p_3} A_3 + \cdots \quad .$$

In other words, the *extrapolated* approximation

$$\widehat{A}^{(1)}(h) = \frac{2^{p_1}\widehat{A}(h) - \widehat{A}(2h)}{2^{p_1}-1} \tag{3.28}$$

has order of accuracy $p_2 > p_1$. It also has an asymptotic error expansion,

$$\widehat{A}^{(1)}(h) = A + h^{p_2} A_2^{(1)} + h^{p_3} A_3^{(1)} + \cdots \quad ,$$

where $A_2^{(1)} = \dfrac{2^{p_1}-2^{p_2}}{2^{p_1}-1} A_2$, and so on.

Richardson extrapolation can be repeated to remove more asymptotic error terms. For example,

$$\widehat{A}^{(2)}(h) = \frac{2^{p_2}\widehat{A}^{(1)}(h) - \widehat{A}^{(1)}(2h)}{2^{p_2}-1}$$

has order p_3. Since $\widehat{A}^{(1)}(h)$ depends on $\widehat{A}(h)$ and $\widehat{A}(2h)$, $\widehat{A}^{(2)}(h)$ depends on $\widehat{A}(h)$, $\widehat{A}(2h)$, and $\widehat{A}(4h)$. It is not necessary to use powers of 2, but this is natural in many applications. Richardson extrapolation will not work if the underlying approximation, $\widehat{A}(h)$, has accuracy of order h^p in the $O(h^p)$ sense without at least one term of an asymptotic expansion.

Richardson extrapolation, allows us to derive higher order difference approximations from low order ones. Start, for example, with the first order one sided approximation to $f'(x)$ given by (3.14a). Taking $p_1 = 1$ in (3.28) leads to the second order approximation

$$\begin{aligned}f'(x) &\approx 2 \cdot \frac{f(x+h)-f(x)}{h} - \frac{f(x+2h)-f(x)}{2h} \\ &= \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} \quad ,\end{aligned}$$

which is the second order three point one sided difference approximation (3.14d). Starting with the second order centered approximation (3.14c) (with $p_1 = 2$ and $p_2 = 4$) leads to the fourth order approximation (3.14e). The second order one sided formula has $p_1 = 2$ and $p_2 = 3$. Applying Richardson extrapolation to it gives a one sided formula that uses $f(x+4h)$, $f(x+2h)$, $f(x+h)$, and $f(x)$ to give a third order approximation. A better third order one sided approximation would use $f(x+3h)$ instead of $f(x+4h)$. Section 3.5 explains how to do this.

Richardson extrapolation may also be applied to the output of a complex code. Run it with step size h and $2h$ and apply (3.28) to the output. This is sometimes applied to stochastic differential equations as an alternative to making up high order schemes from scratch, which can be time consuming and intricate.

3.3.2 Convergence analysis

We can test a code, and the algorithm it is based on, using ideas related to Richardson extrapolation. A naive test would be to do runs with decreasing h values to check whether $\widehat{A}(h) \to A$ as $h \to 0$. A *convergence analysis* based on asymptotic error expansions can be better. For one thing, we might not know A. Even if we run a test case where A is known, it is common that a code with mistakes limps to convergence, but not as accurately or reliably as the correct code would. If we bother to write a code that is more than first order accurate, we should test that we are getting the order of accuracy we worked for.

There are two cases, the case where A is known and the case where A is not known. While we probably would not write a code for a problem to which we know the answer, it is often possible to apply a code to a problem with a known answer for debugging. In fact, a code should be written modularly so that it is easy to apply it to a range or problems broad enough to include some trivial and at least one nontrivial problem[9] with a known answer.

If A is known, we can run the code with step size h and $2h$ and, from the resulting approximations, $\widehat{A}(h)$ and $\widehat{A}(2h)$, compute

$$\begin{aligned} E(h) &\approx A_1 h^{p_1} + A_2 h^{p_2} + \cdots \ , \\ E(2h) &\approx 2^{p_1} A_1 h^{p_1} + 2^{p_2} A_2 h^{p_2} + \cdots \ . \end{aligned}$$

For small h the first term is a good enough approximation so that the ratio should be approximately the characteristic value

$$R(h) = \frac{E(2h)}{E(h)} \approx 2^{p_1} \ . \tag{3.29}$$

Figure 3.3 is a computational illustration of this phenomenon. As $h \to 0$, the ratios converge to the expected result $2^{p_1} = 2^3 = 8$. Figure 3.4 shows what may happen when we apply this convergence analysis to an approximation that is second order accurate in the big O sense without having an asymptotic error expansion. The error gets very small but the error ratio does not have simple behavior as in Figure 3.3.

[9] A trivial problem is one that is too simple to test the code fully. For example, if you compute the derivative of a linear function, any of the formulae (3.14a) – (3.14e) would give the exact answer. The fourth order approximation (3.14e) gives the exact answer for any polynomial of degree less than five.

h	Error: $E(h)$	Ratio: $E(h)/E(h/2)$
.1	4.8756e-04	3.7339e+00
.05	1.3058e-04	6.4103e+00
.025	2.0370e-05	7.3018e+00
.0125	2.7898e-06	7.6717e+00
6.2500e-03	3.6364e-07	7.8407e+00
3.1250e-03	4.6379e-08	7.9215e+00
1.5625e-03	5.8547e-09	7.9611e+00
7.8125e-04	7.3542e-10	——— -

Figure 3.3: Convergence study for a third order accurate approximation. As $h \to 0$, the ratio converges to $2^3 = 8$. The h values in the left column decrease by a factor of two from row to row.

h	Error: $E(h)$	Ratio: $E(h)/E(h/2)$
.1	1.9041e-02	2.4014e+00
.05	7.9289e-03	1.4958e+01
.025	5.3008e-04	-1.5112e+00
.0125	-3.5075e-04	3.0145e+00
6.2500e-03	-1.1635e-04	1.9880e+01
3.1250e-03	-5.8529e-06	-8.9173e-01
1.5625e-03	6.5635e-06	2.8250e+00
7.8125e-04	2.3233e-06	——— -

Figure 3.4: Convergence study for an approximation that is second order accurate in the sense that $|E(h)| = O(h^2)$ but that has no asymptotic error expansion. The h values are the same as in Figure 3.3. The errors decrease in an irregular fashion.

Convergence analysis can be applied even when A is not known. In this case we need three approximations, $\widehat{A}(4h)$, $\widehat{A}(2h)$, and $\widehat{A}(h)$. Again assuming the existence of an asymptotic error expansion (3.26), we get, for small h,

$$R'(h) = \frac{\widehat{A}(4h) - \widehat{A}(2h)}{\widehat{A}(2h) - \widehat{A}(h)} \approx 2^{p_1} \quad . \tag{3.30}$$

3.4 Integration

Numerical integration means finding approximations for quantities such as

$$I = \int_a^b f(x)dx \quad .$$

Rectangle	$\widehat{I}_k = h_k f(x_k)$	1^{st} order
Trapezoid	$\widehat{I}_k = \dfrac{h_k}{2}\left(f(x_k) + f(x_{k+1})\right)$	2^{nd} order
Midpoint	$\widehat{I}_k = h_k f(x_{k+1/2})$	2^{nd} order
Simpson	$\widehat{I}_k \approx \dfrac{h_k}{6}\big(f(x_k) + 4f(x_{k+1/2}) + f(x_{k+1})\big)$	4^{th} order
2 point GQ	$\widehat{I}_k = \dfrac{h_k}{2}\big(f(x_{k+1/2} - h_k\xi) + f(x_{k+1/2} + h_k\xi)\big)$	4^{th} order
3 point GQ	$\widehat{I}_k = \dfrac{h_k}{18}\big(5f(x_{k+1/2} - h_k\eta) + 8f(x_{k+1/2}) + 5f(x_{k+1/2} + h_k\eta)\big)$	6^{th} order

Figure 3.5: Common panel integration rules. The last two are Gauss quadrature (Gauss – Legendre to be precise) formulas. The definitions are $\xi = \frac{1}{2\sqrt{3}}$ and $\eta = \frac{1}{2}\sqrt{\frac{3}{5}}$.

We discuss only *panel methods* here, though there are other elegant methods. In a panel method, the integration interval, $[a, b]$, is divided into n subintervals, or *panels*, $P_k = [x_k, x_{k+1}]$, where $a = x_0 < x_1 < \cdots < x_n = b$. If the panel P_k is small, we can get an accurate approximation to

$$I_k = \int_{P_k} f(x)dx = \int_{x_k}^{x_{k+1}} f(x)dx \tag{3.31}$$

using a few evaluations of f inside P_k. Adding these approximations gives an approximation to I:

$$\widehat{I} = \sum_{k=0}^{n-1} \widehat{I}_k \; . \tag{3.32}$$

Some of the more common panel integral approximations are given in Figure 3.5, where we write $x_{k+1/2} = (x_{k+1} + x_k)/2$ for the midpoint of the panel and $h_k = x_{k+1} - x_k$ is the width. Note that x_k is the left endpoint of P_k and the right endpoint of P_{k-1}. In the trapezoid rule and Simpon's rule, we need not evaluate $f(x_k)$ twice.

For our error analysis, we assume that all the panels are the same size

$$h = \Delta x = |P_k| = x_{k+1} - x_k \;\text{ for all } k.$$

Given this restriction, not every value of h is allowed because $b - a = nh$ and n is an integer. When we take $h \to 0$, we will assume that h only takes allowed values $h = (b - a)/n$. The *local truncation error* is the integration error over one panel. The overall *global* error is the sum of the local truncation errors in all the panels. The global error usually is one power of h larger than the local truncation error. If the error per panel is $O(h^q)$, then the total error will be of the order of h^q multiplied by n, the number of panels. Since $n = (b - a)/h$, this suggests that the global error will be of order $h^q \cdot (b - a)/h = O(h^{q-1})$.

For the local truncation error analysis, let $P = [x_*, x_* + h]$ be a generic panel. The panel integration rule approximates the panel integral

$$I_P = \int_P f(x)dx = \int_{x_*}^{x_*+h} f(x)dx$$

with the approximation, $\widehat{I}_P$. For example, the rectangle rule (top row of Figure 3.5) has panel integration rule

$$\int_{x_*}^{x_*+h} f(x)dx \approx \widehat{I}_P(h) = hf(x_*) \quad .$$

To estimate the difference between I_P and $\widehat{I}_P(h)$, we expand f in a Taylor series about x_*:

$$f(x) \sim f(x_*) + f'(x_*)(x - x_*) + \frac{1}{2}f''(x_*)(x - x_*)^2 + \cdots \quad .$$

Integrating this term by term leads to

$$\begin{aligned} I_P &\sim \int_P f(x_*)dx + \int_P f'(x_*)(x - x_*)dx + \cdots \\ &= f(x_*)h + \frac{1}{2}f'(x_*)h^2 + \frac{1}{6}f''(x_*)h^3 + \cdots \quad . \end{aligned}$$

The error in integration over this panel then is

$$E(P, h) = \widehat{I}_P(h) - I_P \sim -\frac{1}{2}f'(x_*)h^2 - \frac{1}{6}f''(x_*)h^3 - \cdots \quad . \tag{3.33}$$

This shows that the local truncation error for the rectangle rule is $O(h^2)$ and identifies the leading error coefficient.

$$\begin{aligned} E &= \widehat{I} - I \\ &= \sum_{n=0}^{n-1} \widehat{I}_k - I_k \\ E &\sim -\sum_{k=0}^{n-1} \frac{1}{2}f'(x_k)h^2 - \sum_{k=0}^{n-1} \frac{1}{6}f''(x_k)h^3 - \cdots \quad . \end{aligned} \tag{3.34}$$

We sum over k and use simple inequalities to get the order of magnitude of the global error:

$$\begin{aligned} |E| &<\approx \frac{1}{2}\sum_{k=0}^{n-1} |f'(x_k)| \cdot h^2 \\ &\leq n \cdot \frac{1}{2} \max_{a \leq x \leq b} |f'(x)| \cdot h^2 \\ &= \frac{b-a}{h} O(h^2) \\ &= O(h) \ . \end{aligned}$$

This shows that the rectangle rule is first order accurate overall.

Looking at the global error in more detail leads to an asymptotic error expansion. Applying the rectangle rule error bound to another function, $g(x)$, we have

$$\sum_{k=0}^{n-1} g(x_k)h = \int_a^b g(x)dx + O(h) \quad .$$

Taking $g(x) = f'(x)$ gives

$$\sum_{k=0}^{n-1} f'(x_k)h = \int_a^b f'(x)dx + O(h) = f(b) - f(a) + O(h) \quad .$$

From (3.34) we have

$$\begin{aligned}
E &\approx -\left(\sum_{k=0}^{n-1} f'(x_k)h\right)\frac{h}{2} \\
&\approx -\left(\int_a^b f'(x)dx\right)\frac{h}{2} \\
E &\approx -\frac{1}{2}\left(f(b) - f(a)\right)h \; . \qquad (3.35)
\end{aligned}$$

This gives the first term in the asymptotic error expansion. It shows that the leading error not only is bounded by h, but roughly is proportional to h. It also demonstrates the curious fact that if f is differentiable then the leading error term is determined by the values of f at the endpoints and is independent of the values of f between. This is not true if f has a discontinuity in the interval $[a, b]$.

To get the next term, apply (3.34) to the error itself, i.e.

$$\begin{aligned}
\sum_{k=0}^{n-1} f'(x_k)h &= \int_a^b f'(x)dx - \frac{h}{2}\left(f'(b) - f'(a)\right) + O(h^2) \\
&= f(b) - f(a) - \frac{h}{2}\left(f'(b) - f'(a)\right) + O(h^2) \quad .
\end{aligned}$$

In the same way, we find that

$$\sum_{k=0}^{n-1} f''(x_k)\frac{h^3}{6} = \left(f'(b) - f'(a)\right)\frac{h^2}{6} + O(h^3) \quad .$$

Combining all these gives the first two terms in the error expansion:

$$E(h) \sim -\frac{1}{2}\left(f(b) - f(a)\right)h + \frac{1}{12}\left(f'(b) - f'(a)\right)h^2 + \cdots \quad . \qquad (3.36)$$

It is clear that this procedure can be used to continue the expansion as far as we want, but you would have to be very determined to compute, for example,

n	Computed Integral	Error	Error/h	$(E - A_1 h)/h^2$	$(E - A_1 h - A_2 h^2)/h^3$
10	3.2271	-0.2546	-1.6973	0.2900	-0.7250
20	3.3528	-0.1289	-1.7191	0.2901	-0.3626
40	3.4168	-0.0649	-1.7300	0.2901	-0.1813
80	3.4492	-0.0325	-1.7354	0.2901	-0.0907
160	3.4654	-0.0163	-1.7381	0.2901	-0.0453

Figure 3.6: Computational experiment illustrating the asymptotic error expansion for rectangle rule integration.

n	Computed Integral	Error	Error/h	$(E - A_1 h)/h^2$
10	7.4398e-02	-3.1277e-02	-3.1277e-01	-4.2173e-01
20	9.1097e-02	-1.4578e-02	-2.9156e-01	-4.1926e-01
40	9.8844e-02	-6.8314e-03	-2.7326e-01	-1.0635e-01
80	1.0241e-01	-3.2605e-03	-2.6084e-01	7.8070e-01
160	1.0393e-01	-1.7446e-03	-2.7914e-01	-1.3670e+00
320	1.0482e-01	-8.5085e-04	-2.7227e-01	-5.3609e-01
640	1.0526e-01	-4.1805e-04	-2.6755e-01	1.9508e+00
1280	1.0546e-01	-2.1442e-04	-2.7446e-01	-4.9470e+00
2560	1.0557e-01	-1.0631e-04	-2.7214e-01	-3.9497e+00
5120	1.0562e-01	-5.2795e-05	-2.7031e-01	1.4700e+00

Figure 3.7: Computational experiment illustrating the breakdown of the asymptotic expansion for a function with a continuous first derivative but discontinuous second derivative.

the coefficient of h^4. An elegant and more systematic discussion of this error expansion is carried out in the book of Dahlquist and Bjork. The resulting error expansion is called the *Euler McLaurin* formula. The coefficients 1/2, 1/12, and so on, are related to the *Bernoulli numbers*.

The error expansion (3.36) will not be valid if the integrand, f, has singularities inside the domain of integration. Suppose, for example, that $a = 0$, $b = 1$, $u = 1/\sqrt{2}$, and $f(x) = 0$ for $x \leq u$ and $f(x) = \sqrt{x - u}$ for $x \geq u$. In this case the error expansion for the rectangle rule approximation to $\int_0^1 f(x)dx$ has one valid term only. This is illustrated in Figure 3.7. The "Error/h" column shows that the first coefficient, A_1, exists. Moreover, A_1 is given by the formula (3.36). The numbers in the last column do not tend to a limit. This shows that the coefficient A_2 does not exist. The error expansion does not exist beyond the first term.

The analysis of the higher order integration methods listed in Figure 3.5 is easier if we use a symmetric basic panel. From now on, the panel of length h will have x_* in the center, rather at the left end, that is

$$P = [x_* - h/2, x_* + h/2] \quad .$$

If we now expand $f(x)$ in a Taylor series about x_* and integrate term by term, we get

$$\int_P f(x)dx = \int_{x=x_*-\frac{h}{2}}^{x_*+\frac{h}{2}} f(x)dx \sim f(x_*)h + \frac{f''(x_*)}{24}h^3 + \frac{f^{(4)}(x_*)}{384}h^5 + \cdots \ .$$

For the midpoint rule, this leads to a global error expansion in even powers of h, $E \approx A_1h^2 + A_2h^4 + \cdots$, with $A_1 = (f'(b) - f'(a))/24$. Each of the remaining panel methods is symmetric about the center of the panel. This implies that each of them has local truncation error containing only odd powers of h and global error containing only even powers of h.

The leading power of h in the error expansion is the order of accuracy. It can be determined by a simple observation: the order of the local truncation error is one more than the degree of the lowest monomial that is not integrated exactly by the panel method. For example, the rectangle rule integrates $f(x) = x^0 \equiv 1$ exactly but gets $f(x) = x^1 \equiv x$ wrong. The order of the lowest monomial not integrated exactly is 1 so the local truncation error is $O(h^2)$ and the global error is $O(h)$. The midpoint rule integrates x^0 and x^1 correctly but gets x^2 wrong. The order of the lowest monomial not integrated exactly is 2 so the local truncation error is $O(h^3)$ and the global error is $O(h^2)$. If the generic panel has x_* in the center, then

$$\int_P (x - x_*)^n \, dx$$

is always done exactly if n is odd. This is because both the exact integral and its panel method approximation are zero by symmetry.

To understand why this rule works, think of the Taylor expansion of $f(x)$ about the midpoint, x_*. This approximates f by a sum of monomials. Applying the panel integral approximation to f is the same as applying the approximation to each monomial and summing the results. Moreover, the integral of a monomial $(x - x_*)^n$ over P is proportional to h^{n+1}, as is the panel method approximation to it, regardless of whether the panel method is exact or not. The first monomial that is not integrated exactly contributes something proportional to h^{n+1} to the error.

Using this rule it is easy to determine the accuracy of the approximations in Figure 3.5. The trapezoid rule integrates constants and linear functions exactly, but it gets quadratics wrong. This makes the local truncation error third order and the global error second order. The Simpson's rule coefficients 1/6 and 2/3 are designed exactly to integrate constants and quadratics exactly, which they do. Simpson's rule integrates cubics exactly (by symmetry) but gets quartics wrong. This gives Simpson's rule fourth order global accuracy. The two point Gauss quadrature also does constants and quadratics correctly but quartics wrong (check this!). The three point Gauss quadrature rule does constants, quadratics, and quartics correctly but gets $(x - x_*)^6$ wrong. That makes it sixth order accurate.

3.5 The method of undetermined coefficients

The method of undetermined coefficients is a general way to find an approximation formula of a desired type. Suppose we want to estimate some A in terms of given data $g_1(h)$, $g_2(h)$, ... The method is to assume a linear estimation formula of the form

$$\widehat{A}(h) = a_1(h)g_1(h) + a_2(h)g_2(h) + \cdots \ , \tag{3.37}$$

then determine the unknown coefficients $a_k(h)$ by matching Taylor series up to the highest possible order. The coefficients often take the form of a constant times some power of h: $a_k(h) = a_k h^{p_k}$. The algebra is simpler if we guess or figure out the powers first. The estimator is *consistent* if $\widehat{A}(h) - A \to 0$ as $h \to \infty$. Generally (but not always), being consistent is the same as being at least first order accurate. At the end of our calculations, we may discover that there is no consistent estimator of the desired type.

We illustrate the method in a simple example: estimate $f'(x)$ from $g_1 = f(x)$ and $g_2 = f(x+h)$. As above, we will leave out the argument x whenever possible and write f for $f(x)$, f' for $f'(x)$, etc. The estimator is (dropping the x argument)

$$f' \approx \widehat{A} = a_1(h)f + a_2(h)f(x+h) \ .$$

Now expand in Taylor series:

$$f(x+h) = f + f'h + \frac{1}{2}f'' + \cdots \ .$$

The estimator is

$$\widehat{A} = a_1(h)f + a_2(h)f + a_2(h)f'h + a_2(h)f''h^2 + \cdots \ . \tag{3.38}$$

Looking at the right hand side, we see various coefficients, f, f', and so on. Since the relation is supposed to work whatever the values of f, f', etc. may be, we choose a_1 and a_2 so that the coefficient of f is zero. From (3.38), this leads to

$$0 = a_1(h) + a_2(h) \ .$$

To make the estimator consistent, we try

$$1 = a_2(h)h \ .$$

These two conditions lead to

$$a_2 = \frac{1}{h} \ , \quad a_1 = \frac{-1}{h} \ , \tag{3.39}$$

so the estimate is

$$\begin{aligned} f'(x) \approx \widehat{A} &= \frac{-1}{h}f(x) + \frac{1}{h}f(x+h) \\ &= \frac{f(x+h) - f(x)}{h} \ . \end{aligned}$$

This is the first order one sided difference approximation we saw earlier. Plugging the values (3.39) into (3.38) shows that the estimator satisfies $\widehat{A} = f' + O(h)$, which is the first order accuracy we found before.

A more complicated problem is to estimate $f'(x)$ from $f(x)$, $f(x-h)$, $f(x+h)$, $f(x+2h)$. This is not centered nor is it completely one sided, but it is biased to one side. It has proven useful in high accuracy wave simulations. This time we guess that all the coefficients have a power $1/h$, as all the estimates of f' so far have this property. Thus assume the form:

$$f' \approx \widehat{A} = \frac{1}{h}\left(a_{-1}f(x-h) + a_0 f + a_1 f(x+h) + a_2 f(x+2h)\right) \quad .$$

The Taylor series expansions are

$$\begin{array}{ccccccccccc}
f(x-h) & = & f & - & f'h & + & \frac{f''}{2}h^2 & - & \frac{f'''}{6}h^3 & + & \frac{f^{(4)}}{24}h^4 + \cdots \\
f(x+h) & = & f & + & f'h & + & \frac{f''}{2}h^2 & + & \frac{f'''}{6}h^3 & + & \frac{f^{(4)}}{24}h^4 + \cdots \\
f(x+2h) & = & f & + & 2f'h & + & 2f''h^2 & + & \frac{4f'''}{3}h^3 & + & \frac{2f^{(4)}}{3}h^4 + \cdots
\end{array}$$

Equating powers of h turns out to be the same as equating the coefficients of f, f', etc. from both sides:

$$\begin{array}{lll}
f\ ,\ \ O(h^{-1}) : & & 0 = a_{-1} + a_0 + a_1 + a_2 \\
f'\ ,\ \ O(h^{0}) : & & 1 = -a_{-1} + a_1 + 2a_2 \\
f''\ ,\ \ O(h^{1}) : & & 0 = \frac{1}{2}a_{-1} + \frac{1}{2}a_1 + 2a_2 \\
f'''\ ,\ \ O(h^{2}) : & & 0 = \frac{-1}{6}a_{-1} + \frac{1}{6}a_1 + \frac{4}{3}a_2
\end{array} \tag{3.40}$$

We could compute the $O(h^3)$ equation but already we have four equations for the four unknown coefficients. If we would use the $O(h^3)$ equation in place of the $O(h^2)$ equation, we loose an order of accuracy in the resulting approximation.

These are a system of 4 linear equations in the four unknowns a_{-1} through a_2, which we solve in an ad hoc way. Notice that the combination $b = -a_{-1} + a_1$ appears in the second and fourth equations. If we substitute b, these equations are

$$1 = b + 2a_2 \ ,$$

$$0 = \frac{1}{6}b + \frac{4}{3}a_2 \ .$$

which implies that $b = -8a_2$ and then that $a_2 = -\frac{1}{6}$ and $b = \frac{4}{3}$. Then, since $-4a_2 = \frac{2}{3}$, the third equation gives $a_{-1} + a_1 = \frac{2}{3}$. Since $b = \frac{4}{3}$ is known, we get two equations for a_{-1} and a_1:

$$a_1 - a_{-1} = \frac{4}{3} \ ,$$

$$a_1 + a_{-1} = \frac{2}{3} \ .$$

The solution is $a_1 = 1$ and $a_{-1} = \frac{-1}{3}$. With these, the first equation leads to $a_0 = \frac{-1}{2}$. Finally, our approximation is

$$f'(x) = \frac{1}{h}\left(\frac{-1}{3}f(x-h) - \frac{1}{2}f(x) + f(x+h) - \frac{1}{6}f(x+2h)\right) + O(h^3) \; .$$

Note that the first step in this derivation was to approximate f by its Taylor approximation of order 3, which would be exact if f were a polynomial of order 3. The derivation has the effect of making $\widehat{A}$ exact on polynomials of degree 3 or less. The four equations (3.40) arise from asking $\widehat{A}$ to be exact on constants, linear functions, quadratics, and cubics. We illustrate this approach with the problem of estimating $f''(x)$ as accurately as possible from $f(x)$, $f(x+h)$, $f'(x)$ and $f'(x+h)$. The estimator we seek has the form

$$f'' \approx \widehat{A} = af + bf(x+h) + cf' + df'(x+h) \; .$$

We can determine the four unknown coefficients a, b, c, and d by requiring the approximation to be exact on constants, linears, quadratics, and cubics. It does not matter what x value we use, so let us take $x = 0$. This gives, respectively, the four equations:

$$\begin{array}{ll} 0 = a + b & (\text{constants, } f = 1) \; , \\ 0 = bh + c + d & (\text{linears, } f = x) \; , \\ 1 = b\dfrac{h^2}{2} + dh & (\text{quadratics, } f = x^2/2) \; , \\ 0 = b\dfrac{h^3}{6} + d\dfrac{h^2}{2} & (\text{cubics, } f = x^3/6) \; . \end{array}$$

Solving these gives

$$a = \frac{-6}{h^2} \;\; , \quad b = \frac{6}{h^2} \;\; , \quad c = \frac{-4}{h} \;\; , \quad d = \frac{-2}{h} \;\; .$$

and the approximation

$$f''(x) \approx \frac{6}{h^2}\big(-f(x) + f(x+h)\big) - \frac{2}{h}\big(2f'(x) + f'(x+h)\big) \; .$$

A Taylor series calculation shows that this is second order accurate.

3.6 Adaptive parameter estimation

In most real computations, the computational strategy is not fixed in advance, but is adjusted *adaptively* as the computation proceeds. If we are using one of the approximations $\widehat{(A)}(h)$, we might not know an appropriate h when we write the program, and the user might not have the time or expertise to choose h for each application. For example, exercise 12 involves hundreds or thousands of numerical integrations. It is out of the question for the user to experiment manually to find a good h for each one. We need instead a systematic computational procedure for finding an appropriate step size.

Suppose we are computing something about the function f, a derivative or an integral. We want a program that takes f, and a desired level of accuracy[10], e, and returns $\widehat{A}$ with $\left|\widehat{A} - A\right| \leq e$ with a high degree of confidence. We have $\widehat{A}(h)$ that we can evaluate for any h, and we want an automatic way to choose h so that $\left|\widehat{A}(h) - A\right| \leq e$. A natural suggestion would be to keep reducing h until the answer stops changing. We seek a quantitative version of this.

Asymptotic error expansions of Section 3.3 give one approach. For example, if $\widehat{A}(h)$ is a second order accurate approximation to an unknown A and h is small enough we can estimate the error using the leading term:

$$E(h) = \widehat{A}(h) - A \approx A_1 h^2 \ .$$

We can estimate $A_1 h^2$ from $\widehat{A}(h)$ and $\widehat{A}(2h)$ using the ideas that give (3.28). The result is the Richardson extrapolation error estimate

$$E(h) \approx A_1 h^2 \approx \frac{1}{3}\big(\widehat{A}(2h) - \widehat{A}(h)\big) \ . \tag{3.41}$$

The adaptive strategy would be to keep reducing h by a factor of two until the estimated error (3.41) is within the tolerance[11]:

$$\begin{array}{lll}
\texttt{for} \;\; (& & \\
\quad \text{evaluate } \widehat{A}(2h) \text{ and } \widehat{A}(h) \; ; & \texttt{// initialize} & \\
\quad \left|\widehat{A}(2h) - \widehat{A}(h)\right| \geq 3\epsilon \; ; & \texttt{// stopping test} & (3.42) \\
\quad \{ \; \texttt{h = h/2;} \;\; \text{evaluate } \widehat{A}(h) \; \} \;\; \texttt{;} & \texttt{// increment} & \\
) \; ; & &
\end{array}$$

A natural strategy might be to stop when $\left|\widehat{A}(2h) - \widehat{A}(h)\right| \leq e$. Our quantitative asymptotic error analysis shows that this strategy is off by a factor of 3. We achieve accuracy roughly e when we stop at $\left|\widehat{A}(2h) - \widehat{A}(h)\right| \leq 3e$. This is because $\widehat{A}(h)$ is more accurate than $\widehat{A}(2h)$.

We can base reasonably reliable software on refinements of the basic strategy (3.42). Some drawbacks of (3.42) are that

1. It needs an *initial guess*, a starting value of h.

2. It may be an infinite loop.

3. It might terminate early if the initial h is outside the *asymptotic range* where error expansions are accurate.

4. If $\widehat{A}(h)$ does not have an asymptotic error expansion, the program will not detect this.

[10] This is absolute error. We also could seek a bound on relative error: $\left|\widehat{A} - A\right| / |A| \leq \epsilon$.

[11] In the increment part we need not evaluate $\widehat{A}(2h)$ because this is what we called $\widehat{A}(h)$ before we replaced h with $h/2$.

5. It does not return the best possible estimate of A.

A plausible initial guess, h_0, will depend on the scales (length or time, etc.) of the problem. For example 10^{-10} meters is natural for a problem in atomic physics but not in airplane design. The programmer or the user should supply h_0 based on understanding of the problem. The programmer can take $h_0 = 1$ if he or she thinks the user will use natural units for the problem (Ångströms for atomic physics, meters for airplanes). It might happen that you need $h = h_0/1000$ to satisfy (3.42), but you should give up if $h = h_0 \cdot \epsilon_{\text{mach}}$. For integration we need an initial $n = (b-a)/h$. It might be reasonable to take $n_0 = 10$, so that $h_0 = (b-a)/10$.

Point 2 says that we need some criterion for giving up. As discussed more in Section 3.7 we should anticipate the ways our software can fail and report failure. When to give up should depend on the problem. For numerical differentiation, we can stop when roundoff or propagated error from evaluating f (see Chapter 2, Section?) creates an error as big as the answer. For integration limiting the number of refinements to 20, would limit the number of panels to $n_0 \cdot 2^{20} \approx n_0 \cdot 10^6$. The revised program might look like

```
h = h0;
hMin = 10*macheps*h0;   //macheps = machine precision
while  ( |Â(2h) − Â(h)| ≥ 3e )
  if ( h <= hMin ) {
    Print an error message.
    errorCode = HMIN_REACHED; return -1;}
  h = h/2 ;
return  A(h) ;
```
(3.43)

```
h = h0;
hMin = 10*macheps*h0;   //macheps = machine precision
for  (
  evaluate Â(2h) and Â(h) ;          // initialize
  |Â(2h) − Â(h)| ≥ 3ε ;              // stopping test
  { h = h/2;  evaluate Â(h) } ;      // increment
  )  {
    if ( h <= hMin ) {
    Print an error message;
    errorCode = HMIN_REACHED;
    return -1;}
   }
```
(3.44)

We cannot have perfect protection from point 3, though *premature termination* is unlikely if h_0 is sensible and e (the desired accuracy) is small enough. A more cautious programmer might do more convergence analysis, for example asking that the $\widehat{A}(4h)$ and $\widehat{A}(2h)$ error estimate be roughly 2^p times larger than the $\widehat{A}(2h)$ and $\widehat{A}(h)$ estimate. There might be irregularities in $f(x)$, possibly jumps in some derivative, that prevent the asymptotic error analysis but do not prevent convergence. It would be worthwhile returning an error flag in this case, as some commercial numerical software packages do.

Part of the risk in point 4 comes from the possibility that $\widehat{A}(h)$ converges more slowly than the hoped for order of accuracy suggests. For example if $\widehat{A}(h) \approx A + A_1 h^{1/2}$, then the error is three times that suggested by (3.42). The extra convergence analysis suggested above might catch this.

Point 5 is part of a paradox afflicting many error estimation strategies. We estimate the size of $E(h) = \widehat{A}(h) - A$ by estimating the value of $E(h)$. This leaves us a choice. We could ignore the error estimate (3.41) and report $\widehat{A}(h)$ as the approximate answer, or we could subtract out the estimated error and report the more accurate $\widehat{A}(h) - \widehat{E}(h)$. This is the same as applying one level of Richardson extrapolation to $\widehat{A}$. The corrected approximation probably is more accurate, but we have no estimate of its error. The only reason to be dissatisfied with this is that we cannot report an answer with error less than e until the error is far less than e.

3.7 Software

There are several things a scientific programmer can do to make codes easier to debug and more reliable. Everyone has had the experience of *breaking* a code, making a change that for some reason makes the whole thing stop working. A program that is designed to be changed is *flexible*, less likely to be broken in this way. *Modular* programs have different pieces in separate procedures (methods, subroutines) that can be tested separately. Many of the modules can be tested using *convergence analysis* described in Section 3.3.2. Programmers should go far as possible to prevent *silent failure*. It is better to have no answer than a wrong one.

3.7.1 Flexible programming

Suppose you want to compute $I = \int_0^2 f(x)dx$ using a panel method. The rectangle rule $\widehat{I} = \Delta x \sum_{k=0}^{n-1} f(x_k)$ with $n = 100$ could be coded:

```
double I = 0;                       // line 1
for ( int k = 0; k < 100; k++)  // line 2
   I += .02*f(.02*k);               // line 3
```

Here the number $n = 100$ is *hard wired*, which means built into the code in a way that makes it hard to change on a whim. It would be easy to break this code by changing line 2 to `for ( int k = 0; k < 90; k++)` but forgetting to change .02 to $2/90 \approx .0222$, or changing it in only one place: `I += (2/90)*f(.02*k);`. By the way, this last has the bug that `(2/90)` evaluates to zero because it is an integer divide.

A more flexible version would be:

```
int n    = 100;   // The number of points
double a = 0;     // The left endpoint for integration
double b = 2;     // The right endpoint for integration
double dx = ( b - a ) / n;   // Width of a panel

double I = 0;                    // line 1
double x;
for ( int k = 0; k < 100; k++) { // line 2
   x  = k*dx;
   I += dx*f(x);                // line 3
  }
```

Changing to `int n = 90;` would give a correct program. Because the variable `dx` has a name, you can check in the debugger that it has the correct value. The more flexible version has a few more lines of code which take a few extra seconds to type.

You probably will want to test each module of your program on a problem you know the answer to. For example, if you are calculating $H(T) = \int_0^T e^{\sin(x)} dx$, you would code it in a way that it is easy to apply to $G(T) = \int_0^T e^x dx$.

3.7.2 Modular programming

Modular programming is helpful for any software project. In scientific computing we design procedures that can be tested separately on their own test problems. For example, if we apply adaptive Richardson extrapolation as in Section 3.6 for an integral, we would write a generic adaptive Richardson extrapolation program and test it on data that does not come from an integrator, then we would write an integrator and test it outside the Richardson procedure. If the two procedures work correctly separately, they have a good chance to work well together.

Designing scientific software, or any software, involves more than choosing data structures and procedure interfaces. You also need a testing plan. The code should be designed so that the modules can be tested separately. For example, even if I know my production integration code will use $n = 50$ panels, I might put the number of panels as a calling argument so I can do a convergence study.

3.7.3 Report failure

Error handling is another indispensable part of software design. Any module that can fail (most can) must have a way to report failure. The designer will choose an appropriate mechanism. The simplest is just to print something and stop the program when something goes wrong. This is not appropriate for commercial software but could be fine for a research code that only the author will run. More sophisticated might be to write to a log file and return an error flag, or even to throw an exception that could be caught by the calling routine.

Equally important is detecting failure. This means checking and forwarding all the internal failure reporting, such as:

```
double *vec;                     // Treat vec as an array.
vec = new double[n];             // Allocate memory for n values
if ( vec == NIL ) .. REPORT ..   // Complain if didn't work.
```

It also means checking calling arguments for plausibility:

```
#define MAXN 10000  /* Largest allowed # of panels */
int Integrate(                  // Integrate f(x)
   double (*f)(double),         // supply f implicitly
   double  *I,                  // return the estimated integral
   int      n) {                // using n panels.

   if ( n <= 0 ) {
      .. complain that n is too small ..;
      return 1;  // The error flag for negative n.
   if ( n > MAXN ) {
      cout << "In Integrate, got n = " << n << " greater than"
           << " MAXN = << MAXN << endl;
      return 2; // The error flag for n too large.

    . . .do the integral . . .
   *I = . . . answer . . .
   return 0;  //  error flag = 0 means it worked.
 }
```

Of course, this only works if the calling program checks the error flag, i.e. `eFlag = Integrate( f, @I, n); if (eFlag) ..  WRONG ANSWER ..  ;`

Lastly, it means thinking of all loops that could be infinite loops, such as

```
while ( error > targetError ){ // try to get error < targetError

  ...  make the solution more accurate...;

 }
```

This could be an infinite loop if `targetError` is too small or the asymptotic error expansion is wrong. Instead, at least put in a trip counter

```
#define MAX_TRIP_COUNT 1000  /* stop a runaway refinement loop  */
int tripCount = 0;
while ( error > targetError )  {

  ...  make the solution more accurate...;

if ( ++ tripCount > MAX_TRIP_COUNT ) report error & quit.
}
```

3.8 References and further reading

For a review of one variable calculus, I recommend the Schaum outline. The chapter on Taylor series explains the remainder estimate clearly.

There several good old books on classical numerical analysis. Two favorites are *Numerical Methods* by Germund Dahlquist and Åke Björk, and *Analysis of Numerical Methods* by Gene Isaacson and Herb Keller. Particularly interesting subjects are the symbolic calculus of finite difference operators and Gaussian quadrature.

There are several applications of convergent Taylor series in scientific computing. One example is the *fast multipole method* of Leslie Greengard and Vladimir Rokhlin.

3.9 Exercises

1. Verify that (3.23) represents the leading error in the approximation (3.22). *Hint*, this does not require multidimensional Taylor series. Why?

2. Use multidimensional Taylor series to show that the *rotated* five point operator

$$\frac{1}{2h^2}\big(f(x+h,y+h)+f(x+h,y-h)+f(x-h,y+h)+f(x-h,y-h)-4f\big)$$

is a consistent approximation to $\triangle f$. Use a symmetry argument to show that the approximation is at least second order accurate. Show that the leading error term is

$$\frac{h^2}{12}\left(\partial_x^4 f + 6\partial_x^2\partial_y^2 f + \partial_y^4 f\right) .$$

3. What coefficient should we use in place of $\frac{4}{3}$ if $\widehat{A}(h)$ is first order accurate? Find the coefficient as a function of p, the order of accuracy.

4. Find a formula that estimates $f''(x)$ using the four values $f(x)$, $f(x+h)$, $f(x+2h)$, and $f(x+3h)$ with the highest possible order of accuracy. What is this order of accuracy? For what order polynomials does the formula give the exact answer?

5. Suppose we have panels P_k as in (3.31) and panel averages $F_k = \int_{P_k} f(x)dx/(x_{k+1}-x_k)$.

   (a) What is the order of accuracy of F_k as an estimate of $f((x_k + x_{k+1})/2) = f(x_{k+1/2})$?

   (b) Assuming the panels all have size h, find a higher order accurate estimate of $f(x_{k+1/2})$ using F_k, F_{k-1}, and F_{k+1}.

6. This discusses the function (3.12) more carefully.

(a) Show that the Taylor series for $g(x) = 1/(1-x)$ about $x = 0$ is the geometric series $1/(1-x) = \sum_{n=0}^{\infty} x^n$, which converges for $|x| < 1$.

(b) Prove the simple remainder formula $g(x) = \sum_{n=0}^{p} x^n + R_p(x)$ with $R_p(x) = x^{p+1}/(1-x)$. Hint: factor x^{p+1} out of $R_p(x) = \sum_{n=p+1}^{\infty} x^n$.

(c) Use the formula $-h\partial_x e^{-x/h} = e^{-x/h}$ to prove that $\int_0^\infty e^{-x/h} x^n dx = nh \int_0^\infty e^{-x/h} x^{n-1} dx$, and then $\int_0^\infty e^{-x/h} x^n dx = n! h^{n+1}$.

(d) Use the same integration by parts and (3.11) to show that $\int_{1/2}^\infty e^{-x/h} x^n dx = O(x^p)$ for any p and n.

(e) Note that for $0 \leq x \leq 1/2$, $1/(1-x) \leq 2$. Use this and part b to prove the remainder bound

$$\begin{aligned} \int_0^{1/2} e^{-x/h} R_p(x) dx &\leq 2 \int_0^{1/2} e^{-x/h} x^{p+1} dx \\ &\leq 2 \int_0^{\infty} e^{-x/h} x^{p+1} dx = O(h^{p+1}) \; . \end{aligned}$$

(f) Show that $\int_0^{1/2} e^{-x/h} x^n dx = \int_0^\infty e^{-x/h} x^n dx + O(x^p)$ for any n and p.

(g) Conclude that (3.13) indeed is a valid asymptotic expansion.

7. An application requires accurate values of $f(x) = e^x - 1$ for x very close to zero.

(a) Show that the problem of evaluating $f(x)$ is well conditioned for small x.

(b) How many digits of accuracy would you expect from the code `f = exp(x) - 1;` for $x \sim 10^{-5}$ and for $x \sim 10^{-10}$ in single and in double precision?

(c) Let $f(x) = \sum_{n=1}^{\infty} f_n x^n$ be the Taylor series about $x = 0$. Calculate the first three terms, the terms involving x, x^2, and x^3. Let $p(x)$ be the degree three Taylor approximation of $f(x)$ about $x = 0$.

(d) Assuming that x_0 is so small that the error is nearly equal to the largest neglected term, estimate $\max |f(x) - p(x)|$ when $|x| \leq x_0$.

(e) We will evaluate $f(x)$ using

```
if ( abs(x) > x0 ) f = exp(x) - 1;
else               f = p3(x); // given by part c.
```

What x_0 should we choose to maximize the accuracy of $f(x)$ for $|x| < 1$ assuming double precision arithmetic and that the exponential function is evaluated to full double precision accuracy (exact answer correctly rounded)?

8. Suppose that $f(x)$ is a function that is evaluated to full machine precision but that there is ϵ_{mach} rounding error in evaluating $\widehat{A} = (f(x+h) - f(x))/h$. What value of h minimizes the total error including both rounding and truncation error? This will be $h_*(\epsilon_{mach}) \sim \epsilon_{mach}^q$. Let $e_*(\epsilon_{mach})$ be the resulting best estimate of $f'(x)$. Show that $e_* \sim \epsilon_{mach}^r$ and find r.

9. Repeat Exercise 8 with the two point centered difference approximation to $f'(x)$. Show that the best error possible with centered differencing is much better than the best possible with the first order approximation. This is one of the advantages of higher order finite difference approximations.

10. Verify that the two point Gauss quadrature formula of Figure 3.5 is exact for monomials of degree less than six. This involves checking the functions $f(x) = 1$, $f(x) = x^2$, and $f(x) = x^4$ because the odd order monomials are exact by symmetry. Check that the three point Gauss quadrature formula is exact for monomials of degree less than 8.

11. Find the replacement to adaptive halting criterion (3.41) for a method of order p.

12. We want to know how the function,

$$f(t) = \int_0^1 \cos\left(tx^2\right) dx \quad , \tag{3.45}$$

behaves for large t. There is an approximation,

$$f(t) \sim \sqrt{\frac{\pi}{8t}} + \frac{1}{2t}\sin(t) - \frac{1}{16t^2}\cos(t) + \cdots \quad , \tag{3.46}$$

that is supposed to hold for large t. We want to know how accurate this approximation is and how large t has to be before it is useful. This exercise goes through some of the steps that go into creating scientific software to investigate this question numerically: (a) Design the basic code to be modular and robust, (b) Check that is gives the right answer in one case you can check (note: not the problem of interest), (c) Check that the basic computational method gives the accuracy it should. (d) Write an automatic adaptive version of the code that allows it to produce accurate answers *hands off*, without manual tuning of computational parameters (Δx in this case) for each run. (e) Do the science. To appreciate the value of part (d), the science step involves thousands of integrations to evaluate $f(t)$ for various t values. Nobody wants to do a thousand runs choosing Δx by hand for each one.

   (a) Write a procedure (or "method") to estimate $f(t)$ using a panel integration method with uniformly spaced points. The procedure should be well documented, robust, and clean. Robust will mean many things in later exercises. Here is means: (i) that it should use the correct number of panels even though the points x_k are computed in inexact floating point arithmetic, and (ii) that the procedure will return an error code and possibly print an error message if one of the calling arguments is out of range (here, probably just $n \le 0$). It should take as inputs t and n and return the approximate integral with that t, and $\Delta x = 1/n$. If your panel integration formula uses endpoints of the panel, you must write the code so that $f(x_k)$ is evaluated only once. This routine should be written so that another person could easily substitute a different panel method or a different integrand by changing a few lines of code.

(b) Check the correctness of the procedure from part (a) by seeing whether it gives the right answer for small t. We can estimate $f(t)$ for small t using a few terms of its Taylor series. From $\cos(u) \approx 1 - \frac{1}{2}u^2 + \frac{1}{24}u^4 + \cdots$, we get $\cos(tx^2) \approx 1 - \frac{1}{2}t^2x^4 + \cdots$. Integrating this approximation over the x interval gives $f(t) \approx 1 - \frac{1}{10}t^2 + \cdots$. Compute the next few terms and use them to get an approximate value of $f(t)$ for small t. Then write a *driver* program that calls both the integration procedure, and the procedure that approximates f using Taylor series, and compares the results. Use reasonable but not huge values of n.

(c) With $t = 1$, do a convergence study to verify the second order accuracy of the trapezoid rule and the fourth order accuracy of Simpson's rule. This requires you to write a different driver to call the integration procedure with several values of n and compare the answers in the manner of a convergence study. Once you have done this for the trapezoid rule, it should take less than a minute to redo it for Simpson's rule. This is how you can tell whether you have done part (a) well.

(d) Write a procedure that uses the basic integration procedure from part (a), together with Richardson error estimation to find an n that gives $f(t)$ to within a specified error tolerance. The procedure should work by repeatedly doubling n until the estimated error, based on comparing approximations, is less than the tolerance given. This routine should be robust enough to quit and report failure if it is unable to achieve the requested accuracy. The input should be t and the desired error bound. The output should be the estimated value of f, the number of points used, and an error flag to report failure. Before applying this procedure to the panel integration procedure, apply it to the fake procedure `fakeInt.c` or `fakeInt.C`. Note that these testers have options to make the Richardson program fail or succeed. Try it both ways, to make sure the robustness feature of your Richardson procedure works. Include with your homework output illustrating the behavior of your Richardson procedure when it fails.

(e) Here is the science part of the problem. Make a few plots showing f and its approximations using one, two and all three terms on the right side of (3.46) for t in the range $1 \le t \le 1000$. In all cases we want to evaluate f so accurately that the error in our f value is much less than the error of the approximation (3.46). Note that even for a fixed level of accuracy, more points are needed for large t. Plot the integrand to see why.

Chapter 4

Linear Algebra I, Theory and Conditioning

4.1 Introduction

Linear algebra and calculus are the basic tools of quantitative science. The operations of linear algebra include solving systems of equations, finding subspaces, solving least squares problems, factoring matrices, and computing eigenvalues and eigenvectors. In practice most of these operations will be done by software packages that you buy or download. This chapter discusses formulation and condition number of the main problems in computational linear algebra. Chapter 5.1 discusses algorithms.

Conditioning is the primary concern in many practical linear algebra computations. Easily available linear algebra software is *stable* in the sense that the results are as accurate as the conditioning of the problem allows. Unfortunately, condition numbers as large as 10^{18} occur in not terribly large or rare practical problems. The results of such a calculation in double precision would be completely unreliable.

If a computational method for a well conditioned problem is unstable (much less accurate than its conditioning allows), it is likely because one of the subproblems is ill conditioned. For example, the problem of computing the matrix exponential, e^A, may be well conditioned while the problem of computing the eigenvalues and eigenvectors of A is ill conditioned. A stable algorithm for computing e^A in that case must avoid using the eigenvalues and eigenvectors of A, see Exercise 12.

The condition number measures how small perturbations in the data affect the answer. This is called *perturbation theory* in linear algebra. Suppose[1] A is a matrix and $f(A)$ is the solution of a linear algebra problem involving A, such as x that satisfies $Ax = b$, or λ and v that satisfies $Av = \lambda v$. Perturbation theory seeks to estimate $\Delta f = f(A + \Delta A) - f(A)$ when ΔA is small. Usually, this amounts to calculating the derivative of f with respect to A. We often do this by applying implicit differentiation to the relevant equations (such as $Ax = b$).

It often is helpful to simplify the results of perturbation calculations using simple bounds that involve vector or matrix *norms*. For example, suppose we want to say that all the entries in ΔA or Δv are small. For a vector, v, or a matrix, A, the norm, $\|v\|$ or $\|A\|$, is a number that characterizes the size of v or A. Using norms, we can say that the relative size of a perturbation in A is $\|\Delta A\| / \|A\|$.

The condition number of a problem involving A depends on the problem as well as on A. For example, the condition number of $f(A) = A^{-1}b$, the problem of finding x so that $Ax = b$, informally[2] is given by

$$\|A\| \left\|A^{-1}\right\| \ . \tag{4.1}$$

The problem of finding the eigenvectors of A has a condition number that does

[1] This notation replaces our earlier $A(x)$. In linear algebra, A always is a matrix and x never is a matrix.

[2] To get this result, we not only maximize over ΔA but also over b. If the relative error really were increased by a factor on the order of $\|A\| \left\|A^{-1}\right\|$ the finite element method, which is the main computational technique for structural analysis, would not work.

not resemble (4.1). For example, finding eigenvectors of A can be well conditioned even when $\|A^{-1}\|$ is infinite (A is singular).

There are several ways to represent an $n \times n$ matrix in the computer. The simplest is to store the n^2 numbers in an $n \times n$ array. If this direct storage is efficient, we say A is *dense*. Much of the discussion here and in Chapter 5.1 applies mainly to dense matrices. A matrix is *sparse* if storing all its entries directly is inefficient. A modern (2006) desktop computer has enough memory for n^2 numbers if if n is less than about[3] $50,000$. This makes dense matrix methods impractical for solving systems of equations with more than $50,000$ variables. The computing time solving $n = 50,000$ linear equations in this way would be about a day. Sparse matrix methods can handle larger problems and often give faster methods even for problems that can be handled using dense matrix methods. For example, finite element computations often lead to sparse matrices with orders of magnitude larger n that can be solved in minutes

One way a matrix can be sparse is for most of its entries to be zero. For example, discretizations of the Laplace equation in three dimensions have as few as seven non-zero entries per row, so that $7/n$ is the fraction of entries of A that are not zero. Sparse matrices in this sense also arise in circuit problems, where a non-zero entry in A corresponds to a direct connection between two elements in the circuit. Such matrices may be stored in *sparse matrix format*, in which we keep lists noting which entries are not zero and the values of the non-zero elements. Computations with such sparse matrices try to avoid *fill in*. For example, they would avoid explicit computation of A^{-1} because most of its entries are not zero. Sparse matrix software has heuristics that often do very well in avoiding fill in. The interested reader should consult the references.

In some cases it is possible to compute the *matrix vector product* $y = Ax$ for a given x efficiently without calculating the entries of A explicitly. One example is the *discrete Fourier transform* (DFT) described in Chapter 1. This is a *full* matrix (every entry different from zero) with n^2 non-zero entries, but the FFT (*fast Fourier transform*) algorithm computes $y = Ax$ in $O(n \log(n))$ operations. Another example is the *fast multipole method* that computes forces from mutual electrostatic interaction of n charged particles with b bits of accuracy in $O(nb)$ work. Many finite element packages never *assemble* the stiffness matrix, A.

Computational methods can be *direct* or *iterative*. A direct method would get the exact answer in exact arithmetic using a predetermined number of arithmetic operations. For example, Gauss elimination computes the LU factorization of A using $O(n^3)$ operations. Iterative methods produce a sequence of approximate solutions that converge to the exact answer as the number of iterations goes to infinity. They usually are faster than direct methods for very large problems, particularly when A is sparse.

[3]With n^2 floating point numbers and 8 bytes per number (double precision), we need $50,000^2 \times 8 = 2 \cdot 10^{10}$ bytes, which is 20GBytes.

4.2 Review of linear algebra

This section recalls some aspects of linear algebra we make use of later. It is not a substitute for a course on linear algebra. We make use of many things from linear algebra, such as matrix inverses, without explanation. People come to scientific computing with vastly differing points of view in linear algebra. This section should give everyone a common language.

4.2.1 Vector spaces

Linear algebra gets much of its power through the interaction between the abstract and the concrete. Abstract linear transformations are represented by concrete arrays of numbers forming a matrix. The set of solutions of a *homogeneous* system of equations forms an abstract subspace of R^n that we can try to characterize. For example, a basis for such a subspace may be computed by factoring a matrix in a certain way.

A vector space is a set of elements that may be added and multiplied by *scalar* numbers[4] (either real or complex numbers, depending on the application). Vector addition is commutative ($u + v = v + u$) and associative ($(u + v) + w = u + (v + w)$). Multiplication by scalars is distributive over vector addition ($a(u + v) = au + av$ and $(a + b)u = au + bu$ for scalars a and b and vectors u and v). There is a unique zero vector, 0, with $0 + u = u$ for any vector u.

The standard vector spaces are R^n (or C^n), consisting of column vectors

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ u_n \end{pmatrix}$$

where the *components*, u_k, are arbitrary real (or complex) numbers. Vector addition and scalar multiplication are done componentwise.

If V is a vector space and $V' \subset V$, then we say that V' is a *subspace* of V if V' is also a vector space with the same vector addition and scalar multiplication operations. We may always add elements of V' and multiply them by scalars, but V' is a subspace if the result always is an element of V'. We say V' is a subspace if it is *closed* under vector addition and scalar multiplication. For example, suppose $V = R^n$ and V' consists of all vectors whose components sum to zero ($\sum_{k=1}^n u_k = 0$). If we add two such vectors or multiply by a scalar, the result also has the zero sum property. On the other hand, the set of vectors whose components sum to one ($\sum_{k=1}^n u_k = 1$) is not closed under vector addition or scalar multiplication.

[4]Physicists use the word "scalar" in a different way. For them, a scalar is a number that is the same in any coordinate system. The components of a vector in a particular basis are not scalars in this sense.

A *basis* for vector space V is a set of vectors f_1, …, f_n so that any $u \in V$ may be written in a unique way as a *linear combination* of the vectors f_k:

$$u = u_1 f_1 + \cdots + u_n f_n \ ,$$

with scalar *expansion coefficients* u_k. The standard vector spaces R^n and C^n have standard bases e_k, the vector with all zero components but for a single 1 in position k. This is a basis because

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ u_n \end{pmatrix} = u_1 \begin{pmatrix} 1 \\ 0 \\ \cdot \\ \cdot \\ 0 \end{pmatrix} + u_2 \begin{pmatrix} 0 \\ 1 \\ \cdot \\ \cdot \\ 0 \end{pmatrix} + \cdots + u_n \begin{pmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ 1 \end{pmatrix} = \sum_{k=1}^{n} u_k e_k \ .$$

In view of this, there is little distinction between coordinates, components, and expansion coefficients, all of which are called u_k. If V has a basis with n elements, we say the *dimension* of V is n. It is possible to make this definition because of the theorem that states that every basis of V has the same number of elements. A vector space that does not have a finite basis is called *infinite dimensional*[5]. The vector space of all polynomials (with no limit on their degree) is infinite dimensional.

Polynomials provide other examples of vector spaces. A polynomial in the variable x is a linear combination of powers of x, such as $2 + 3x^4$, or 1, or $\frac{1}{3}(x-1)^2(x^3-3x)^6$. We could multiply out the last example to write it as a linear combination of powers of x. The degree of a polynomial is the highest power that it contains. The complicated product above has degree 20. One vector space is the set, P_d, of all polynomials of degree not more than d. This space has a basis consisting of $d+1$ elements:

$$f_0 = 1 \ , \ \ f_1 = x \ , \ \ \ldots \ , \ \ f_d = x^d \ .$$

Another basis of P_3 consists of the "Hermite" polynomials

$$H_0 = 1 \ , \ \ H_1 = x \ , \ \ H_2 = x^2 - 1 \ , \ \ H_3 = x^3 - 3x \ .$$

These are useful in probability because if x is a standard normal random variable, then they are uncorrelated:

$$E\left[H_j(X) H_k(X)\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} H_j(x) H_k(x) e^{-x^2/2} dx = 0 \quad \text{if } j \neq k.$$

Still another basis of P_3 consists of Lagrange interpolating polynomials for the points 1, 2, 3, and 4:

$$l_1 = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \ , \ \ l_2 = \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \ ,$$

[5] An infinite dimensional vector space might have an infinite basis, whatever that might mean.

$$l_3 = \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} \; , \quad l_4 = \frac{(x-1)(x-2)(x-3)}{(3-1)(3-2)(3-4)} \; .$$

These are useful for interpolation because, for example, $l_1(1) = 1$ while $l_2(1) = l_3(1) = l_4(1) = 0$. If we want $u(x)$ to be a polynomial of degree 3 taking specified values $u(1) = u_1$, $u(2) = u_2$, $u(3) = u_3$, and $u(4) = u_4$, the answer is

$$u(x) = u_1 l_1(x) + u_2 l_2(x) + u_3 l_3(x) + u_4 l_4(x) \; .$$

For example, at $x = 2$, the l_2 term takes the value u_2 while the other three terms are zero. The Lagrange interpolating polynomials are linearly independent because if $0 = u(x) = u_1 l_1(x) + u_2 l_2(x) + u_3 l_3(x) + u_4 l_4(x)$ for all x then in particular $u(x) = 0$ at $x = 1$, 2, 3, and 4, so $u_1 = u_2 = u_3 = u_4 = 0$.

If $V' \subset V$ is a subspace of dimension m of a vector space of dimension n, then it is possible to find a basis, f_k, of V so that the first m of the f_k form a basis of V'. For example, if $V = P_3$ and V' is the polynomials that vanish at $x = 2$ and $x = 3$, we can take

$$f_1 = (x-2)(x-3) \; , \quad f_2 = x(x-2)(x-3) \; , \quad f_3 = 1 \; , \quad f_4 = x \; .$$

Note the general (though not universal) rule that the dimension of V' is equal to the dimension of V minus the number of constraints or conditions that define V'. Whenever V' is a *proper* subspace of V, there is some $u \in V$ that is not in V', $m < n$. One common task in computational linear algebra is finding a *well conditioned* basis for a subspace specified in some way.

4.2.2 Matrices and linear transformations

Suppose V and W are vector spaces. A *linear transformation* from V to W is a function that produces an element of W for any element of V, written $w = Lv$, that is linear. Linear means that $L(v_1 + v_2) = Lv_1 + Lv_2$ for any two vectors in V, and $Lxv = xLv$ for any scalar x and vector $v \in V$. By convention we write Lv instead of $L(v)$ even though L represents a function from V to W. This makes algebraic manipulation with linear transformations look just like algebraic manipulation of matrices, deliberately blurring the distinction between linear transformations and matrix multiplication. The simplest example is the situation of $V = R^n$, $W = R^m$, and $Lu = A \cdot u$, for some $m \times n$ matrix A. The notation $A \cdot u$ means that we should multiply the vector u by the matrix A. Most of the time we leave out the dot.

Any linear transformation between finite dimensional vector spaces may be represented by a matrix. Suppose $f_1, \ldots, f_n$ is a basis for V, and $g_1, \ldots, g_m$ is a basis for W. For each k, the linear transformation of f_k is an element of W and may be written as a linear combination of the g_j:

$$Lf_k = \sum_{j=1}^{m} a_{jk} g_j \; .$$

Because the transformation is linear, we can calculate what happens to a vector $u \in V$ in terms of its expansion $u = \sum_k u_k f_k$. Let $w \in W$ be the *image* of u, $w = Lu$, written as $w = \sum_j w_j g_j$. We find

$$w_j = \sum_{k=1}^{n} a_{jk} u_k \ ,$$

which is ordinary matrix-vector multiplication.

The matrix that represents L depends on the basis. For example, suppose $V = P_3$, $W = P_2$, and L represents differentiation:

$$L\left(p_0 + p_1 x + p_2 x^2 + p_3 x^3\right) = \frac{d}{dx}\left(p_0 + p_1 x + p_2 x^2 + p_3 x^3\right) = p_1 + 2p_2 x + 3p_3 x^2 \ .$$

If we take the basis 1, x, x^2, x^3 for V, and 1, x, x^2 for W, then the matrix is

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

The matrix would be different if we used the Hermite polynomial basis for V. (See Exercise 1).

Conversely, an $m \times n$ matrix, A, represents a linear transformation from R^n to R^m (or from C^n to C^m). We denote this transformation also by A. If $v \in R^m$ is an m component column vector, then $w = Av$, the matrix vector product, determines $w \in R^n$, a column vector with n components. As before, the notation deliberately is ambiguous. The matrix A is the matrix that represents the linear transformation A using standard bases of R^n and R^m.

A matrix also may represent a change of basis within the same space V. If f_1, $\ldots$, f_n, and g_1, $\ldots$, g_n are different bases of V, and u is a vector with expansions $u = \sum_k v_k f_k$ and $u = \sum_j w_j g_j$, then we may write

$$\begin{pmatrix} v_1 \\ \cdot \\ \cdot \\ v_n \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \cdot & a_{nn} \end{pmatrix} \begin{pmatrix} w_1 \\ \cdot \\ \cdot \\ w_n \end{pmatrix}$$

As before, the matrix elements a_{jk} are the expansion coefficients of g_j with respect to the f_k basis[6]. For example, suppose $u \in P_3$ is given in terms of Hermite polynomials or simple powers: $u = \sum_{j=0}^{3} v_j H_j(x) = \sum_{k=0}^{3} w_j x^j$, then

$$\begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

[6]We write a_{jk} for the (j,k) entry of A.

We may reverse the change of basis by using the inverse matrix:

$$\begin{pmatrix} w_1 \\ \cdot \\ \cdot \\ w_n \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \cdot & a_{nn} \end{pmatrix}^{-1} \begin{pmatrix} v_1 \\ \cdot \\ \cdot \\ v_n \end{pmatrix}$$

Two bases must have the same number of elements because only square matrices can be invertible.

Composition of linear transformations corresponds to matrix multiplication. If L is a linear transformation from V to W, and M is a linear transformation from W to a third vector space, Z, then ML is the *composite* tranformation that takes V to[7] W. The composite of L and M is defined if the *target* (or *range*) of L, is the same as the *source* (or *domain*) of M, W in this case. If A is an $m \times n$ matrix and and B is $p \times q$, then the target of A is $W = R^n$ and the source of B is R^q. Therefore, the composite AB is defined if $n = p$. This is the condition for the matrix product $A \cdot B$ (usually written without the dot) to be defined. The result is a transformation from $V = R^m$ to $Z = R^p$, i.e., the $m \times p$ matrix AB.

For vector spaces V and W, the set of linear transformations from V to W forms a vector space. We can add two linear transformations and multiply a linear transformation by a scalar. This applies in particular to $m \times n$ matrices, which represent linear transformations from R^n to R^m. The entries of $A + B$ are $a_{jk} + b_{jk}$. An $n \times 1$ matrix has a single column and may be thought of as a column vector. The product Au is the same whether we consider u to be a column vector or an $n \times 1$ matrix. A $1 \times n$ matrix has a single row and may be thought of as a *row vector*. If r is such a row vector, the product rA makes sense, but Ar does not. It is useful to distinguish between row and column vectors although both are determined by n components. The product ru is a 1×1 matrix ($(1 \times n) \cdot (n \times 1)$ gives 1×1), i.e., a scalar.

If the source and targets are the same, $V = W$, or $n = m = p = q$, then both composites LM and ML both are defined, but they probably are not equal. Similarly, if A and B are $n \times n$ matrices, then AB and BA are defined, but $AB \neq BA$ in general. If A, B, and C are matrices so that the products AB and BC both are defined, then the products $A \cdot (BC)$ and $(AB) \cdot C$ also are defined. The *associative* property of matrix multiplication is the fact that these are equal: $A(BC) = (AB)C$. In actual computations it may be better to compute AB first then multiply by C than to compute BC first the compute $A(BC)$. For example suppose u is a $1 \times n$ row vector, and B and C are $n \times n$ matrices. Then $uB = v$ is the product of a row vector and a square matrix, which takes $O(n^2)$ additions and multiplications (*flops*) to compute, and similarly for $vC = (uB)C$. On the other hand, BC is the product of $n \times n$ matrices, which takes $O(n^3)$ flops to compute. Thus $(uB)C$ takes an order of magnitude fewer flops to compute than $u(BC)$.

[7] First v goes to $w = Lv$, then w goes to $z = Mw$. In the end, v has gone to $M(Lv)$.

If A is an $m \times n$ matrix with real entries a_{jk}, the *transpose* of A, written A^*, is the $n \times m$ matrix whose (j,k) entry is $a^*_{jk} = a_{kj}$. If A has complex entries, then A^*, the *adjoint* of A, is the $n \times m$ matrix wth entries $a^*_{jk} = \overline{a}_{kj}$ ($\overline{a}$ is the complex conjugate of a.). If the entries of A are real, the adjoint and transpose are the same. The transpose often is written A^t, but we use the same notation for adjoint and transpose. The transpose or adjoint of a column vector is a row vector with the same number of entries, and vice versa. By convention, the vector spaces R^n and C^n consist of n component column vectors. A real square ($n = m$) matrix is *symmetric* if $A = A^*$. A complex square matrix is *self-adjoint* if $A^* = A$. We often use the term "self-adjoint" for either case.

If u and v are n component column vectors ($u \in C^n$, $v \in C^n$), their *standard inner product* is

$$\langle u, v \rangle = u^* v = \sum_{k=1}^{n} \overline{u}_k v_k \ . \tag{4.2}$$

The inner product is *antilinear* in its first argument, which means that

$$\langle xu + yv, w \rangle = \overline{x} \langle u, w \rangle + \overline{y} \langle v, w \rangle \ ,$$

for complex numbers x and y. It is linear in the second argument:

$$\langle u, xv + yw \rangle = x \langle u, w \rangle y + \langle v, w \rangle \ .$$

It determines the standard *euclidean norm* $\|u\| = \langle u, u \rangle^{1/2}$. The complex conjugates are not needed when the entries of u and v are real. The reader should check that $\overline{\langle u, v \rangle} = \langle v, u \rangle$. The adjoint of A is determined by the requirement that $\langle A^* u, v \rangle = \langle u, Av \rangle$ for all u and v.

4.2.3 Vector norms

A *norm* is a way to describe the size of a vector. It is a single number extracted from the information describing u, which we write $\|u\|$ (read "the norm of u"). There are several different norms that are useful in scientific computing. We say $\|u\|$ is a norm if it has the following properties. First, $\|u\| \geq 0$, with $\|u\| = 0$ only when $u = 0$. Second, it should be *homogeneous*: $\|xu\| = |x| \, \|u\|$ for any scalar, x. Third, it should satisfy the *triangle inequality*, $\|u + v\| \leq \|u\| + \|v\|$, for any two vectors u and v.

There are several simple norms for R^n or C^n that have names. One is the l^1 norm

$$\|u\|_1 = \|u\|_{l^1} = \sum_{k=1}^{n} |u_k| \ .$$

Another is the l^∞ norm, also called the *sup* norm or the *max* norm:

$$\|u\|_\infty = \|u\|_{l^\infty} = \max_{k=1,\ldots,n} |u_k| \ .$$

Another is the l^2 norm, also called the *Euclidian* norm

$$\|u\|_2 = \|u\|_{l^2} = \left(\sum_{k=1}^{n} |u_k|^2 \right)^{1/2} = \langle u, u \rangle^{1/2} \; .$$

The l^2 norm is natural for vectors representing positions or velocities in three dimensional space. If the components of $u \in R^n$ represent probabilities, the l^1 norm might be more appropriate. In some cases we may have a norm defined indirectly or with a definition that is hard to turn into a number. For example in the vector space P_3 of polynomials of degree 3, we can define a norm

$$\|p\| = \max_{a \le x \le b} |p(x)| \; . \tag{4.3}$$

There is no simple formula for $\|p\|$ in terms of the coefficients of p.

The notion of vector norms is not perfect. For one thing, the choice of norm seems arbitrary in many cases . For example, what norm should we use for the two dimensional subspace of P_3 consisting of polynomials that vanish when $x = 2$ and $x = 3$? Another criticism is that norms may not make dimensional sense if the different components of u have different units. This might happen, for example, if the components of u represent different factors (or variables) in a linear regression. The first factor, u_1, might be age of a person, the second, u_2, income, the third the number of children. In some units, we might get (because the computer stores only numbers, not units)

$$u = \begin{pmatrix} 45 \\ 50000 \\ 2 \end{pmatrix} \tag{4.4}$$

In situations like these we can define for example, a *dimensionless* version of the l^1 norm:

$$\|u\| = \sum_{k=1}^{n} \frac{1}{\overline{u}_k} \cdot |u_k| \; ,$$

where $\overline{u}_k$ is a typical value of a quantity with the units of u_k in the problem at hand.

We can go further and use the basis

$$f_k = \overline{u}_k e_k \tag{4.5}$$

of R^n. In this basis, the components of u are $\widetilde{u}_k = \frac{1}{\overline{u}_k} u_k$, which are dimensionless. This is *balancing* or *diagonal scaling*. For example, if we take a typical age to be $\overline{u}_1 = 40$(years), a typical salary to be $\overline{u}_2 = 60,000$(dollars/year), and a typical number of children to be $\overline{u}_3 = 2.3$ (US national average), then the normalized components are comparable well scaled numbers: $(f_1, f_2, f_3) = (1.125, .833, .870)$. The matrix representing a linear transformation in the f_k basis is likely to be better conditioned than the one using the e_k basis.

4.2.4 Norms of matrices and linear transformations

Suppose L is a linear transformation from V to W. If we have norms for the spaces V and W, we can define a corresponding norm of L, written $\|L\|$, as the largest amount by which it stretches a vector:

$$\|L\| = \max_{u \neq 0} \frac{\|Lu\|}{\|u\|} \ . \tag{4.6}$$

The norm definition (4.6) implies that for all u,

$$\|Lu\| \leq \|L\| \cdot \|u\| \ . \tag{4.7}$$

Moreover, $\|L\|$ is the *sharp constant* in the inequality (4.7) in the sense that if $\|Lu\| \leq C \cdot \|u\|$ for all u, then $C \geq \|L\|$. Thus, (4.6) is equivalent to saying that $\|L\|$ is the sharp constant in (4.7).

The different vector norms give rise to different matrix norms. The matrix norms corresponding to certain standard vector norms are written with corresponding subscripts, such as

$$\|L\|_{l^2} = \max_{u \neq 0} \frac{\|Lu\|_{l^2}}{\|u\|_{l^2}} \ . \tag{4.8}$$

For $V = W = R^n$, it turns out that (for the linear transformation represented in the standard basis by A)

$$\|A\|_{l^1} = \max_k \sum_j |a_{jk}| \ ,$$

and

$$\|A\|_{l^\infty} = \max_j \sum_k |a_{jk}| \ .$$

Thus, the l^1 matrix norm is the maximum *column sum* while the max norm is the maximum *row sum*. Other norms are hard to compute explicitly in terms of the entries of A. People often say that $\|L\|_{l^2}$ is the largest *singular value* of L, but this is not too helpful because (4.8) is the definition of the largest singular value of L.

Any norm defined by (4.6) in terms of vector norms has several properties derived from corresponding properties of vector norms. One is homogeneity: $\|xL\| = |x| \, \|L\|$. Another is that $\|L\| \geq 0$ for all L, with $\|L\| = 0$ only for $L = 0$. The triangle inequality for vector norms implies that if L and M are two linear transformations from V to W, then $\|L + M\| \leq \|L\| + \|M\|$. Finally, we have $\|LM\| \leq \|L\| \, \|M\|$. This is because the composite transformation stretches no more than the product of the individual maximum stretches:

$$\|M(Lu)\| \leq \|M\| \, \|Lu\| \leq \|M\| \, \|L\| \, \|u\| \ .$$

Of course, all these properties hold for matrices of the appropriate sizes.

All of these norms have uses in the theoretical parts of scientific computing, the l^1 and l^∞ norms because they are easy to compute and the l^2 norm because it is invariant under orthogonal transformations such as the discrete Fourier transform. The norms are not terribly different from each other. For example, $\|A\|_{l^1} \leq n \|A\|_{l^\infty}$ and $\|A\|_{l^\infty} \leq n \|A\|_{l^1}$. For $n \leq 1000$, this factor of n may not be so important if we are asking, for example, about catastrophic ill-conditioning.

4.2.5 Eigenvalues and eigenvectors

Let A be an $n \times n$ matrix, or a linear transformation from V to V. We say that λ is an *eigenvalue*, and $r \neq 0$ the corresponding (right) *eigenvector* if

$$Ar = \lambda r \; .$$

Eigenvalues and eigenvectors are useful in understanding dynamics related to A. For example, the differential equation $\frac{du}{dt} = \dot{u} = Au$ has solutions $u(t) = e^{\lambda t} r$. Moreover, eigenvalues and their relatives, *singular values*, are the basis of *principal component analysis* in statistics. In general, eigenvalues and eigenvectors may be complex even though A is real. This is one reason people work with complex vectors in C^n, even for applications that seem to call for R^n.

Although their descriptions seem similar, the *symmetric eigenvalue problem* (A symmetric for real A or self-adjoint for complex A), is vastly different from the general, or *unsymmetric* problem. This contrast is detailed below. Here I want to emphasize the differences in conditioning. In some sense, the eigenvalues of a symmetric matrix are always well-conditioned functions of the matrix – a rare example of uniform good fortune. By contrast, the eigenvalues of an unsymmetric matrix may be so ill-conditioned, even for n as small as 20, that they are not computable (in double precision arithmetic) and essentially useless. Eigenvalues of unsymmetric matrices are too useful to ignore, but we can get into trouble if we ignore their potential ill-conditioning. Eigenvectors, even for symmetric matrices, may be ill-conditioned, but the consequences of this ill-conditioning seem less severe.

We begin with the unsymmetric eigenvalue problem. Nothing we say is wrong if A happens to be symmetric, but some of the statements might be misleading. An $n \times n$ matrix may have as many as n eigenvalues, denoted λ_k, $k = 1, \ldots, n$. If all the eigenvalues are distinct, the corresponding eigenvectors, denoted r_k, with $Ar_k = \lambda_k r_k$ must be linearly independent, and therefore form a basis for R^n. These n linearly independent vectors can be assembled to form the columns of an $n \times n$ eigenvector matrix that we call the *right eigenvector* matrix.

$$R = \begin{pmatrix} \vdots & & & & \vdots \\ r_1 & \cdot & \cdot & \cdot & r_n \\ \vdots & & & & \vdots \end{pmatrix} \; . \tag{4.9}$$

We also consider the diagonal eigenvalue matrix with the eigenvalues on the diagonal and zeros in all other entries:

$$\Lambda = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix} .$$

The eigenvalue – eigenvector relations may be expressed in terms of these matrices as

$$AR = R\Lambda . \tag{4.10}$$

To see this, check that multiplying R by A is the same as multiplying each of the columns of R by A. Since these are eigenvectors, we get

$$\begin{aligned} A \begin{pmatrix} \vdots & & & & \vdots \\ r_1 & \cdot & \cdot & \cdot & r_n \\ \vdots & & & & \vdots \end{pmatrix} &= \begin{pmatrix} \vdots & & & & \vdots \\ \lambda_1 r_1 & \cdot & \cdot & \cdot & \lambda_n r_n \\ \vdots & & & & \vdots \end{pmatrix} \\ &= \begin{pmatrix} \vdots & & & & \vdots \\ r_1 & \cdot & \cdot & \cdot & r_n \\ \vdots & & & & \vdots \end{pmatrix} \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix} \\ &= R\Lambda . \end{aligned}$$

Since the columns of R are linearly independent, R is invertible, we can multiply (4.10) from the right and from the left by R^{-1} to get

$$R^{-1}ARR^{-1} = R^{-1}R\Lambda R^{-1} ,$$

then cancel the $R^{-1}R$ and RR^{-1}, and define[8] $L = R^{-1}$ to get

$$LA = \Lambda L .$$

This shows that the k^{th} row of L is an eigenvector of A if we put the A on the right:

$$l_k A = \lambda_k l_k .$$

Of course, the λ_k are the same: there is no difference between "right" and "left" eigenvalues. The matrix equation we used to define L, $LR = I$, gives useful relations between left and right eigenvectors. The (j, k) entry of LR is the product of row j of L with row k of R. When $j = k$ this product should be a diagonal entry of I, namely one. When $j \neq k$, the product should be zero. That is

$$\left.\begin{aligned} l_k r_k &= 1 \\ l_j r_k &= 0 \qquad \text{if } j \neq k. \end{aligned}\right\} \tag{4.11}$$

[8]Here L refers to a matrix, not a general linear transformation.

These are called *biorthogonality* relations. For example, r_1 need not be orthogonal to r_2, but it is orthogonal to l_2. The set of vectors r_k is not orthogonal, but the two sets l_j and r_k are biorthogonal. The left eigenvectors are sometimes called *adjoint eigenvectors* because their transposes form right eigenvectors for the adjoint of A:

$$A^* l_j^* = \lambda_j l_j^* \; .$$

Still supposing n distinct eigenvalues, we may take the right eigenvectors to be a basis for R^n (or C^n if the entries are not real). As discussed in Section 2.2, we may express the action of A in this basis. Since $Ar_j = \lambda_j r_j$, the matrix representing the linear transformation A in this basis will be the diagonal matrix Λ. In the framework of Section 2.2, this is because if we expand a vector $v \in R^n$ in the r_k basis, $v = v_1 r_1 + \cdots + v_n r_n$, then $Av = \lambda_1 v_1 r_1 + \cdots + \lambda_n v_n r_n$. For this reason finding a complete set of eigenvectors and eigenvalues is called *diagonalizing* A. A matrix with n linearly independent right eigenvectors is *diagonalizable.*

If A does not have n distinct eigenvalues then there may be no basis in which the action of A is diagonal. For example, consider the matrix

$$J = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \; .$$

Clearly, $J \neq 0$ but $J^2 = 0$. A diagonal or diagonalizable matrix cannot have this property: if $\Lambda^2 = 0$ then $\Lambda = 0$, and if the relations $A \neq 0$, $A^2 = 0$ in one basis, they hold in any other basis. In general a *Jordan block* with eigenvalue λ is a $k \times k$ matrix with λ on the diagonal, 1 on the *superdiagonal* and zeros elsewhere:

$$\begin{pmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \lambda & 1 \\ 0 & 0 & \cdots & 0 & \lambda \end{pmatrix} \; .$$

If a matrix has fewer than n distinct eigenvalues, it might or might not be diagonalizable. If A is not diagonalizable, a theorem of linear algebra states that there is a basis in which A has *Jordan form.* A matrix has Jordan form if it is *block diagonal* with Jordan blocks of various sizes and eigenvalues on the diagonal. It can be difficult to compute the Jordan form of a matrix numerically, as we will see.

The eigenvalue – eigenvector problem for symmetric or self-adjoint matrices is different and in many ways simpler than the general nonsymmetric eigenvalue problem. The eigenvalues are real. The left eigenvectors are transposes of the right eigenvectors: $l_k = r_k^*$. There are no Jordan blocks; every symmetric matrix is diagonalizable even if the number of distinct eigenvalues is less than n. The biorthogonality relations (4.11) become the normalization conditions

$$r_j^* r_j = 1 \; , \tag{4.12}$$

and the orthogonality relations

$$r_j^* r_k = 0 \qquad (4.13)$$

A set of vectors satisfying both (4.12) and (4.13) is called *orthonormal*. A complete set of eigenvectors of a symmetric matrix forms an orthonormal basis. It is easy to check that the orthonormality relations are equivalent to the matrix R in (4.9) satisfying $R^* R = I$, or, equivalently, $R^{-1} = R^*$. Such a matrix is called *orthogonal*. The matrix form of the eigenvalue relation (4.10) may be written $R^* A R = \Lambda$, or $A = R \Lambda R^*$, or $R^* A = \Lambda R^*$. The latter shows (yet again) that the rows of R^*, which are r_k^*, are left eigenvectors of A.

4.2.6 Differentiation and perturbation theory

The main technique in the perturbation theory of Section 4.3 is implicit differentiation. We use the formalism of *virtual perturbations* from mechanical engineering, which is related to tangent vectors in differential geometry. It may seem roundabout at first, but it makes actual calculations quick.

Suppose $f(x)$ represents m functions, $f_1(x), \ldots, f_m(x)$ of n variables, $x_1, \ldots, x_n$. The Jacobian matrix[9], $f'(x)$, is the $m \times n$ matrix of partial derivatives $f'_{jk}(x) = \partial_{x_k} f_j(x)$. If f is differentiable (and f' is Lipschitz continuous), then the first derivative approximation is (writing x_0 for x to clarify some discussion below)

$$f(x_0 + \Delta x) - f(x_0) = \Delta f = f'(x_0)\Delta x + O\left(\|\Delta x\|^2\right) . \qquad (4.14)$$

Here Δf and Δx are column vectors.

Suppose s is a scalar "parameter" and $x(s)$ is a differentiable curve in R^n with $x(0) = x_0$. The function $f(x)$ then defines a curve in R^m with $f(x(0)) = f(x_0)$. We define two vectors, called *virtual perturbations*,

$$\dot{x} = \frac{dx(s)}{ds}(0) \; , \quad \dot{f} = \frac{df(x(s))}{ds}(0) \; .$$

The multivariate calculus chain rule implies the virtual perturbation formula

$$\dot{f} = f'(x_0)\dot{x} \; . \qquad (4.15)$$

This formula is nearly the same as (4.14). The virtual perturbation strategy is to calculate the linear relationship (4.15) between virtual perturbations and use it to find the approximate relation (4.14) between actual perturbations. For this, it is important that any $\dot{x} \in R^n$ can be the virtual perturbation corresponding to some curve: just take the straight "curve" $x(s) = x_0 + s\dot{x}$.

The Leibnitz rule (product rule) for matrix multiplication makes virtual perturbations handy in linear algebra. Suppose $A(s)$ and $B(s)$ are differentiable

[9]See any good book on multivariate calculus.

curves of matrices and compatible for matrix multiplication. Then the virtual perturbation of AB is given by the product rule

$$\left.\frac{d}{ds}AB\right|_{s=0} = \dot{A}B + A\dot{B} \; . \qquad (4.16)$$

To see this, note that the (jk) entry of $A(s)B(s)$ is $\sum_l a_{jl}(s)b_{lk}(s)$. Differentiating this using the ordinary product rule then setting $s = 0$ yields

$$\sum_l \dot{a}_{jl}b_{lk} + \sum_l a_{jl}\dot{b}_{lk} \; .$$

These terms correspond to the terms on the right side of (4.16). We can differentiate products of more than two matrices in this way.

As an example, consider perturbations of the inverse of a matrix, $B = A^{-1}$. The variable x in (4.14) now is the matrix A, and $f(A) = A^{-1}$. Apply implicit differentiation to the formula $AB = I$, use the fact that I is constant, and we get $\dot{A}B + A\dot{B} = 0$. Then solve for $\dot{B}$ (and use $A^{-1} = B$, and get

$$\dot{B} = -A^{-1}\dot{A}A^{-1} \; .$$

The corresponding actual perturbation formula is

$$\Delta\left(A^{-1}\right) = -A^{-1}\;\Delta A\;A^{-1} + O\left(\|\Delta A\|^2\right) \; . \qquad (4.17)$$

This is a generalization of the fact that the derivative of $1/x$ is $-1/x^2$, so $\Delta(1/x) \approx -1/x^2\Delta x$. When x is replaced by A and ΔA does not commute with A, we have to worry about the order of the factors. The correct order is (4.17).

For future reference we comment on the case $m = 1$, which is the case of one function of n variables. The $1 \times n$ Jacobian matrix may be thought of as a row vector. We often write this as $\bigtriangledown f$, and calculate it from the fact that $\dot{f} = \bigtriangledown f(x) \cdot \dot{x}$ for all $\dot{x}$. In particular, x is a stationary point of f if $\bigtriangledown f(x) = 0$, which is the same as $\dot{f} = 0$ for all $\dot{x}$. For example, suppose $f(x) = x^*Ax$ for some $n \times n$ matrix A. This is a product of the $1 \times n$ matrix x^* with A with the $n \times 1$ matrix x. The Leibnitz rule (4.16) gives, if A is constant,

$$\dot{f} = \dot{x}^*Ax + x^*A\dot{x} \; .$$

Since the 1×1 real matrix $\dot{x}^*Ax$ is equal to its transpose, this is

$$\dot{f} = x^*(A + A^*)\dot{x} \; .$$

This implies that (both sides are row vectors)

$$\bigtriangledown\;(\;x^*Ax\;) = x^*(A + A^*) \; . \qquad (4.18)$$

If $A^* = A$, we recognize this as a generalization of $n = 1$ formula $\frac{d}{dx}(ax^2) = 2ax$.

4.2.7 Variational principles for the symmetric eigenvalue problem

A *variational principle* is a way to find something by solving a maximization or minimization problem. The *Rayleigh quotient* for an $n \times n$ matrix is

$$Q(x) = \frac{x^* A x}{x^* x} = \frac{\langle x, Ax \rangle}{\langle x, x \rangle} \ . \tag{4.19}$$

If x is real, x^* is the transpose of x, which is a row vector. If x is complex, x^* is the adjoint. In either case, the denominator is $x^* x = \sum_{k=1}^n |x_k|^2 = \|x\|_{l^2}^2$. The Rayleigh quotient is defined for $x \neq 0$. A vector r is a *stationary point* if $\bigtriangledown Q(r) = 0$. If r is a stationary point, the corresponding value $\lambda = Q(r)$ is a *stationary value.*

Theorem 1 *If A is a real symmetric or a complex self-adjoint matrix, each eigenvector of A is a stationary point of the Rayleigh quotient and the corresponding eigenvalue is the corresponding stationary value. Conversely, each stationary point of the Rayleigh quotient is an eigenvector and the corresponding stationary value the corresponding eigenvalue.*

Proof: We give the proof for real x and real symmetric A. The complex self-adjoint case is an exercise. Underlying the theorem is the calculation (4.18) that if $A^* = A$ (this is where symmetry matters) then $\bigtriangledown \ (\ x^* A x \ ) = 2x^* A$. With this we calculate (using the quotient rule and (4.18) with $A = I$)

$$\bigtriangledown Q = 2 \left(\frac{1}{x^* x} \right) x^* A - 2 \left(\frac{x^* A x}{(x^* x)^2} \right) x^* \ .$$

If x is a stationary point ($\bigtriangledown Q = 0$), then $x^* A = \left(\frac{x^* A x}{x^* x} \right) x^*$, or, taking the transpose,

$$Ax = \left(\frac{x^* A x}{x^* x} \right) x \ .$$

This shows that x is an eigenvector with

$$\lambda = \frac{x^* A x}{x^* x} = Q(x)$$

as the corresponding eigenvalue. Conversely if $Ar = \lambda r$, then $Q(r) = \lambda$ and the calculations above show that $\bigtriangledown Q(r) = 0$. This proves the theorem.

A simple observation shows that there is at least one stationary point of Q for Theorem 1 to apply to. If α is a real number, then $Q(\alpha x) = Q(x)$. We may choose α so that[10] $\|\alpha x\| = 1$. This shows that

$$\max_{x \neq 0} Q(x) = \max_{\|x\|=1} Q(x) = \max_{\|x\|=1} x^* A x \ .$$

[10] In this section and the next, $\|x\| = \|x\|_{l^2}$.

A theorem of analysis states that if $Q(x)$ is a continuous function on a compact set, then there is an r so that $Q(r) = \max_x Q(x)$ (the max is attained). The set of x with $\|x\| = 1$ (the *unit sphere*) is compact and Q is continuous. Clearly if $Q(r) = \max_x Q(x)$, then $\bigtriangledown Q(r) = 0$, so r is a stationary point of Q and an eigenvector of A.

The key to finding the other $n-1$ eigenvectors is a simple orthogonality relation. The principle also is the basis of the *singular value decomposition.* If r is an eigenvector of A and x is orthogonal to r, then x also is orthogonal to Ar. Since $Ar = \lambda r$, and x is orthogonal to r if $x^* r = 0$, this implies $x^* Ar = x^* \lambda r = 0$. More generally, suppose we have $m < n$ and eigenvectors $r_1, \ldots, r_m$. eigenvectors $r_1, \ldots, r_m$ with $m < n$. Let $V_m \subseteq R^n$ be the set of $x \in R^n$ with $x^* r_j = 0$ for $j = 1, \ldots, m$. It is easy to check that this is a subspace of R^n. The orthogonality principle is that if $x \in V_m$ then $Ax \in V_m$. That is, if $x^* r_j = 0$ for all j, then $(Ax)^* r_j = 0$ for all j. But $(Ax)^* r_j = x^* A^* r_j = x^* A r_j = x^* \lambda_j r_j = 0$ as before.

The variational and orthogonality principles allow us to find $n-1$ additional orthonormal eigenvectors as follows. Start with r_1, a vector that maximizes Q. Let V_1 be the vector space of all $x \in R^n$ with $r_1^* x = 0$. We just saw that V_1 is an *invariant subspace* for A, which means that $Ax \in V_1$ whenever $x \in V_1$. Thus A defines a linear transformation from V_1 to V_1, which we call A_1. Chapter 5.1 gives a proof that A_1 is symmetric in a suitable basis. Therefore, Theorem 1 implies that A_1 has at least one real eigenvector, r_2, with real eigenvalue λ_2. Since $r_2 \in V_1$, the action of A and A_1 on r_2 is the same, which means that $Ar_2 = \lambda_2 r_2$. Also since $r_2 \in V_1$, r_2 is orthogonal to r_1. Now let $V_2 \subset V_1$ be the set of $x \in V_1$ with $x^* r_2 = 0$. Since $x \in V_2$ means $x^* r_1 = 0$ and $x^* r_2 = 0$, V_2 is an invariant subspace. Thus, there is an $r_3 \in V_2$ with $Ar_3 = A_1 r_3 = A_2 r_3 = \lambda_3 r_3$. And again r_3 is orthogonal to r_2 because $r_3 \in V_2$ and r_3 is orthogonal to r_1 because $r_3 \in V_2 \subset V_1$. Continuing in this way, we eventually find a full set of n orthogonal eigenvectors.

4.2.8 Least squares

Suppose A is an $m \times n$ matrix representing a linear transformation from R^n to R^m, and we have a vector $b \in R^m$. If $n < m$ the linear equation system $Ax = b$ is *overdetermined* in the sense that there are more equations than variables to satisfy them. If there is no x with $Ax = b$, we may seek an x that minimizes the *residual*

$$r = Ax - b \, . \tag{4.20}$$

This *linear least squares* problem

$$\min_x \|Ax - b\|_{l^2} \ , \tag{4.21}$$

is the same as finding x to minimize the sum of squares

$$SS = \sum_{j=1}^{n} r_j^2 \, .$$

Linear least squares problems arise through *linear regression* in statistics. A *linear regression model* models the response, b, as a linear combination of m *explanatory* vectors, a_k, each weighted by a *regression coefficient*, x_k. The residual, $R = (\sum_{k=1}^m a_k x_k) - b$, is modeled as a Gaussian random variable[11] with mean zero and variance σ^2. We do n *experiments*. The explanatory variables and response for experiment j are a_{jk}, for $k = 1.\ldots, m$, and b_j, and the residual (for given regression coefficients) is $r_j = \sum_{k=1}^m a_{jk} x_k - b_j$. The *log likelihood* function is (r depends on x through (4.20) $f(x) = -\sigma^2 \sum_{j=1}^n r_j^2$. Finding regression coefficients to maximize the log likelihood function leads to (4.21).

The *normal equations* give one approach to least squares problems. We calculate:

$$\begin{aligned} \|r\|_{l^2}^2 &= r^* r \\ &= (Ax - b)^* (Ax - b) \\ &= x^* A^* A x - 2x^* A^* b + b^* b \ . \end{aligned}$$

Setting the gradient to zero as in the proof of Theorem 1 leads to the normal equations

$$A^* A x = A^* b \ , \tag{4.22}$$

which can be solved by

$$x = (A^* A)^{-1} A^* b \ . \tag{4.23}$$

The matrix $M = A^* A$ is the *moment matrix* or the *Gram matrix*. It is symmetric, and *positive definite* if A has rank m, so the *Choleski decomposition* of M (see Chapter 5.1) is a good way to solve (4.22). The matrix $(A^* A)^{-1} A^*$ is the *pseudoinverse* of A. If A were square and invertible, it would be A^{-1} (check this). The normal equation approach is the fastest way to solve dense linear least squares problems, but it is not suitable for the subtle ill-conditioned problems that arise often in practice.

The singular value decomposition in Section 4.2.9 and the QR decomposition from Section 5.4 give better ways to solve ill-conditioned linear least squares problems.

4.2.9 Singular values and principal components

Eigenvalues and eigenvectors of a symmetric matrix have at many applications. They can be used to solve dynamical problems involving A. Because the eigenvectors are orthogonal, they also determine the l^2 norm and condition number of A. Eigenvalues and eigenvectors of a non-symmetric matrix are not orthogonal so the eigenvalues do not determine the norm or condition number. *Singular values* for non-symmetric or even non-square matrices are a partial substitute.

Let A be an $m \times n$ matrix that represents a linear transformation from R^n to R^m. The *right singular vectors*, $v_k \in R^n$ form an orthonormal basis for

[11] See any good book on statistics for definitions of Gaussian random variable and the log likelihood function. What is important here is that a systematic statistical procedure, the *maximum likelihood* method, tells us to minimize the sum of squares of residuals.

R^n. The *left singular vectors*, $u_k \in R^m$, form an orthonormal basis for R^m. Corresponding to each v_k and u_k pair is a non-negative *singular value*, σ_k with

$$Av_k = \sigma_k u_k \; . \tag{4.24}$$

By convention these are ordered so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. If $n < m$ we interpret (4.24) as saying that $\sigma_k = 0$ for $k > n$. If $n > m$ we say $\sigma_k = 0$ for $k > m$.

The non-zero singular values and corresponding singular vectors may be found one by one using variational and orthogonality principles similar to those in Section 4.2.7. We suppose A is not the zero matrix (not all entries equal to zero). The first step is the variational principle:

$$\sigma_1 = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} \; . \tag{4.25}$$

As in Section 4.2.7, the maximum is achieved, and $\sigma_1 > 0$. Let $v_1 \in R^n$ be a maximizer, normalized to have $\|v_1\| = 1$. Because $\|Av_1\| = \sigma_1$, we may write $Av_1 = \sigma_1 u_1$ with $\|u_1\| = 1$. This is the relation (4.24) with $k = 1$.

The optimality condition calculated as in the proof of Theorem 1 implies that

$$u_1^* A = \sigma_1 v_1^* \; . \tag{4.26}$$

Indeed, since $\sigma_1 > 0$, (4.25) is equivalent to[12]

$$\begin{aligned} \sigma_1^2 &= \max_{x \neq 0} \frac{\|Ax\|^2}{\|x\|^2} \\ &= \max_{x \neq 0} \frac{(Ax)^*(Ax)}{x^* x} \\ \sigma_1^2 &= \max_{x \neq 0} \frac{x^*(A^*A)x}{x^* x} \; . \end{aligned} \tag{4.27}$$

Theorem 1 implies that the solution to the maximization problem (4.27), which is v_1, satisfies $\sigma^2 v_1 = A^* A v_1$. Since $Av_1 = \sigma u_1$, this implies $\sigma_1 v_1 = A^* u_1$, which is the same as (4.26).

The analogue of the eigenvalue orthogonality principle is that if $x^* v_1 = 0$, then $(Ax)^* u_1 = 0$. This is true because

$$(Ax)^* u_1 = x^* (A^* u_1) = x^* \sigma_1 v_1 = 0 \; .$$

Therefore, if we define $V_1 \subset R^n$ by $x \in V_1$ if $x^* v_1 = 0$, and $U_1 \subset R^m$ by $y \in U_1$ if $y^* u_1 = 0$, then A also defines a linear transformation (called A_1) from V_1 to U_1. If A_1 is not identically zero, we can define

$$\sigma_2 = \max_{\substack{x \in V_1 \\ x \neq 0}} \frac{\|Ax\|^2}{\|x\|^2} = \max_{\substack{x^* v_1 = 0 \\ x \neq 0}} \frac{\|Ax\|^2}{\|x\|^2} \; ,$$

[12] These calculations make constant use of the associativity of matrix multiplication, even when one of the matrices is a row or column vector.

and get $Av_2 = \sigma_2 u_2$ with $v_2^* v_1 = 0$ and $u_2^* u_1 = 0$. This is the second step constructing orthonormal bases satisfying (4.24). Continuing in this way, we can continue finding orthonormal vectors v_k and u_k that satisfy (4.24) until reach $A_k = 0$ or $k = m$ or $k = n$. After that point, the may complete the v and u bases arbitrarily as in Chapter 5.1 with remaining singular values being zero.

The *singular value decomposition* (*SVD*) is a matrix expression of the relations (4.24). Let U be the $m \times m$ matrix whose columns are the left singular vectors u_k (as in (4.9)). The orthonormality relations $u_j^* u_k = \delta_{jk}$ are equivalent to U being an orthogonal matrix: $U^* U = I$. Similarly, we can form the orthogonal $n \times n$ matrix, V, whose columns are the right singular vectors v_k. Finally, the $m \times n$ matrix, Σ, has the singular values on its diagonal (with somewhat unfortunate notation), $\sigma_{jj} = \sigma_j$, and zeros off the diagonal, $\sigma_{jk} = 0$ if $j \neq k$. With these definitions, the relations (4.24) are equivalent to $AV = U\Sigma$, which more often is written

$$A = U \Sigma V^* \ . \tag{4.28}$$

This the singular value decomposition. Any matrix may be factored, or decomposed, into the product of the form (4.28) where U is an $m \times m$ orthogonal matrix, Σ is an $m \times n$ diagonal matrix with nonnegative entries, and V is an $n \times n$ orthogonal matrix.

A calculation shows that $A^* A = V \Sigma^* \Sigma V^* = V \Lambda V^*$. This implies that the eigenvalues of $A^* A$ are given by $\lambda_j = \sigma_j^2$ and the right singular vectors of A are the eigenvectors of $A^* A$. It also implies that $\kappa_{l^2}(A^* A) = \kappa_{l^2}(A)^2$. This means that the condition number of solving the normal equations (4.22) is the square of the condition number of the original least squares problem (4.21). If the condition number of a least squares problem is $\kappa_{l^2}(A) = 10^5$, even the best solution algorithm can amplify errors by a factor of 10^5. Solving using the normal equations can amplify rounding errors by a factor of 10^{10}.

Many people call singular vectors u_k and v_k *principal components*. They refer to the singular value decomposition as *principal component analysis*, or *PCA*. One application is *clustering*, in which you have n objects, each with m measurements, and you want to separate them into two clusters, say "girls" and "boys". You assemble the data into a matrix, A, and compute, say, the largest two singular values and corresponding left singular vectors, $u_1 \in R^m$ and $u_2 \in R^m$. The data for object k is $a_k \in R^m$, and you compute $z_k \in R^2$ by $z_{k1} = u_1^* a_k$ and $z_{k2} = u_2^* a_k$, the components of a_k in the principal component directions. You then plot the n points z_k in the plane and look for clusters, or maybe just a line that separates one group of points from another. Surprising as may seem, this simple procedure does identify clusters in practical problems.

4.3 Condition number

Ill-conditioning can be a serious problem in numerical solution of problems in linear algebra. We take into account possible ill-conditioning when we choose computational strategies. For example, the matrix exponential $\exp(A)$ (see Exercise 12) can be computed using the eigenvectors and eigenvalues of A. We will see in Section 4.3.3 that the eigenvalue problem may be ill conditioned even when the problem of computing $\exp(A)$ is fine. In such cases we need a way to compute $\exp(A)$ that does not use the eigenvectors and eigenvalues of A.

As we said in Section 2.7 (in slightly different notation), the condition number is the ratio of the change in the answer to the change in the problem data, with (*i*) both changed measured in relative terms, and (*ii*) the change in the problem data being small. Norms allow us to make this definition more precise in the case of multivariate functions and data. Let $f(x)$ represent m functions of n variables, with Δx being a change in x and $\Delta f = f(x+\Delta x) - f(x)$ the corresponding change in f. The size of Δx, relative to x is $\|\Delta x\| \,/\, \|x\|$, and similarly for Δf. In the multivariate case, the size of Δf depends not only on the size of Δx, but also on the direction. The norm based condition number seeks the worst case Δx, which leads to

$$\kappa(x) = \lim_{\epsilon \to 0} \max_{\|\Delta x\| = \epsilon} \frac{\frac{\|f(x+\Delta x) - f(x)\|}{\|f(x)\|}}{\frac{\|\Delta x\|}{\|x\|}} \quad . \tag{4.29}$$

Except for the maximization over directions Δx with $\|\Delta x\| = \epsilon$, this is the same as the earlier definition 2.8.

Still following Section 2.7, we express (4.29) in terms of derivatives of f. We let $f'(x)$ represent the $m \times n$ *Jacobian* matrix of first partial derivatives of f, as in Section 4.2.6, so that, $\Delta f = f'(x)\Delta x + O\left(\|\Delta x\|^2\right)$. Since $O\left(\|\Delta x\|^2\right) / \|\Delta x\| = O\left(\|\Delta x\|\right)$, the ratio in (4.29) may be written

$$\frac{\|\Delta f\|}{\|\Delta x\|} \cdot \frac{\|x\|}{\|f\|} = \frac{\|f'(x)\Delta x\|}{\|\Delta x\|} \cdot \frac{\|x\|}{\|f\|} + O\left(\|\Delta x\|\right) \quad .$$

The second term on the right disappears as $\Delta x \to 0$. Maximizing the first term on the right over Δx yields the norm of the matrix $f'(x)$. Altogether, we have

$$\kappa(x) = \|f'(x)\| \cdot \frac{\|x\|}{\|f(x)\|} \quad . \tag{4.30}$$

This differs from the earlier condition number definition (2.10) in that it uses norms and maximizes over Δx with $\|\Delta x\| = \epsilon$.

In specific calculations we often use a slightly more complicated way of stating the definition (4.29). Suppose that P and Q are two positive quantities and there is a C so that $P \leq C \cdot Q$. We say that C is the *sharp* constant if there is no C' with $C' < C$ so that $P \leq C' \cdot Q$. For example, we have the inequality $\sin(2\epsilon) \leq 3 \cdot \epsilon$ for all x. But $C = 3$ is not the sharp constant because the inequality also is true with $C' = 2$, which is sharp. But this definition is clumsy, for example, if we ask for the sharp constant in the inequality $\tan(\epsilon) \leq C \cdot \epsilon$. For one thing, we must restrict to $\epsilon \to 0$. The sharp constant should be $C = 1$ because if $C' > 1$, there is an ϵ_0 so that if $\epsilon \leq \epsilon_0$, then $\tan(\epsilon) \leq C' \cdot \epsilon$. But the inequality is not true with $C = 1$ for any ϵ. Therefore we write

$$P(\epsilon) \underset{\sim}{\overset{\leq}{}} CQ(\epsilon) \quad \text{as } \epsilon \to 0 \tag{4.31}$$

if

$$\lim_{\epsilon \to 0} \frac{P(\epsilon)}{Q(\epsilon)} \leq C \; .$$

In particular, we can seek the sharp constant, the smallest C so that

$$P(\epsilon) \leq C \cdot Q(\epsilon) + O(\epsilon) \quad \text{as } \epsilon \to 0.$$

The definition (**??**) is precisely that $\kappa(x)$ is the sharp constant in the inequality

$$\frac{\|\Delta f\|}{\|f\|} \lesssim \frac{\|\Delta x\|}{\|x\|} \quad \text{as } \|x\| \to 0. \tag{4.32}$$

4.3.1 Linear systems, direct estimates

We start with the condition number of calculating $b = Au$ in terms of u with A fixed. This fits into the general framework above, with u playing the role of x, and Au of $f(x)$. Of course, A is the Jacobian of the function $u \to Au$, so (4.30) gives

$$\kappa(A, u) = \|A\| \cdot \frac{\|u\|}{\|Au\|} \quad . \tag{4.33}$$

The condition number of solving a linear system $Au = b$ (finding u as a function of b) is the same as the condition number of the computation $u = A^{-1}b$. The formula (4.33) gives this as

$$\kappa(A^{-1}, b) = \left\|A^{-1}\right\| \cdot \frac{\|b\|}{\|A^{-1}b\|} = \|A{-}1\| \cdot \frac{\|Au\|}{\|u\|} \quad .$$

For future reference, not that this is not the same as (4.33).

The traditional definition of the condition number of the Au computation takes the worst case relative error not only over perturbations Δu but also over vectors u. Taking the maximum over Δu led to (4.33), so we need only maximize it over u:

$$\kappa(A) = \|A\| \cdot \max_{u \neq 0} \frac{\|u\|}{\|Au\|} \quad . \tag{4.34}$$

Since $A(u + \Delta u) - Au = A\Delta u$, and u and Δu are independent variables, this is the same as

$$\kappa(A) = \max_{u \neq 0} \frac{\|u\|}{\|Au\|} \cdot \max_{\Delta u \neq 0} \frac{\|A\Delta u\|}{\|\Delta u\|} \quad . \tag{4.35}$$

To evaluate the maximum, we suppose A^{-1} exists.[13] Substituting $Au = v$, $u = A^{-1}v$, gives

$$\max_{u \neq 0} \frac{\|u\|}{\|Au\|} = \max_{v \neq 0} \frac{\left\|A^{-1}v\right\|}{\|v\|} = \left\|A^{-1}\right\| \; .$$

[13] See exercise 8 for a the l^2 condition number of the $u \to Au$ problem with singular or non-square A.

Thus, (4.34) leads to

$$\kappa(A) = \|A\| \left\|A^{-1}\right\| \tag{4.36}$$

as the worst case condition number of the forward problem.

The computation $b = Au$ with

$$A = \begin{pmatrix} 1000 & 0 \\ 0 & 10 \end{pmatrix}$$

illustrates this discussion. The error amplification relates $\|\Delta b\| \,/\, \|b\|$ to $\|\Delta u\| \,/\, \|u\|$. The worst case would make $\|b\|$ small relative to $\|u\|$ and $\|\Delta b\|$ large relative to $\|\Delta u\|$: amplify u the least and Δu the most. This is achieved by taking $u = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ so that $Au = \begin{pmatrix} 0 \\ 10 \end{pmatrix}$ with amplification factor 10, and $\Delta u = \begin{pmatrix} \epsilon \\ 0 \end{pmatrix}$ with $A\Delta u = \begin{pmatrix} 1000\epsilon \\ 0 \end{pmatrix}$ and amplification factor 1000. This makes $\|\Delta b\| \,/\, \|b\|$ 100 times larger than $\|\Delta u\| \,/\, \|u\|$. For the condition number of calculating $u = A^{-1}b$, the worst case is $b = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\Delta b = \begin{pmatrix} \epsilon \\ 0 \end{pmatrix}$, which amplifies the error by the same factor of $\kappa(A) = 100$.

The informal condition number (4.34) has advantages and disadvantages over the more formal one (4.33). At the time we design a computational strategy, it may be easier to estimate the informal condition number than the formal one, as we may know more about A than u. If we have no idea what u will come up, we have a reasonable chance of getting something like the worst one. Moreover, $\kappa(A)$ defined by (4.34) determines the convergence rate of iterative strategies for solving linear systems involving A. On the other hand, there are cases, particularly when solving partial differential equations, where $\kappa(A)$ is much more pessimistic than $\kappa(A, u)$. For example, in Exercise 11, $\kappa(A)$ is on the order of n^2, where n is the number of unknowns. The truncation error for the second order discretization is on the order of $1/n^2$. A naive estimate using (4.34) might suggest that solving the system amplifies the $O(n^{-2})$ truncation error by a factor of n^2 to be on the same order as the solution itself. This does not happen because the u we seek is smooth, and not like the worst case.

The informal condition number (4.36) also has the strange feature than $\kappa(A) = \kappa(A^{-1})$, since $\left(A^{-1}\right)^{-1} = A$. For one thing, this is not true, even using informal definitions, for nonlinear problems. Moreover, it is untrue in important linear problems, such as the heat equation.[14] Computing the solution at time $t > 0$ from “initial data” at time zero is a linear process that is well conditioned and numerically stable. Computing the solution at time zero from the solution at time $t > 0$ is so ill conditioned as to be essentially impossible. Again, the more precise definition (4.33) does not have this drawback.

[14] See, for example, the book by Fritz John on partial differential equations.

4.3.2 Linear systems, perturbation theory

If $Au = b$, we can study the dependence of u on A through perturbation theory. The starting point is the perturbation formula (4.17). Taking norms gives

$$\|\Delta u\| \lessapprox \left\|A^{-1}\right\| \|\Delta A\| \|u\| \ , \qquad \text{(for small } \Delta A\text{)}, \tag{4.37}$$

so

$$\frac{\|\Delta u\|}{\|u\|} \lessapprox \left\|A^{-1}\right\| \|A\| \cdot \frac{\|\Delta A\|}{\|A\|} \tag{4.38}$$

This shows that the condition number satisfies $\kappa \leq \left\|A^{-1}\right\| \|A\|$. The condition number is equal to $\left\|A^{-1}\right\| \|A\|$ if the inequality (4.37) is (approximately for small ΔA) sharp, which it is because we can take $\Delta A = \epsilon I$ and u to be a maximum stretch vector for A^{-1}.

We summarize with a few remarks. First, the condition number formula (4.36) applies to the problem of solving the linear system $Au = b$ both when we consider perturbations in b and in A, though the derivations here are different. However, this is not the condition number in the strict sense of (4.29) and (4.30) because the formula (4.36) assumes taking the worst case over a family of problems (varying b in this case), not just over all possible small perturbations in the data. Second, the formula (4.36) is independent of the size of A. Replacing A with cA leaves $\kappa(A)$ unchanged. What matters is the ratio of the maximum to minimum stretch, as in (4.35).

4.3.3 Eigenvalues and eigenvectors

The eigenvalue relationship is $Ar_j = \lambda_j r_j$. Perturbation theory allows to estimate the changes in λ_j and r_j that result from a small ΔA. These perturbation results are used throughout science and engineering. We begin with the symmetric or self-adjoint case, it often is called *Rayleigh Schödinger* perturbation theory[15] Using the virtual perturbation method of Section 4.2.6, differentiating the eigenvalue relation using the product rule yields

$$\dot{A}r_j + A\dot{r}_j = \dot{\lambda}_j r_j + \lambda_j \dot{r}_j \ . \tag{4.39}$$

Multiply this from the left by r_j^* and use the fact that r_j^* is a left eigenvector[16] gives

$$r_j^* \dot{A}_j r_j = \dot{\lambda}_l r_j^* r_j \ .$$

If r_j is normalized so that $r_j^* r_j = \|r_j\|_{l^2}^2 = 1$, then the right side is just $\dot{\lambda}_j$. Trading virtual perturbations for actual small perturbations turns this into the famous formula

$$\Delta\lambda_j \ = \ r_j^* \ \Delta A \ r_j \ + \ O\left(\|\Delta A\|^2\right) \ . \tag{4.40}$$

[15] Lord Rayleigh used it to study vibrational frequencies of plates and shells. Later Erwin Schrödinger used it to compute energies (which are eigenvalues) in quantum mechanics.

[16] $Ar_j = \lambda_j r_j \Rightarrow (Ar_j)^* = (\lambda_j r_j)^* \Rightarrow r_j^* A = \lambda_j r_j^*$ since $A^* = A$ and λ_j is real.

We get a condition number estimate by recasting this in terms of relative errors on both sides. The important observation is that $\|r_j\|_{l^2} = 1$, so $\|\Delta A \cdot r_j\|_{l^2} \leq \|\Delta A\|_{l^2}$ and finally

$$\left| r_j^* \; \Delta A \; r_j \right| \lessapprox \|\Delta A\|_{l^2} \; .$$

This inequality is sharp because we can take $\Delta A = \epsilon r_j r_j^*$, which is an $n \times n$ matrix with (see exercise 7) $\left\|\epsilon r_j r_j^*\right\|_{l^2} = |\epsilon|$. Putting this into (4.40) gives the also sharp inequality,

$$\left| \frac{\Delta \lambda_j}{\lambda_j} \right| \leq \frac{\|A\|_{l^2}}{|\lambda_j|} \frac{\|\Delta A\|_{l^2}}{\|A\|_{l^2}} \; .$$

We can put this directly into the abstract condition number definition (4.29) to get the conditioning of λ_j:

$$\kappa_j(A) = \frac{\|A\|_{l^2}}{|\lambda_j|} = \frac{|\lambda|_{max}}{|\lambda_j|} \tag{4.41}$$

Here, $\kappa_j(A)$ denotes the condition number of the problem of computing λ_j, which is a function of the matrix A, and $\|A\|_{l^2} = |\lambda|_{max}$ refers to the eigenvalue of largest absolute value.

The condition number formula (4.41) predicts the sizes of errors we get in practice. Presumably λ_j depends in some way on all the entries of A and the perturbations due to roundoff will be on the order of the entries themselves, multiplied by the machine precision, ϵ_{mach}, which are on the order of $\|A\|$. Only if λ_j is very close to zero, by comparison with $|\lambda_{max}|$, will it be hard to compute with high relative accuracy. All of the other eigenvalue and eigenvector problems have much worse condition number difficulties.

The eigenvalue problem for non-symmetric matrices can by much more sensitive. To derive the analogue of (4.40) for non-symmetric matrices we start with (4.39) and multiply from the left with the corresponding left eigenvector, l_j. After simplifying, the result is

$$\dot{\lambda}_j = l_j \dot{A} r_j \; , \quad \Delta \lambda_j = l_j \Delta A r_j + O\left(\|\Delta A\|^2\right) \; . \tag{4.42}$$

In the non-symmetric case, the eigenvectors need not be orthogonal and the eigenvector matrix R need not be well conditioned. For this reason, it is possible that l_k, which is a row of R^{-1} is very large. Working from (4.42) as we did for the symmetric case leads to

$$\left| \frac{\Delta \lambda_j}{\lambda_j} \right| \leq \kappa_{LS}(R) \frac{\|A\|}{|\lambda_j|} \frac{\|\Delta A\|}{\|A\|} \; .$$

Here $\kappa_{LS}(R) = \left\|R^{-1}\right\| \|R\|$ is the linear systems condition number of the right eigenvector matrix. Therefore, the condition number of the non-symmetric eigenvalue problem is (again because the inequalities are sharp)

$$\kappa_j(A) = \kappa_{LS}(R) \frac{\|A\|}{|\lambda_j|} \; . \tag{4.43}$$

Since A is not symmetric, we cannot replace $\|A\|$ by $|\lambda|_{max}$ as we did for (4.41). In the symmetric case, the only reason for ill-conditioning is that we are looking for a (relatively) tiny number. For non-symmetric matrices, it is also possible that the eigenvector matrix is ill-conditioned. It is possible to show that if a family of matrices approaches a matrix with a Jordan block, the condition number of R approaches infinity. For a symmetric or self-adjoint matrix, R is orthogonal or unitary, so that $\|R\|_{l^2} = \|R^*\|_{l^2} = 1$ and $\kappa_{LS}(R) = 1$.

The eigenvector perturbation theory uses the same ideas, with the extra trick of expanding the derivatives of the eigenvectors in terms of the eigenvectors themselves. We expand the virtual perturbation $\dot{r}_j$ in terms of the eigenvectors r_k. Call the expansion coefficients m_{jk}, and we have

$$\dot{r}_j = \sum_{l=1}^{n} m_{jl} r_l \; .$$

For the symmetric eigenvalue problem, if all the eigenvalues are distinct, the formula follows from multiplying (4.39) from the left by r_k^*:

$$m_{jk} = \frac{r_k^* \dot{A} r_j}{\lambda_j - \lambda_k} \; ,$$

so that

$$\Delta r_j = \sum_{k \neq j} \frac{r_k^* \Delta A r_j}{\lambda_j - \lambda_k} + O\left(\|\Delta A\|^2\right) \; .$$

(The term $j = k$ is omitted because $m_{jj} = 0$: differentiating $r_j^* r_j = 1$ gives $r_j^* \dot{r}_j = 0$.) This shows that the eigenvectors have condition number "issues" even when the eigenvalues are well-conditioned, if the eigenvalues are close to each other. Since the eigenvectors are not uniquely determined when eigenvalues are equal, this seems plausible. The unsymmetric matrix perturbation formula is

$$m_{kj} = \frac{l_j \dot{A} r_k}{\lambda_j - \lambda_k} \; .$$

Again, we have the potential problem of an ill-conditioned eigenvector basis, combined with the possibility of close eigenvalues. The conclusion is that the eigenvector conditioning can be problematic, even though the eigenvalues are fine, for closely spaced eigenvalues.

4.4 Software

4.4.1 Software for numerical linear algebra

There have been major government-funded efforts to produce high quality software for numerical linear algebra. This culminated in the public domain software package *LAPack*. LAPack is a combination and extension of earlier packages *Eispack*, for solving eigenvalue problems, and *Linpack*, for solving systems of equations. Many of the high quality components of Eispack and Linpack also were incorporated into Matlab, which accounts for the high quality of Matlab linear algebra routines.

Our software advice is to use LAPack or Matlab software for computational linear algebra whenever possible. It is worth the effort in coax the LAPack `make` files to work in a particular computing environment. You may avoid wasting time coding algorithms that have been coded thousands of times before, or you may avoid suffering from the subtle bugs of the codes you got from next door that came from who knows where. The professional software often does it better, for example using *balancing* algorithms to improve condition numbers. The professional software also has clever condition number estimates that often are more sophisticated than the basic algorithms themselves.

4.4.2 Test condition numbers

Part of the error flag in linear algebra computation is the condition number. Accurate computation of the condition number may be more expensive than the problem at hand. For example, computing $\|A\| \left\|A^{-1}\right\|$ is more several times more expensive than solving $Ax = b$. However, there are cheap heuristics that generally are reliable, at least for identifying severe ill-conditioning. If the condition number is so large that all relative accuracy in the data is lost, the routine should return an error flag. Using such condition number estimates makes the code slightly slower, but it makes the computed results much more trustworthy.

4.5 Resources and further reading

If you need to review linear algebra, the Schaum's Outline review book on Linear Algebra may be useful. For a practical introduction to linear algebra written by a numerical analyst, try the book by Gilbert Strang. More theoretical treatments may be found in the book by Peter Lax or the one by Paul Halmos. An excellent discussion of condition number is given in the SIAM lecture notes of Lloyd N. Trefethen. Beresford Parlett has a nice but somewhat out-of-date book on the theory and computational methods for the symmetric eigenvalue problem. The Linear Algebra book by Peter Lax also has a beautiful discussion of eigenvalue perturbation theory and some of its applications. More applications may be found in the book *Theory of Sound* by Lord Rayleigh (reprinted by Dover Press) and in any book on quantum mechanics. I do not know an elementary book that covers perturbation theory for the non-symmetric eigenvalue problem in a simple way.

The software repository *Netlib*, `http://netlib.org`, is a source for LAPack. Other sources of linear algebra software, including Mathematica and *Numerical Recipes*, are not recommended. Netlib also has some of the best available software for solving sparse linear algebra problems.

4.6 Exercises

1. Let L be the differentiation operator that takes P_3 to P_2 described in Section 4.2.2. Let $f_k = H_k(x)$ for $k = 0, 1, 2, 3$ be the Hermite polynomial basis of P_3 and $g_k = H_k(x)$ for $k = 0, 1, 2$ be the Hermite basis of P_2. What is the matrix, A, that represents this L in these bases?

2. Suppose L is a linear transformation from V to V and that f_1, ..., f_n, and g_1, ..., g_n are two bases of V. Any $u \in V$ may be written in a unique way as $u = \sum_{k=1}^{n} v_k f_k$, or as $u = \sum_{k=1}^{n} w_k g_k$. There is an $n \times n$ matrix, R that relates the f_k expansion coefficients v_k to the g_k coefficients w_k by $v_j = \sum_{k=1}^{n} r_{jk} w_k$. If v and w are the column vectors with components v_k and w_k respectively, then $v = Rw$. Let A represent L in the f_k basis and B represent L in the g_k basis.

   (a) Show that $B = R^{-1}AR$.

   (b) For $V = P_3$, and $f_k = x^k$, and $g_k = H_k$, find R.

   (c) Let L be the linear transformation $Lp = q$ with $q(x) = \partial_x(xp(x))$. Find the matrix, A, that represents L in the monomial basis f_k.

   (d) Find the matrix, B, that represents L in the Hermite polynomial basis H_k.

   (e) Multiply the matrices to check explicitly that $B = R^{-1}AR$ in this case.

3. If A is an $n \times m$ matrix and B is an $m \times l$ matrix, then AB is an $n \times l$ matrix. Show that $(AB)^* = B^*A^*$. Note that the incorrect suggestion A^*B^* in general is not compatible for matrix multiplication.

4. Let $V = R^n$ and M be an $n \times n$ real matrix. This exercise shows that $\|u\| = (u^*Mu)^{1/2}$ is a vector norm whenever M is *positive definite* (defined below).

   (a) Show that $u^*Mu = u^*M^*u = u^*\left(\frac{1}{2}(M + M^*)\right)u$ for all $u \in V$. This means that as long as we consider functions of the form $f(u) = u^*Mu$, we may assume M is symmetric. For the rest of this question, assume M is symmetric. Hint: u^*Mu is a 1×1 matrix and therefore equal to its transpose.

   (b) Show that the function $\|u\| = (u^*Mu)^{1/2}$ is homogeneous: $\|xu\| = |x|\,\|u\|$.

   (c) We say M is positive definite if $u^*Mu > 0$ whenever $u \neq 0$. Show that if M is positive definite, then $\|u\| \geq 0$ for all u and $\|u\| = 0$ only for $u = 0$.

   (d) Show that if M is symmetric and positive definite (SPD), then $|u^*Mv| \leq \|u\|\,\|v\|$. This is the *Cauchy Schwartz inequality.* Hint (a famous old trick): $\phi(t) = (u + tv)^*M(u + tv)$ is a quadratic function of t that is non-negative for all t if M is positive definite. The Cauchy Schwartz inequality follows from requiring that the minimum value of ϕ is not negative, assuming $M^* = M$.

(e) Use the Cauchy Schwartz inequality to verify the triangle inequality in its squared form $\|u+v\|^2 \leq \|u\|^2 + 2\|u\|\|u\| + \|v\|^2$.

(f) Show that if $M = I$ then $\|u\|$ is the l^2 norm of u.

5. Verify that $\|p\|$ defined by (4.3) on $V = P_3$ is a norm as long as $a < b$.

6. Suppose A is the $n \times n$ matrix that represents a linear transformation from R^n to R^n in the standard basis e_k. Let B be the matrix of the same linear transformation in the scaled basis (4.5).

   (a) Find a formula for the entries b_{jk} in terms of the a_{jk} and $\overline{u}_k$.

   (b) Find a matrix formula for B in terms of A and the *diagonal scaling* matrix $W = \text{diag}(\overline{u}_k)$ (defined by $w_{kk} = \overline{u}_k$, $w_{jk} = 0$ if $j \neq k$) and W^{-1}.

7. Show that if $u \in R^m$ and $v \in R^n$ and $A = uv^*$, then $\|A\|_{l^2} = \|u\|_{l^2} \cdot \|v\|_{l^2}$. Hint: Note that $Aw = mu$ where m is a scalar, so $\|Aw\|_{l^2} = |m| \cdot \|u\|_{l^2}$. Also, be aware of the *Cauchy Schwarz* inequality: $|v^*w| \leq \|v\|_{l^2}\|w\|_{l^2}$.

8. Suppose that A is an $n \times n$ invertible matrix. Show that

$$\left\|A^{-1}\right\| = \max_{u \neq 0} \frac{\|u\|}{\|Au\|} = \left(\min_{u \neq 0} \frac{\|Au\|}{\|u\|}\right)^{-1} .$$

9. The *symmetric part* of the real $n \times n$ matrix is $M = \frac{1}{2}(A + A^*)$. Show that $\bigtriangledown\left(\frac{1}{2}x^*Ax\right) = Mx$.

10. The *informal* condition number of the problem of computing the action of A is

$$\kappa(A) = \max_{x \neq 0,\ \Delta x \neq 0} \frac{\frac{\|A(x+\Delta x) - Ax\|}{\|Ax\|}}{\frac{\|x+\Delta x\|}{\|x\|}} .$$

Alternatively, it is the sharp constant in the estimate

$$\frac{\|A(x+\Delta x) - Ax\|}{\|Ax\|} \leq C \cdot \frac{\|x+\Delta x\|}{\|x\|} ,$$

which bounds the worst case relative change in the answer in terms of the relative change in the data. Show that for the l^2 norm,

$$\kappa = \sigma_{max}/\sigma_{min} ,$$

the ratio of the largest to smallest singular value of A. Show that this formula holds even when A is not square.

11. We wish to solve the *boundary value problem* for the differential equation

$$\frac{1}{2}\partial_x^2 u = f(x) \quad \text{for } 0 < x < 1, \tag{4.44}$$

with *boundary conditions*

$$u(0) = u(1) = 0 \; . \tag{4.45}$$

We *discretize* the interval $[0,1]$ using a uniform *grid* of points $x_j = j\Delta x$ with $n\Delta x = 1$. The $n-1$ unknowns, U_j, are approximations to $u(x_j)$, for $j = 1, \ldots, n-1$. If we use a second order approximation to $\frac{1}{2}\partial_x^2 u$, we get discrete equations

$$\frac{1}{2}\frac{1}{\Delta x^2}\left(U_{j+1} - 2U_j + U_{j-1}\right) = f(x_j) = F_j \; . \tag{4.46}$$

Together with boundary conditions $U_0 = U_n = 0$, this is a system of $n-1$ linear equations for the vector $U = (U_1, \ldots, U_{n-1})^*$ that we write as $AU = F$.

(a) Check that there are $n-1$ distinct eigenvectors of A having the form $r_{kj} = \sin(k\pi x_j)$. Here r_{kj} is the j component of eigenvector r_k. Note that $r_{k,j+1} = \sin(k\pi x_{j+1}) = \sin(k\pi(x_j + \Delta x))$, which can be evaluated in terms of r_{kj} using trigonometric identities.

(b) Use the eigenvalue information from part (a) to show that $\left\|A^{-1}\right\| \to 2/\pi^2$ as $n \to \infty$ and $\kappa(A) = O(n^2)$ (in the informal sense) as $n \to \infty$. All norms are l^2.

(c) Suppose $\widetilde{U}_j = u(x_j)$ where $u(x)$ is the exact but unknown solution of (4.44), (4.45). Show that if $u(x)$ is smooth then the *residual*[17], $R = A\widetilde{U} - F$, satisfies $\|R\| = O(\Delta x^2) = O(1/n^2)$. For this to be true we have to adjust the definition of $\|U\|$ to be consistent with the L^2 integral $\|u\|_{L^2}^2 = \int_{x=0}^1 u^2(x)dx$. The discrete approximation is $\|U\|_{l^2}^2 = \Delta x \sum_{k=1}^n U_j^2$.

(d) Show that $A\left(U - \widetilde{U}\right) = R$. Use part (b) to show that $\left\|U - \widetilde{U}\right\| = O(\Delta x^2)$ (with the Δx modified $\|\cdot\|$).

(e) (harder) Create a fourth order five point central difference approximation to $\partial_x^2 u$. You can do this using Richardson extrapolation from the second order three point formula. Use this to find an A so that solving $AU = F$ leads to a fourth order accurate U. The hard part is what to do at $j = 1$ and $j = n-1$. At $j = 1$ the five point approximation to $\partial_x^2 u$ involves U_0 and U_{-1}. It is fine to take $U_0 = u(0) = 0$. Is it OK to take $U_{-1} = -U_1$?

[17] Residual refers to the extent to which equations are not satisfied. Here, the equation is $AU = F$, which $\widetilde{U}$ does not satisfy, so $R = A\widetilde{U} - F$ is the residual.

(f) Write a program in Matlab to solve $AU = F$ for the second order method. The matrix A is symmetric and *tridiagonal* (has nonzeros only on three *diagonals*, the *main diagonal*, and the immediate sub and super diagonals). Use the Matlab matrix operation appropriate for symmetric positive definite tridiagonal matrices. Do a convergence study to show that the results are second order accurate.

(g) (extra credit) Program the fourth order method and check that the results are fourth order when $f(x) = \sin(\pi x)$ but not when $f(x) = \max(0, .15 - (x - .5)^2)$. Why are the results different?

12. This exercise explores conditioning of the non-symmetric eigenvalue problem. It shows that although the problem of computing the fundamental solution is well-conditioned, computing it using eigenvalues and eigenvectors can be an unstable algorithm because the problem of computing eigenvalues and eigenvectors is ill-conditioned. For parameters $0 < \lambda < \mu$, there is a *Markov chain transition rate* matrix, A, whose entries are $a_{jk} = 0$ if $|j - k| > 1$ If $1 \leq j \leq n - 2$, $a_{j,j-1} = \mu$, $a_{jj} = -(\lambda + \mu)$, and $a_{j,j+1} = \lambda$ (taking j and k to run from 0 to $n - 1$). The other cases are $a_{00} = -\lambda$, $a_{01} = \lambda$, $a_{n-1,n-1} = -\mu$, and $a_{n-1,n-2} = \mu$. This matrix describes a continuous time Markov process with a random walker whose position at time t is the integer $X(t)$. Transitions $X \to X + 1$ happen with rate λ and transitions $X \to X - 1$ have rate μ. The transitions $0 \to -1$ and $n - 1 \to n$ are not allowed. This is the *M/M/1 queue* used in operations research to model queues ($X(t)$ is the number of customers in the queue at time t, λ is the rate of arrival of new customers, μ is the service rate. A customer arrival is an $X \to X+1$ transition.). For each t, we can consider the row vector $p(t) = (p_1(t), \ldots, p_n(t))$ where $p_j(t) = \text{Prob}(X(t) = j)$. These probabilities satisfy the differential equation $\dot{p} = \frac{d}{dt}p = pA$. The solution can be written in terms of the *fundamental solution*, $S(t)$, which in an $n \times n$ matrix that satisfies $\dot{S} = SA$, $S(0) = I$.

(a) Show that if $\dot{S} = SA$, $S(0) = I$, then $p(t) = p(0)S(t)$.

(b) The *matrix exponential* may be defined through the Taylor series $\exp(B) = \sum_{k=0}^{\infty} \frac{1}{k!} B^k$. Use matrix norms and the fact that $\left\|B^k\right\| \leq \|B\|^k$ to show that the infinite sum of matrices converges.

(c) Show that the fundamental solution is given by $S(t) = \exp(tA)$. Top do this, it is enough to show that $\exp(tA)$ satisfies the differential equation $\frac{d}{dt}\exp(tA) = \exp(tA)A$ using the infinite series, and show $\exp(0A) = I$.

(d) Suppose $A = L\Lambda R$ is the eigenvalue and eigenvector decomposition of A, show that $\exp(tA) = L\exp(t\Lambda)R$, and that $\exp(t\Lambda)$ is the obvious diagonal matrix.

(e) Use the Matlab function `[R,Lam] = eig(A);` to calculate the eigenvalues and right eigenvector matrix of A. Let r_k be the k^{th} column of R. For $k = 1, \ldots, n$, print r_k, Ar_k, $\lambda_k r_k$, and $\|\lambda_k - Ar_k\|$ (you choose the norm). Mathematically, one of the eigenvectors is a multiple of the vector **1** defined in part h. The corresponding eigenvalue is $\lambda = 0$. The computed eigenvalue is not exactly zero. Take $n = 4$ for this, but do not hard wire $n = 4$ into the Matlab code.

(f) Let $L = R^{-1}$, which can be computed in Matlab using `L=R^(-1);`. Let l_k be the k^{th} row of L, check that the l_k are left eigenvectors of A as in part e. Corresponding to $\lambda = 0$ is a left eigenvector that is a multiple of p_∞ from part h. Check this.

(g) Write a program in Matlab to calculate $S(t)$ using the eigenvalues and eigenvectors of A as above. Compare the results to those obtained using the Matlab built in function `S = expm(t*A);`. Use the values $\lambda = 1$, $\mu = 4$, $t = 1$, and n ranging from $n = 4$ to $n = 80$. Compare the two computed $\widehat{S}(t)$ (one using eigenvalues, the other just using `expm`) using the l^1 matrix norm. Use the Matlab routine `cond(R)` to compute the condition number of the eigenvector matrix, R. Print three columns of numbers, n, error, condition number. Comment on the quantitative relation between the error and the condition number.

(h) Here we figure out which of the answers is correct. To do this, use the known fact that $\lim_{t\to\infty} S(t) = S_\infty$ has the simple form $S_\infty = \mathbf{1}p_\infty$, where **1** is the column vector with all ones, and p_∞ is the row vector with $p_{\infty,j} = ((1-r)/(1-r^n))r^j$, with $r = \lambda/\mu$. Take $t = 3*n$ (which is close enough to $t = \infty$ for this purpose) and the same values of n and see which version of $S(t)$ is correct. What can you say about the stability of computing the matrix exponential using the ill conditioned eigenvalue/eigenvector problem?

13. This exercise explores eigenvalue and eigenvector perturbation theory for the matrix A defined in exercise 12. Let B be the $n \times n$ matrix with $b_{jk} = 0$ for all (j,k) except $b_{00} = -1$ and $b_{1,n-1} = 1$ (as in exercise 12, indices run from $j = 0$ to $j = n-1$). Define $A(s) = A + sB$, so that $A(0) = A$ and $\frac{dA(s)}{ds} = B$ when $s = 0$.

(a) For $n = 20$, print the eigenvalues of $A(s)$ for $s = 0$ and $s = .1$. What does this say about the condition number of the eigenvalue eigenvector problem? All the eigenvalues of a real tridiagonal matrix are real[18] but that $A(s = .1)$ is not tridiagonal and its eigenvalues are not real.

(b) Use first order eigenvalue perturbation theory to calculate $\dot{\lambda}_k = \frac{d}{ds}\lambda_k$ when $s = 0$. What size s do you need for $\Delta\lambda_k$ to be accurately approximated by $s\dot{\lambda}_k$? Try $n = 5$ and $n = 20$. Note that first order perturbation theory always predicts that eigenvalues stay real, so $s = .1$ is much too large for $n = 20$.

[18] It is easy to see that if A is tridiagonal then there is a diagonal matrix, W, so that WAW^{-1} is symmetric. Therefore, A has the same eigenvalues as the symmetric matrix WAW^{-1}.

Chapter 5

Linear Algebra II, Algorithms

5.1 Introduction

As we say earlier, many algorithms of numerical linear algebra may be formulated as ways to calculate matrix factorizations. This point of view gives conceptual insight. Since computing the factorization is usually much more expensive than using it, storing the factors makes it possible, for example, to solve many systems of equations, $Ax = b$, with the same the same A but different b (and therefore different x), faster than if we had started over each time. Finally, when we seek high performance, we might take advantage of alternative ways to organize computations of the factors.

This chapter does not cover the many factorization algorithms in great detail. This material is available, for example, in the book of Golub and van Loan and many other places. My aim is to make the reader aware of what the computer does (roughly), and how long it should take. First I explain how the classical Gaussian elimination algorithm may be viewed as a matrix factorization, the LU factorization. The algorithm presented is not the practical one because it does not include "pivoting". Next, I discuss the Choleski (LL^*) decomposition, which is a natural version of LU for symmetric positive definite matrices. Understanding the details of the Choleski decomposition will be useful later when we study optimization methods and still later when we discuss sampling multivariate normal random variables with correlations. Finally, we show how to compute matrix factorizations, such as the QR decomposition, that lead to orthogonal matrices.

5.2 Gauss elimination and the LU decomposition

Gauss elimination is a simple systematic way to solve systems of linear equations. For example, suppose we have the system of equations

$$\begin{array}{ll} e_1 : & 2x + y + z = 4 \;\; , \\ e_2 : & x + 2y + z = 3 \;\; , \\ e_3 : & x + y + 2z = 4 \;\; . \end{array}$$

To find the values of x, y, and z, we first try to write equations that contain fewer variables, and eventually just one. We can "eliminate" x from the equation e_2 by subtracting $\frac{1}{2}$ of both sides of e_1 from e_2.

$$e_2' : \qquad x + 2y + z - \frac{1}{2}(2x + y + z) = 3 - \frac{1}{2} \cdot 4 \; ,$$

Which involves just y and z:

$$e_2' : \qquad \frac{3}{2}y + \frac{1}{2}z = 2 \; .$$

We can do the same to eliminate x from e_3, subtracting $\frac{1}{2}$ of each side of e_1 from the corresponding side of e_3:

$$e_3' : \qquad x + y + 2z - \frac{1}{2}(2x + y + z) = 4 - \frac{1}{2} \cdot 4 \ ,$$

which gives

$$e_3' : \qquad \frac{1}{2}y + \frac{3}{2}z = 2 \ .$$

We now have a pair of equations, e_2', and e_3' that involve only y and z. We can use e_2' to eliminate y from e_3; we subtract $\frac{1}{3}$ of each side of e_2' from the corresponding side of e_3' to get:

$$e_3'' : \qquad \frac{1}{2}y + \frac{3}{2}z - \frac{1}{3}(\frac{3}{2}y + \frac{1}{2}z) = 2 - \frac{1}{3} \cdot 2 \ \ ,$$

which simplifies to:

$$e_3'' : \qquad \frac{4}{3}z = \frac{5}{3} \ .$$

This completes the elimination phase. In the "back substitution" phase we successively find the values of z, y, and x. First, from e_3'' we immediately find

$$z = \frac{5}{4} \ .$$

Then we use e_2' (not e_3') to get y:

$$\frac{3}{2}y + \frac{1}{2} \cdot \frac{5}{4} = 1 \implies y = \frac{1}{4} \ .$$

Lastly, e_1 yields x:

$$2x + \frac{1}{4} + \frac{5}{4} = 4 \implies x = \frac{9}{4} \ .$$

The reader can (and should) check that $x = \frac{9}{4}$, $y = \frac{1}{4}$, and $z = \frac{5}{4}$ satisfies the original equations e_1, e_2, and e_3.

The above steps may be formulated in matrix terms. The equations, e_1, e_2, and e_3, may be assembled into a single equation involving a matrix and two vectors:

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \ .$$

The operation of eliminating x from the second equation may be carried out by multiplying this equation from the left on both sides by the *elementary* matrix

$$E_{21} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \ . \tag{5.1}$$

The result is

$$\begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \quad .$$

Doing the matrix multiplication gives

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 4 \end{pmatrix} \quad .$$

Note that the middle row of the matrix contains the coefficients from e_2'. Similarly, the effect of eliminating x from e_3 comes from multiplying both sides from the left by the elementary matrix

$$E_{31} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \quad .$$

This gives

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 1 \\ 4 \end{pmatrix} \quad ,$$

which multiplies out to become

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{3}{2} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 2 \end{pmatrix} \quad .$$

This is the matrix form of three equations e_1, e_2', and e_3'. The last elimination step removes y from the last equation using

$$E_{32} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \quad .$$

This gives

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{3}{2} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 1 \\ 2 \end{pmatrix} \quad ,$$

which multiplies out to be

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ \frac{5}{3} \end{pmatrix} \quad . \tag{5.2}$$

Because the matrix in (5.2) is upper triangular, we may solve for z, then y, then x, as before. The matrix equation (5.2) is equivalent to the system e_1, e_2', and e_3''.

We can summarize this sequence of multiplications with elementary matrices by saying that we multiplied the original matrix,

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

first by E_{21}, then by E_{31}, then by E_{32} to get the upper triangular matrix

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} \ .$$

This may be written formally as

$$E_{32}E_{31}E_{21}A = U \ .$$

We turn this into a factorization of A by multiplying successively by the inverses of the elementary matrices:

$$A = E_{21}^{-1}E_{31}^{-1}E_{32}^{-1}U \ .$$

It is easy to check that we get the inverse of an elementary matrix, E_{jk} simply by reversing the sign of the number below the diagonal. For example,

$$E_{31}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix}$$

since

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \ .$$

Also, the product of the elementary matrices just has the nonzero subdiagonal elements of all of them in their respective positions (check this):

$$\begin{aligned} L &= E_{21}^{-1}E_{31}^{-1}E_{32}^{-1} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{3} & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{1}{3} & 1 \end{pmatrix} \end{aligned}$$

Finally, the reader should verify that we actually have $A = LU$:

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{1}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} \end{pmatrix} .$$

Now we know that performing Gauss elimination on the three equations e_1, e_2, and e_3 is equivalent to finding an LU factorization of A where the lower triangular factor has ones on its diagonal.

Finally, we can turn this process around and seek the elements of L and U directly from the structure of L and U. In terms of the entries of L and U, the matrix factorization becomes

$$\begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} . \tag{5.3}$$

We may find the entries l_{jk} and u_{jk} one by one by multiplying out the product on the left and comparing to the known element on the right. For the $(1,1)$ element, we get

$$1 \cdot u_{11} = 2 ,$$

Which gives $u_{11} = 2$, as we had before. With this, we may calculate either l_{21} from matching the $(2,1)$ entries, or u_{12} from the $(1,2)$ entries. The former gives

$$l_{21} \cdot u_{11} = 1 ,$$

which, given $u_{11} = 2$, gives $l_{21} = \frac{1}{2}$. The latter gives

$$1 \cdot u_{12} = 1 .$$

and then $u_{12} = 1$. These calculations show that the LU factorization, if it exists, is unique (remembering to put ones on the diagonal of L). They also show that there is some freedom in the order in which we compute the l_{jk} and u_{jk}.

We may compute the LU factors of A without knowing the right hand side

$$\begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} .$$

If we know L and U and then learn the right hand side, we may find x, y, and z in a two stage process. First, *forward substitution* finds x', y', and z' so that

$$L \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} . \tag{5.4}$$

Then *back substitution* finds x, y, and z so that

$$U \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} . \tag{5.5}$$

This vector solves the original equations because

$$A\begin{pmatrix} x \\ y \\ z \end{pmatrix} = L \cdot U \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = L \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix} \quad .$$

Since L is lower triangular, we may solve (5.4) in the forward order, first find x', then y', then z'. The first equation is $1 \cdot x\prime = 4$ (because $l_{11} = 1$). The next is $l_{21}x' + 1 \cdot y' = 3$, which gives $y' = 1$, since $l_{21} = \frac{1}{2}$. Finally, $l_{13} = \frac{1}{2}$ and $l_{23} = \frac{1}{3}$ give $z' = \frac{5}{3}$.

Knowing x', then y', then z', we may find x, y, and z from (5.5) in the backward order (U being upper triangular), first z, then y, then x. First $u_{33}z = z'$, or $\frac{4}{3}z = \frac{5}{3}$ gives $z = \frac{5}{4}$. Then $u_{22}y + u_{23}z = y'$, or $\frac{3}{2}y + \frac{1}{2}\frac{5}{4} = 1$ gives $y = \frac{1}{4}$. Then $u_{11}x + u_{12}y + u_{23}z = x'$, or $2 \cdot x + 1 \cdot \frac{1}{4} + 1 \cdot \frac{5}{4} = 4$, gives $x = \frac{9}{4}$. The reader will recognize that these calculations are essentially the same as the ones at the beginning of the section, as solving a system using LU factorization is essentially the same as using elementary Gauss elimination.

The general LU algorithm for solving linear systems should be clear from this example. We have an $n \times n$ matrix A, and a column vector $b \in R^n$, and we wish to find another column vector x so that $Ax = b$. This is equivalent to n linear equations in the n unknowns $x_1, \ldots, x_n$. We first compute the LU decomposition of A, in one of several related ways. Then we solve a lower triangular system of equations $Ly = b$ using forward elimination. The intermediate vector entries, $y_1, \ldots, y_n$, are what we would have gotten had we applied Gauss elimination to the right hand side and A at the same time. Finally, we perform back substitution, finding x with $Ux = y$. Multiplying this by L and using $LU = a$ and $Ly = b$, we see that this x solves our problem.

Dense linear algebra computations (computations with dense matrices) like these can take lots of computer time for large n. We make an abstract count of the number of floating point operations involved in Gauss elimination. The elimination process goes from $k = 1$ to $k = n-1$, at each stage removing (setting to zero through elimination) the elements a_{jk} for $j > k$ by subtracting a multiple of row k from row j. Since rows k and j at this stage have each have $n-k$ nonzero entries, the work for row j is $n - k$ additions and multiplications. The number of rows eliminated at stage k is $n - 1$, so the work for stage k is about $(n - k)^2$. The total work for all the stages is about $\sum_{k=1}^{n}(n - k)^2 = \sum_{k=1}^{n} k^2 \approx \frac{1}{3}n^3$. Once we have L and U, the forward and backward substitutions take $O(n^2)$ operations each, far fewer than $\frac{1}{3}n^3$. For this reason, if we need to solve more than one problem $Ax = b$ with the same A, we should compute and save the LU factors once, then use them with the different b vectors.

We can appreciate the $\frac{1}{3}n^3$ work estimate in several ways. If the computer time to factor an $n \times n$ matrix is T, the time for a $2n \times 2n$ matrix will be roughly $8T$. Ten seconds would become a minute and twenty seconds. If a one gigahertz computer performed one addition and multiplication per cycle, that would be 10^9 adds and multiplies per second. Factoring a hypothetical $50,000 \times 50,000$ matrix requires about $\frac{1}{3}(50,000)^3 \approx 40 \cdot 10^{12}$ operations, which would take about

$4 \cdot 10^4$ seconds, which is about eleven hours. I doubt any desktop computer could do it nearly that fast.

The elimination and factorization algorithms just described may fail or be numerically unstable even when A is well conditioned. To get a stable algorithm, we need to introduce "pivoting". In the present context[1] this means adaptively reordering the equations or the unknowns so that the elements of L do not grow. Details are in the references.

5.3 Choleski factorization

Many applications call for solving linear systems of equations with a symmetric and positive definite A. An $n \times n$ matrix is *positive definite* if $x^*Ax > 0$ whenever $x \neq 0$. Symmetric positive definite (SPD) matrices arise in many applications. If B is an $m \times n$ matrix with $m \geq n$ and $\text{rank}(B) = n$, then the product $A = B^*B$ is SPD. This is what happens when we solve a linear least squares problem using the normal equations, see Section 4.2.8). If $f(x)$ is a scalar function of $x \in R^n$, the *Hessian* matrix of second partials has entries $h_{jk}(x) = \partial^2 f(x)/\partial x_j \partial x_k$. This is symmetric because $\partial^2 f/\partial x_j \partial x_k = \partial^2 f/\partial x_k \partial x_j$. The minimum of f probably is taken at an x_* with $H(x_*)$ positive definite, see Chapter 6. Solving elliptic and parabolic partial differential equations often leads to large sparse SPD linear systems. The variance/covariance matrix of a multivariate random variable is symmetric, and positive definite except in degenerate cases.

We will see that A is SPD if and only if A has an LU factorization with $U = L^*$, i.e. $A = LL^*$ for a lower triangular matrix, L. This is the *Choleski factorization*, or *Choleski decomposition* of A. As with the LU factorization, we can find the entries of L from the equations for the entries of $LL^* = A$ one at a time, in a certain order. We write it out:

$$\begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & \vdots \\ l_{31} & l_{32} & l_{33} & \ddots & \\ \vdots & \vdots & & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & & l_{nn} \end{pmatrix} \cdot \begin{pmatrix} l_{11} & l_{21} & l_{31} & \cdots & l_{n1} \\ 0 & l_{22} & l_{32} & \cdots & l_{n2} \\ 0 & 0 & l_{33} & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & \cdots & & l_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\ a_{21} & a_{22} & a_{32} & \cdots & a_{n2} \\ a_{31} & a_{32} & a_{33} & \ddots & \vdots \\ \vdots & \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & & a_{nn} \end{pmatrix} .$$

Notice that we have written, for example, a_{32} for the $(2,3)$ entry because A is symmetric. We start with the top left corner. Doing the matrix multiplication

[1] The term "pivot" means something different, for example, in linear programming.

gives

$$l_{11}^2 = a_{11} \implies l_{11} = \sqrt{a_{11}} \; .$$

The square root is real because $a_{11} > 0$ because A is positive definite and[2] $a_{11} = e_1^* A e_1$. Next we match the $(2,1)$ entry of A. The matrix multiplication gives:

$$l_{21} l_{11} = a_{21} \implies l_{21} = \frac{1}{l_{11}} a_{21} \quad .$$

The denominator is not zero because $l_{11} > 0$ because $a_{11} > 0$. We could continue in this way, to get the whole first column of L. Alternatively, we could match $(2,2)$ entries to get l_{22}:

$$l_{21}^2 + l_{22}^2 = a_{22} \implies l_{22} = \sqrt{a_{22} - l_{21}^2} \quad .$$

It is possible to show (see below) that if the square root on the right is not real, then A was not positive definite. Given l_{22}, we can now compute the rest of the second column of L. For example, matching $(3,2)$ entries gives:

$$l_{31} \cdot l_{21} + l_{32} \cdot l_{22} = a_{32} \implies l_{32} = \frac{1}{l_{22}} \left(a_{32} - l_{31} \cdot l_{21} \right) \quad .$$

Continuing in this way, we can find all the entries of L. It is clear that if L exists and if we always use the positive square root, then all the entries of L are uniquely determined.

A slightly different discussion of the Choleski decomposition process makes it clear that the Choleski factorization exists whenever A is positive definite. The algorithm above assumed the existence of a factorization and showed that the entries of L are uniquely determined by $LL^* = A$. Once we know the factorization exists, we know the equations are solvable, in particular, that we never try to take the square root of a negative number. This discussion represents L as a product of elementary lower triangular matrices, a point of view that will be useful in constructing the QR decomposition (Section 5.4).

Suppose we want to apply Gauss elimination to A and find an elementary matrix of the type (5.1) to set $a_{21} = a_{12}$ to zero. The matrix would be

$$E_{21} = \begin{pmatrix} 1 & 0 & \cdots & \\ \frac{-a_{12}}{a_{11}} & 1 & 0 & \cdots \\ 0 & 0 & \ddots & \\ \vdots & & & \end{pmatrix} \quad .$$

Multiplying out gives:

$$E_{21} A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ 0 & a_{22}' & a_{23}' & \cdots \\ a_{13} & a_{23} & a_{33} & \\ \vdots & & & \end{pmatrix} \quad .$$

[2]Here e_1 is the vector with one as its first component and all the rest zero. Similarly $a_{kk} = e_k^* A e_k$.

Only the entries in row two have changed, with the new values indicated by primes. Note that $E_{21}A$ has lost the symmetry of A. We can restore this symmetry by multiplying from the right by E_{21}^* This has the effect of subtracting $\frac{a_{12}}{a_{11}}$ times the first column of $E_{21}A$ from the second column. Since the top row of A has not changed, this has the effect of setting the $(1,2)$ entry to zero:

$$E_{21}AE_{21}^* = \begin{pmatrix} a_{11} & 0 & a_{13} & \cdots \\ 0 & a'_{22} & a'_{23} & \cdots \\ a_{13} & a'_{23} & a_{33} & \\ \vdots & & & \end{pmatrix} .$$

Continuing in this way, elementary matrices E_{31}, etc. will set to zero all the elements in the first row and top column except a_{11}. Finally, let D_1 be the diagonal matrix which is equal to the identity except that $d_{11} = 1/\sqrt{a_{11}}$. All in all, this gives ($D_1^* = D_1$):

$$D_1 E_{n1} \cdots E_{31} E_{21} A E_{21}^* E_{31}^* \cdots E_{n1}^* D_1^* = \begin{pmatrix} 1 & 0 & 0 & \cdots \\ 0 & a'_{22} & a'_{23} & \cdots \\ 0 & a'_{23} & a'_{33} & \\ \vdots & & & \end{pmatrix} . \qquad (5.6)$$

We define L_1 to be the lower triangular matrix

$$L_1 = D_1 E_{n1} \cdots E_{31} E_{21} ,$$

so the right side of (5.6) is $A_1 = L_1 A L_1^*$ (check this). It is nonsingular since D_1 and each of the elementary matrices are nonsingular. To see that A_1 is positive definite, simply define $y = L^* x$, and note that $y \neq 0$ if $x \neq 0$ (L_1 being nonsingular), so $x^* A_1 x = x^* L A L^* x = y^* A y > 0$ since A is positive definite. In particular, this implies that $a'_{22} > 0$ and we may find an L_2 that sets a'_{22} to one and all the a_{2k} to zero.

Eventually, this gives $L_{n-1} \cdots L_1 A L_1^* \cdots L_{n-1}^* = I$. Solving for A by reversing the order of the operations leads to the desired factorization:

$$A = L_1^{-1} \cdots L_{n-1}^{-1} L_{n-1}^{-*} \cdots L_1^{-*} ,$$

where we use the common convention of writing M^{-*} for the inverse of the transpose of B, which is the same as the transpose of the inverse. Clearly, L is given by $L = L_1^{-1} \cdots L_{n-1}^{-1}$.

All this may seem too abstract, but as with Gauss elimination from Section 5.2, the products of inverses of elementary matrices are easy to figure out explicitly.

Once we have the Choleski decomposition of A, we can solve systems of equations $Ax = b$ using forward and back substitution, as we did for the LU factorization.

5.4 Orthogonal matrices, least squares, and the QR factorization

Many problems in linear algebra call for linear transformations that do not change the l^2 norm:

$$\|Qx\|_{l^2} = \|x\|_{l^2} \qquad \text{for all } x \in R^n. \tag{5.7}$$

A real matrix satisfying (5.7) is orthogonal, because[3]

$$\|Qx\|_{l^2}^2 = (Qx)^* \, Qx = x^*Q^*Qx = x^*x = \|x\|_{l^2}^2 \ .$$

(Recall that $Q^*Q = I$ is the definition of orthogonality for square matrices.)

There is a version of Gauss elimination for solving linear least squares problems that uses orthogonal matrices Q_{jk} rather than elementary lower triangular matrices E_{jk} to make A upper triangular. A related problem is constructing an orthonormal basis for a subspace. If $v_1, \ldots, v_m$ span $V \subseteq R^n$, we desire an orthonormal basis for V or an orthonormal basis for the orthogonal complement of V. Finally, reductions using the Q_{jk} are a first step in many algorithms for finding eigenvalues and eigenvectors of symmetric matrices.

The elementary matrix E_{21} operates only on the top two rows of a matrix and modifies only the second row. The *Givens rotation*, $Q_{21}(\theta)$, is an orthogonal matrix that modifies both rows one and two. As with E_{21}, the parameter may be chosen so that $Q_{21}A$ has a zero in the $(2,1)$ position. Suppose we have a 2×2 matrix $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$, and we want an orthogonal matrix $Q = \begin{pmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{pmatrix}$ so that $QA = A' = \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}$ has $a'_{21} = 0$. An orthogonal matrix preserves angles between vectors,[4] so an orthogonal matrix operating on the plane (R^2) must be a simple rotation possibly followed by a reflection about some line (reflect on this). The 2×2 matrix that represents rotation through angle θ is $Q = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$, so $a'_{21} = -\sin(\theta)a_{11} + \cos(\theta)a_{21}$. We achieve $a'_{21} = 0$ with the choice

$$\begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} = \frac{1}{\sqrt{a_{11}^2 + a_{21}^2}} \begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix} \ .$$

A Givens rotation, Q_{jk}, in dimension $n > 2$ acts only on rows j and k of A.

---

[3]This shows an orthogonal matrix satisfies (5.7). Exercise 8 shows that a matrix satisfying (5.7) must be orthogonal.

[4]The angle between vectors x and y is given by $x^*y = \|x\| \cdot \|y\| \cdot \cos(\theta)$. This shows that if Q preserves lengths of vectors and inner products, it also preserves angles.

For example

$$Q_{42} = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & \\ 0 & \cos(\theta) & 0 & \sin(\theta) & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots & \\ 0 & -\sin(\theta) & 0 & \cos(\theta) & 0 & \\ \vdots & 0 & 0 & 0 & 1 & \ddots \\ & & \vdots & & \ddots & \ddots \end{pmatrix} .$$

This is the identity matrix except for the $\cos(\theta)$ and $\sin(\theta)$ entries shown. The matrix $A' = Q_{42}A$ is the same as A except in rows 2 and 4. We could choose θ to set $a'_{42} = 0$, as above.

The Givens' elimination strategy for the least squares problem (4.21) is an orthogonal matrix version of Gauss elimination. It is based on the observation that orthogonal Q does not change the l^2 norm of the residual: $\|Qr\|_{l^2} = \|r\|_{l^2}$. That means that solving (4.21) is equivalent to solving

$$\min_x \|QAx - Qb\|_{l^2} \ .$$

If we choose $Q = Q_{jk}$ with $j > k$ to eliminate (set to zero) all the entries a_{jk} below the diagonal ($j < k$). Of course, we must apply the Q_{jk} in the same order to b. In the end, we have an equivalent problem

$$\min_x \|Rx - b'\|_{l^2} \ , \tag{5.8}$$

where R is upper triangular[5]:

$$\left(\begin{array}{cccc} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & r_{nn} \\ \hline 0 & \cdots & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & & 0 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} - \left(\begin{array}{c} b'_1 \\ b'_2 \\ \vdots \\ b'_n \\ \hline b'_{n+1} \\ \vdots \\ b'_m \end{array}\right) = \left(\begin{array}{c} r'_1 \\ r'_2 \\ \vdots \\ r'_n \\ \hline r'_{n+1} \\ \vdots \\ r'_m \end{array}\right)$$

Assuming none of the diagonals r_{jj} is zero, the best we can do is to choose x so that the first n components of r' are set to zero. Clearly x has no effect on the last $m - n$ components of r'

5.5 Projections and orthogonalization

There is a geometric interpretion of the solution to the linear least squares problem as an *orthogonal projection* in the l^2 norm. This section explains that

[5]The upper triangular part of the LU decomposition is called U while the upper triangluar part of the QR decomposition is called R. The use of r_{jk} for the entries of R and r_j for the entries of the residual is not unique to this book.

interpretation and shows how to use the QR decomposition to compute other projections and to construct a well conditioned basis for a subspace of R^n given by linear constraints. Section 5.4 shows how to calculate this projection using the QR decomposition of A. There are other projections tha

5.6 Rank and conditioning of a basis set

5.7 Software: Performance and cache management

In scientific computing, *performance* refers to the time it takes to run the program that does the computation. Faster computers give more performance, but so do better programs. To write high performance software, we should know what happens inside the computer, something about the compiler and about the hardware. This and several later Software sections explore performance-related issues.

Memory hierarchy is one aspect of computing hardware that influences performance. How long it takes to execute an instruction such as `a = b * c;` depends on where `a`, `b`, and `c` are stored. The time needed to *fetch* the *operands*, `b` and `c`, and to *store* the result, `a`, can be far greater than the time needed to do the operation $a = b * c$. This delay depends on where in the memory hierarchy the variables are stored when the instruction is executed.

A computer processor such as the Pentium 4 (P4) operates in time slots called *cycles*. We will give numbers for a specific but typical P4 that we call Paul. The speed of a processor is given in Hertz (cycles per second). Paul runs at 2 GHz, or 2×10^9 cycles per second. At this rate, each cycle takes $.5 \times 10^{-9}$ seconds, which is half a nanosecond. If the *pipeline* is full (see Section **??**), the processor can perform one floating point operation per cycle, if it can get the data.

The structure and performance of the memory hierarchy differs from processor to processor. On the top of the hierarchy are a small number of *registers* (about 20 for Paul). Next is *level one cache*. For Paul, transfer between a register and level one cache takes one or two cycles. Paul has two separate level one caches of 32 KBytes (32×2^{10} bytes) each, one for instructions (see Section **??**), and the other for data, such as the values of `a`, `b`, and `c`. Next is *level two* cache with a 3 or 4 cycle delay, which has 2 MBytes in Paul's case. Below the cache is main memory. One high end office desktop in 2006 had main memory with a 10 nanosecond *latency* and 6.4 GByte/sec *bandwidth*.[6] Latency is the time delay between asking for a set of numbers (see below) and the arrival of the first one. For Paul, 10 nanoseconds is about 20 cycles. Once data starts

[6]The *band* in *bandwidth* is the range of frequencies, the *frequency band* that can be realized on a given connection. A book on signal processing will show that the number of bits that a connection can transmit per second is proportional to the range of frequencies it can support, its *bandwidth*.

flowing, consecutive bytes are delivered at the rate of 6.4×10^9 bytes/sec, or about one double precision word per nanosecond.

Logically, computer memory is a long list of bytes. For each integer *address*, k, in the *address space* $0 \leq k < MS$ (MS for memory size), there is one byte of information. In a *fetch* operation, the processor specifies an address and the memory provides the contents of that address from somewhere in the hierarchy. In a *store* operation, the processor specifies the contents and the address the memory should write it into. Except for performance, the caches do not change the way the processor views data.

A cache consists of a set of cache *lines*, each one consisting of a short sequence of consecutive memory bytes. Paul's cache lines hold 128 bytes, or 16 double precision floating point words. Each of his level one caches has 256 such lines. The address space is divided into neighboring disjoint lines, any one of which may occupy a level one or level two cache line. At any given moment, some of these lines are in level one or level two caches.

When the processor requests a byte that is not in a register, the memory will get it from the nearest place, level one cache, level two cache, or main memory.[7] For example, if it asks for byte 129 and the line of bytes from 128 to 255 is in cache, the memory will supply the cached value with the delay for the cache it was in. If byte 129 is not in cache, a *cache miss*, the whole cache line of bytes 128 to 255 will be copied from main memory to cache. This takes about 60 cycles for Paul: A 10 nsec latency and 128 byte/(6.4 GByte/sec) = 20 nsec transfer time give 30 nsec, or 60 cycles. Whatever data was in that cache line would be *flushed*, lost from cache but saved in main memory.

Many details of *cahce management* are constantly changing as hardware improves and are beyond the programmer's control. For example, Paul may have to decide which cache line to flush when there is a cache miss. If he stores a byte that is in cache (changing its value), does he *write back* that value to main memory at the same time or only when the line is flushed? How does he decide which lines to put in level one and level two? Processor manufacturers (Intel, AMD, etc.) make available detailed descriptions of the cache strategies for each of their processors.

Variable reuse and *data locality* are two factors under the programmer's control that effect cache and ultimately software performance. If the variable `b` needs to be used many times, the programmer can avoid cache misses by making these consecutive. If a program uses byte k, it is likely that fetching byte $k+1$ immediately after will not cause a cache miss because k and $k+1$ are likely to be in the same cache line.

To see this concretely, consider the product of $n \times n$ matrices $A = BC$ using the vanilla formula $a_{jk} = \sum_{l=1}^{n} b_{jl} c_{lk}$. If n is known at compile time,[8] the code could be

[7] We also could consider the hard drive as a level in the memory hierarchy below main memory. Transfer between main memory and the hard drive is very slow on the nanosecond time scale.

[8] It is a tragedy that C/C++ is so clumsy in handling arrays of more than one index.

```
double a[N][N], b[N][N], c[N][N];
   .
   .   (define B and C)
   .
for ( j = 0; j < N; j++ ) { // j from 0 to N-1 in C.
   for ( k = 0; k < N; k++ ) {
      sum = 0;
      for ( l = 0; l < N; l++ )   // The inner loop.
         sum += b[j][l]*c[l][k];  //    one line l loop
      a[j][k] = sum;
     }  // end k loop
  } // end of j loop
```

Let us examine the pattern of memory references in this code. In C/C++ (and not in FORTRAN) the second index of the array "moves fastest". In memory, the entries of A are stored in the order `a[0][0], a[0][1], ... ,` `a[0][N-1],a[1][0], a[1][1], ...`. The *inner loop*, `for ( l=0; l<N; l++)` `..`, references consecutive entries of b, which is good because each cache miss loads 16 consecutive elements of b, so only one in 16 references to b leads to a cache miss. References to c are offset by a *stride* of N. If $N > 15$, each c reference in the inner loop references a different cache line and therefore probably causes a cache miss.

A simple solution, if memory is not a problem, would be to compute the transpose of `c` before the matrix multiplication:

```
double ct[N][N];
   .
   .   (as before)
   .
for ( j = 0; j < N; j++ )      // Compute the transpose of c.
  for ( k = 0; k < N; k++ )
     ct[j][k] = c[k][j];
for ( j = 0; j < N; j++ ) { // Martix multiply loops as before
           .
           .   (as before)
           .
         sum += b[j][l]*ct[k][l];  // Use ct for data locality.
           .
           .   (as before)
           .
```

The new inner loop accesses both arrays with a unit stride to improve data locality. For large n, the loop that computes C^* uses $2n^2$ memory references, one fetch and one store for each (j, k), while the loop that computes $A = BC$ uses n^3 references to C. Thus, the time computing C^* should be tiny compared to the time saved in computing the matrix product.

5.8 References and resources

The algorithms of numerical linear algebra for dense matrices are described in great detail in the book by Charles vanLoan and Gene Golub and in the book by James Demmel. The book ??? describes computational methods for matrices stored in sparse matrix format. Still larger problems are solved by *iterative methods*. Generally speaking, iterative methods are not very effective unless the user can concoct a good *preconditioner*, which is an approximation to the inverse. Effective preconditioners usually depend in physical understanding of the problem and are problem specific.

Despite its importance for scientific computing, there is a small literature on high performance computing. The best book available in 2006 seems to be *High Performance Computing* by Kevin Down and Charles Severance. Truly high performance computing is done on computers with more than one processor, which is called *parallel computing*. There are many specialized algorithms and programming techniques for parallel computing.

The LAPack software package is designed to make the most of memory hierarchies. In 2006, it is the best way to do high performance computational linear algebra, at least on dense matrices. It comes with a package called *Atlas* that chooses good parameters (block sizes, etc.) for LAPack depending on the cache performance of your particular processor. The LAPack manual is published by SIAM, the *Society for Industrial and Applied Mathematics*.

5.9 Exercises

1. The solution to $Au = b$ may be written $b = A^{-1}u$. This can be a good way to analyze algorithms involving linear systems (see Sections 4.3.1 and 6.3). But we try to avoid forming A^{-1} explicitly in computations because it is more that twice as expensive as solving the linear equations. A good way to form $B = A^{-1}$ is to solve the matrix equation $AB = I$. Gauss elimination applied to A gives $A = LU$, where the entries of L are the pivots used in elimination.

   (a) Show that about $\frac{1}{3}n^3$ work reduces $AB = I$ to $UB = L^{-1}$, where the entries of U and L^{-1} are known.

   (b) Show that computing the entries of B from $UB = L^{-1}$ takes about $\frac{1}{2}n^3$ work. Hint: It takes one flop per element for each of the n elements of the bottom row of B, then two flops per element of the $n-1$ row of B, and so on to the top. The total is $n \times (1+2+\cdots+n)$.

   (c) Use this to verify the claim that computing A^{-1} is more than twice as expensive as solving $Au = b$.

2. Show that a symmetric $n \times n$ real matrix is positive definite if and only if all its eigenvalues are positive. Hint: If R is a right eigenvector matrix, then, for a symmetric matrix we may normalize so that $R^{-1} = R^*$ and

$A = R\Lambda R^*$, where Λ is a diagonal matrix containing the eigenvalues (See Sections 4.2.5 and 4.2.7). Then $x^*Ax = x^*R\Lambda R^*x = (x^*R)\Lambda(R^*x) = y^*\Lambda y$, where $y = R^*x$. Show that $y^*\Lambda y > 0$ for all $y \neq 0$ if and only if all the diagonal entries of Λ are positive. Show that if A is positive definite, then there is a $C > 0$ so that $x^*Ax > C\,\|x\|_{l^2}$ for all x. (Hint: $\|x\|_{l^2} = \|x\|_{l^2}$, $C = \lambda_{\min}$.)

3. Write a program to compute the LL^* decomposition of an SPD matrix A. Your procedure should have as arguments the dimension, n, and the matrix A. The output should be the Choleski factor, L. Your procedure must detect and report a matrix that is not positive definite and should not perform the operation `sqrtc` if $c < 0$. Write another procedure that has n and L as arguments and returns the product LL^*. Hand in: (i) printouts of the two procedures and the driving program, (ii) a printout of results showing that the testing routine reports failure when $LL^* \neq A$, (iii) a printout showing that the Choleski factoring procedure reports failure when A is not positive definite, (iv) a printout showing that the Choleski factoring procedure works correctly when applied to a SPD matrix, proven by checking that $LL^* = A$.

4. A square matrix A has *bandwidth* $2k + 1$ if $a_{jk} = 0$ whenever $|j - k| > k$. A *subdiagonal* or *superdiagonal* is a set of matrix elements on one side of the main diagonal (below for sub, above for super) with $j-k$, the distance to the diagonal, fixed. The bandwidth is the number of nonzero bands. A bandwidth 3 matrix is *tridiagonal*, bandwidth 5 makes *pentadiagonal*, etc.

   (a) Show that a SPD matrix with bandwidth $2k+1$ has a Choleski factor with nonzeros only on the diagonal and up to k bands below.

   (b) Show that the Choleski decomposition algorithm computes this L in work proportional to $k^2 n$ (if we skip operations on entries of A outside its nonzero bands).

   (c) Write a procedure for Choleski factorization of tridiagonal SPD matrices, apply it to the matrix of Exercise 11, compare the running time with this dense matrix factorizer and the one from Exercise 5.3. Of course, check that the answer is the same, up to roundoff.

5. Suppose $v_1, \ldots, v_m$ is an orthonormal basis for a vector space $V \subseteq R^n$. Let L be a linear transofrmation from V to V. Let A be the matrix that represents L in this basis. Show that the entries of A are given by

$$a_{jk} = v_j^* L v_k \; . \tag{5.9}$$

   Hint: Show that if $y \in V$, the representation of y is this basis is $y = \sum_j y_j v_j$, where $y_j = v_j^* y$. In physics and theoretical chemistry, inner products of the form (5.9) are called *matrix elements*. For example, the eigenvalue perturbation formula (4.40) (in physicist terminology) simply says that the perturbation in an eigenvalue is (nearly) equal to to the appropriate matrix element of the perturbation in the matrix.

6. Suppose A is an $n \times n$ symmetric matrix and $V \subset R^n$ is an *invariant subspace* for A (i.e. $Ax \in V$ if $x \in V$). Show that A defines a linear transformation from V to V. Show that there is a basis for V in which this linear transformation (called A *restricted to* V) is represented be a symmmetric matrix. Hint: construct an orthonormal basis for V.

7. If Q is an $n \times n$ matrix, and $(Qx)^* Qy = x^* y$ for all x and y, show that Q is an orthogonal matrix. Hint: If $(Qx)^* Qy = x^*(Q^* Q)y = x^* y$, we can explore the entries of $Q^* Q$ by choosing particular vectors x and y.

8. If $\|Qx\|_{l^2} = \|x\|_{l^2}$ for all x, show that $(Qx)^* Qy = x^* y$ for all x and y. Hint (*polarization*): If $\|Q(x + sy)\|_{l^2}^2 = \|x + sy\|_{l^2}^2$ for all s, then $(Qx)^* Qy = x^* y$.

Chapter 6

Nonlinear Equations and Optimization

6.1 Introduction

This chapter discusses two related computational problems. One is *root finding*, or solving systems of nonlinear equations. This means that we seek values of n variables, $(x_1, \ldots, x_n) = x \in R^n$, to satisfy n nonlinear equations $f(x) = (f_1(x), \ldots, f_n(x)) = 0$. We assume that $f(x)$ is a smooth function of x. The other problem is *smooth optimization*, or finding the minimum (or maximum[1]) value of a smooth *objective function*, $V(x)$. These problems are closely related. Optimization algorithms use the gradient of the objective function, solving the system of equations $g(x) = \nabla V(x) = 0$. However, the optimization problem has special structure that makes it easier than the general root finding problem.

The theory here is for *black box* ("closed box" would be more accurate) methods. This means algorithms that do not depend on details of the definitions of the functions $f(x)$ or $V(x)$. Instead, they use procedures that evaluate $f(x)$ or $V(x)$ for a given x value, such as `int fEval( double* f, double* x)` (f and x probably vectors). The code doing the root finding will learn about f only by "user-supplied" procedures, such as `fEval`, that supply values of f or V and their derivatives. The person writing the root finding or optimization code need not "open the box" to see how `fEval` works. This makes it possible for specialists to create general purpose optimization and root finding software that is efficient and robust, without knowing all the problems it may be applied to. It is unlikely that someone could create a general purpose root finder that is as good as the best available on the web.

There is a strong incentive to use derivative information as well as function values. For root finding, we use the $n \times n$ Jacobian matrix, $f'(x)$, with entries $f'(x)_{jk} = \partial_{x_k} f_j(x)$. For optimization, we use the gradient and the $n \times n$ *Hessian* matrix of second partials $H(x)_{jk} = \partial_{x_j} \partial_{x_k} V(x)$. It may seem like too much extra work to go from the n components of f to the n^2 entries of f', but algorithms that use f' often are much faster and more reliable than those that do not.

There are drawbacks to using general-purpose software that treats each specific problem as a black box. Large-scale computing problems usually have specific features that have a big impact on how they should be solved. Reformulating a problem to fit into a generic $f(x) = 0$ or $\min_x V(x)$ form may increase the condition number. Problem-specific solution strategies may be more effective than the generic Newton's method. In particular, the Jacobian or the Hessian may be sparse in a way that general purpose software cannot take advantage of. Some more specialized algorithms are in Exercise 6c (Marquart-Levenberg for nonlinear least squares), and Section **??** (Gauss-Seidel iteration for large systems). iteration

The algorithms discussed here are *iterative* (see Section 2.4). They produce a sequence of approximations, or *iterates*, that should converge to the desired solution, x_*. In the simplest case, each *iteration* starts with a *current iterate*, $\overline{x}$, and produces a *successor iterate*, $x' = \Phi(\overline{x})$. The algorithm starts from an

[1]Optimization refers either to minimization or maximization. But finding the maximum of $V(x)$ is the same as finding the minimum of $-V(x)$.

initial guess[2], x_0, then produces a sequence of iterates $x_{k+1} = \Phi(x_k)$. The algorithm succeeds if the iterates converge to the solution: $x_k \to x_*$ as $k \to \infty$. An iterative method fails if the iterates fail to converge or converge to the wrong answer. For an algorithm that succeeds, the *convergence rate* is the rate at which $\|x_k - x_*\| \to 0$ as $k \to \infty$.

An iterative method is *locally convergent* if it succeeds whenever the initial guess is close enough to the solution. That is, if there is an $R > 0$ so that if $\|x_0 - x_*\| \leq R$ then $x_k \to x_*$ as $k \to \infty$. The algorithms described here, mostly variants of *Newton's method*, all are locally convergent if the problem is nondegenerate (terminology below). An iterative method is *globally convergent* if it finds the answer from any initial guess. Between these extremes are algorithms that are more or less *robust*. The algorithms described here consist of a relatively simple locally convergent method, usually Newton's method, enhanced with *safeguards* that guarantee that some progress is made toward the solution from any $\overline{x}$. We will see that safeguards based on mathematical analysis and reasoning are more effective than heuristics.

All iterative methods need some kind of *convergence criterion* (more properly, *halting criterion*). One natural possibility is to stop when the relative change in x is small enough: $\|x_{k+1} - x_k\| / \|x_k\| \leq \epsilon$. It also makes sense to check that the *residuals*, the components of $f(x)$ or $\nabla V(x)$, are small. Even without roundoff error, an iterative method would be very unlikely to get the exact answer. However, as we saw in Section 2.4, good algorithms and well conditioned problems still allow essentially optimal accuracy: $\|x_k - x_*\| / \|x_*\| \sim \epsilon_{mach}$.

The final section of this chapter is on *iterative methods* that do not use higher derivatives. The discussion applies to linear or nonlinear problems. For optimization, it turns out that the rate of convergence of simple iterative methods is determined by the condition number of H for solving linear systems involving H, see Section 4.3.1 and the condition number formula (4.36). More precisely, the number of iterations needed to reduce the error by a factor of 2 is proportional to $\kappa(H) = \lambda_{\max}(H)/\lambda_{\min}(H)$. This, more than linear algebra roundoff, explains our fixation on condition number. The condition number $\kappa(H) = 10^4$ could arise in a routine partial differential equation problem. This bothers us not so much because it makes us lose 4 out of 16 double precision digits of accuracy, but because it takes tens of thousands of iterations to solve the damn problem.

6.2 Solving a single nonlinear equation

The simplest problem is that of solving a single equation in a single variable: $f(x) = 0$. Single variable problems are easier than multi-variable problems. There are simple criteria that guarantee a solution exists. Some algorithms for one dimensional problems, Newton's method in particular, have analogues for

[2]Here, the subscript denotes the iteration number, not the component. In n dimensions, iterate x_k has components $x_k = (x_{k1}, \ldots, x_{kn})$.

higher dimensional problems. Others, such as bisection, are strictly one dimensional. Algorithms for one dimensional problems are components for algorithms for higher dimensional problems.

6.2.1 Bisection

The bisection algorithm, or *bisection search*, is the simplest and most robust way to find a zero of a function on one variable. It does not require $f(x)$ to be differentiable, but merely continuous. It is based on a simple topological fact called the *intermediate value theorem*: if $f(x)$ is a continuous real-valued function of x on the interval $a \leq x \leq b$ and $f(a) < 0 < f(b)$, then there is at least one $x_* \in (a,b)$ with $f(x_*) = 0$. A similar theorem applies in the case $b < a$ or $f(a) > 0 > f(b)$.

The bisection search algorithm consists of repeatedly bisecting an interval in which a root is known to lie. Suppose we have an interval[3] $[\overline{a}, \overline{b}]$ with $f(\overline{a}) < 0$ and $f(\overline{b}) > 0$. The intermediate value theorem tells us that there is a root of f in $[\overline{a}, \overline{b}]$. The uncertainty is the location of this root is the length of the interval $\left|\overline{b} - \overline{a}\right|$. To cut that uncertainty in half, we bisect the interval. The midpoint is $\overline{c} = \left(\overline{a} + \overline{b}\right)/2$. We determine the sign of $f(\overline{c})$, probably by evaluating it. If $f(\overline{c}) > 0$ then we know there is a root of f in the sub interval $[\overline{a}, \overline{c}]$. In this case, we take the new interval to be $[a', b']$, with $a' = \overline{a}$ and $b' = \overline{c}$. In the other case, $f(\overline{c}) < 0$, we take $a' = \overline{c}$ and $b' = \overline{b}$. In either case, f changes sign over the half size interval $[a', b']$.

To start the bisection algorithm, we need an initial interval $[a_0, b_0]$ over which f changes sign. Running the bisection procedure then produces intervals $[a_k, b_k]$ whose size decreases at an exponential rate:

$$|b_k - a_k| = 2^{-k} |b_0 - a_0| \ .$$

To get a feeling for the convergence rate, use the approximate formula $2^{10} = 10^3$. This tells us that we get three decimal digits of accuracy for each ten iterations. This may seem good, but Newton's method is much faster, when it works. Moreover, Newton's method generalizes to more than one dimension while there is no useful multidimensional analogue of bisection search. Exponential convergence often is called *linear convergence* because of the linear relationship $|b_{k+1} - a_{k+1}| = \frac{1}{2} |b_k - a_k|$. Newton's method is faster than this.

Although the bisection algorithm is robust, it can fail if the computed approximation to $f(x)$ has the wrong sign. If f is not evaluated exactly, the computed approximation may not be continuous on a fine scale. A bisection code should take this possibility into account, either by refusing to bisect beyond a certain point, or by checking for consistency among the reported signs of f, or by making explicit use of an error estimate for computed f values.

[3] The interval notation $[a, b]$ used here is not intended to imply that $a < b$. For example, the interval $[5, 2]$ consists of all numbers between 5 and 2, endpoints included.

6.2.2 Newton's method for a nonlinear equation

As in the previous section, we want to find a value, x_*, that solves a single nonlinear equation $f(x_*) = 0$. We have procedures that return $f(x)$ and $f'(x)$ for any given x. At each iteration, we have a current iterate, $\overline{x}$ and we want to find an x' that is closer to x_*. Suppose that $\overline{x}$ is close to x_*. The values $f(\overline{x})$ and $f'(\overline{x})$ determine the tangent line to the graph of $f(x)$ at the point $\overline{x}$. The new iterate, x', is the point where this tangent line crosses the x axis. If $f(x)$ is close to zero, then x' should be close to $\overline{x}$ and the tangent line approximation should be close to f at x', which suggests that $f(x')$ should be small.

More analytically, the tangent line approximation (See Section 3.1) is

$$f(x) \approx F^{(1)}(x) = f(\overline{x}) + f'(\overline{x}) \cdot (x - \overline{x}) \ . \tag{6.1}$$

Finding where the line crosses the x axis is the same as setting $F^{(1)}(x) = 0$ and solving for x':

$$x' = \overline{x} - f'(\overline{x})^{-1} f(\overline{x}) \ . \tag{6.2}$$

This is the basic Newton method.

The local convergence rate of Newton's method is governed by the error in the approximation (6.1). The analysis assumes that the root x_* is *non-degenerate*, which means that $f'(x_*) \neq 0$. The convergence for degenerate roots is different, see Exercise 1. For a non-degenerate root, we will have $f'(\overline{x}) \neq 0$ for $\overline{x}$ close enough to x_*. Assuming this, (6.2) implies that $|x' - \overline{x}| = O(|f(\overline{x})|)$. This, together with the Taylor series error bound

$$f(x') - F^{(1)}(x') = O\left(|x' - \overline{x}|^2 \right) \quad ,$$

and the *Newton equation* $F^{(1)}(x') = 0$, implies that

$$|f(x')| = O\left(|f(\overline{x})|^2 \right) \ .$$

This means that there is a $C > 0$ so that

$$|f(x')| \leq C \cdot |f(\overline{x})|^2 \ . \tag{6.3}$$

This manifestation of *local quadratic convergence* says that the residual at the next iteration is roughly proportional[4] to the square of the residual at the current iterate.

Quadratic convergence is very fast. In a typical problem, once $x_k - x_*$ is moderately small, the residual will be at roundoff levels in a few more iterations. For example, suppose that[5] $C = 1$ in (6.3) and that $|x_k - x_*| = .1$. Then $|x_{k+1} - x_*| \leq .01$, $|x_{k+2} - x_*| \leq 10^{-4}$, and $|x_{k+4} - x_*| \leq 10^{-16}$. The number

[4] Strictly speaking, (6.3) is just a bound, not an estimate. However, Exercise 2 shows that $f(x')$ really is approximately proportional to $f(\overline{x})^2$.

[5] This does not make sense on dimensional grounds. It would be more accurate and more cumbersome to describe this stuff in terms of relative error.

of correct digits doubles at each iteration. By contrast, a *linearly convergent* iteration with $|x_{k+1} - x_*| \leq .1 \cdot |x_k - x_*|$ gains one digit of accuracy per iteration and takes 15 iterations rather than 4 to go from .1 to 10^{-16} Bisection search needs about 50 iterations to reduce the error by a factor of 10^{15} ($10^{15} = \left(10^3\right)^5 \approx \left(2^{10}\right)^5 = 2^{50}$).

Unfortunately, the quadratic convergence of Newton's method is local. There is no guarantee that $x_k \to x_*$ as $k \to \infty$ if the initial guess is not close to x_*. A program for finding x_* must take this possibility into account. See Section 3.7.3 for some ideas on how to do this.

6.3 Newton's method in more than one dimension

Newton's method applies also to solving systems of nonlinear equations. The linear approximation (6.1) applies in dimensions $n > 1$ if $f'(\overline{x})$ is the Jacobian matrix evaluated at $\overline{x}$, and f and $(x - \overline{x})$ are column vectors. We write the *Newton step* as $x' - \overline{x} = z$, so $x' = \overline{x} + z$. Newton's method determines z by replacing the nonlinear equations, $f(\overline{x} + z) = 0$, with the linear approximation,

$$0 = f(\overline{x}) + f'(\overline{x})z \; . \tag{6.4}$$

To carry out one step of Newton's method, we must evaluate the function $f(\overline{x})$, the Jacobian, $f'(\overline{x})$, then solve the linear system of equations (6.4). We may write this as

$$z = -\left(f'(\overline{x})\right)^{-1} f(\overline{x}) \; , \tag{6.5}$$

which is a natural generalization of the one dimensional formula (6.2). In computational practice (see Exercise 1) it usually is more expensive to form $\left(f')\overline{x})\right)^{-1}$ than to solve (6.4).

Newton's method for systems of equations also has quadratic (very fast) local convergence to a non-degenerate solution x_*. As for the one dimensional case, this is because of the error bound in the linear approximation (6.1). For $n > 1$, we write the Taylor approximation error bound in terms of norms:

$$\left\| f(\overline{x} + z) - F^{(1)} \right\| = \left\| f(\overline{x} + z) - \left\{ \; f(\overline{x}) + f'(\overline{x})z \; \right\} \right\| = O\left(\|z\|^2 \right) \; .$$

We see from (6.5) that[6]

$$\|z\| \leq C \, \|f(\overline{x})\| \; .$$

Together, these inequalities imply that if $\overline{x} - x_*$ is small enough then

$$\|f(x')\| = \|f(\overline{x} + z)\| \leq C \, \|f(\overline{x})\|^2 \; ,$$

[6]The definition of a non-degenerate solution is that $f'(x_*)$ is nonsingular. If $\overline{x}$ is close enough to x_*, then $f'(\overline{x})$ will be close enough to $f'(x_*)$ that it also will be nonsingular (See (4.17)). Therefore $\|z\| \leq \left\| \left(f'(\overline{x})\right)^{-1} \right\| \|f(\overline{x})\| \leq C \, \|f(\overline{x})\|$.

which is quadratic convergence, exactly as in the one dimensional case.

In practice, Newton's method can be frustrating for its lack of robustness. The user may need some ingenuity to find an x_0 close enough to x_* to get convergence. In fact, it often is hard to know whether a system of nonlinear equations has a solution at all. There is nothing as useful as the intermediate value theorem from the one dimensional case, and there is no multi-dimensional analogue of the robust but slow bisection method in one dimension.

While Newton's method can suffer from extreme ill conditioning, it has a certain robustness against ill conditioning that comes from its *affine invariance*. Affine invariance states Newton's method is invariant under *affine transformations*. An affine transformation is a mapping $x \rightarrow Ax + b$ (it would be linear without the b). An affine transformation[7] of $f(x)$ is $g(y) = Af(By)$, where A and B are invertible $n \times n$ matrices. The affine invariance is that if we start from corresponding initial guesses: $x_0 = By_0$, and create iterates y_k by applying Newton's method to $g(y)$ and x_k by applying Newton's method to $f(x)$, then the iterates also correspond: $x_k = By_k$. This means that Newton's method works exactly as well on the original equations $f(x) = 0$ as on the transformed equations $g(y) = 0$. For example, we can imagine changing from x to variables y in order to give each of the unknowns the same units. If $g(y) = 0$ is the best possible rescaling of the original equations $f(x) = 0$, then applying Newton's method to $f(x) = 0$ gives equivalent iterates.

This argument can be restated informally as saying that Newton's method makes sense on dimensional grounds and therefore is natural. The variables $x_1, \ldots, x_n$ may have different units, as may the functions $f_1, \ldots, f_n$. The n^2 entries $f'(x)$ all may have different units, as may the entries of $(f')^{-1}$. The matrix vector product that determines the components of the Newton step (see (6.4)), $z = -(f')^{-1} f(\overline{x})$, involves adding a number of contributions (entries in the matrix $(f')^{-1}$ multiplying components of f) that might seem likely to have a variety of units. Nevertheless, each of the n terms in the sum implicit in the matrix-vector product (6.5) defining a component z_j has the same units as the corresponding component, x_j. See Section 6.6 for a more detailed discussion of this point.

6.3.1 Quasi-Newton methods

Local quadratic convergence is the incentive for evaluating the Jacobian matrix. Evaluating the Jacobian matrix may not be so expensive if much of the work in evaluating f can be re-used in calculating f'. There are other situations where the Jacobian is nearly impossible to evaluate analytically. One possibility would be to estimate f' using finite differences. Column k of $f'(\overline{x})$ is $\partial_{x_k} f(\overline{x})$. The cheapest and least accurate approximation to this is the first-order one-sided difference formula (e_k is the unit vector in the x_k direction and Δx_k is a step size in that direction. Different components of x may have different units and

[7] Actually, this is a linear transformation. It is traditional to call it affine though the constant terms are missing.

therefore require different step sizes.): $(f(\overline{x}+\Delta x_k e_x) - f(\overline{x}))/\Delta x_k$. Evaluating all of f' in this way would take n extra evaluations of f per iteration, which may be so expensive that it outweighs the fast local convergence.

Quasi Newton methods replace the true $f'(\overline{x})$ in the Newton equations (6.4) by estimates of $f'(\overline{x})$ built up from function values over a sequence of iterations. If we call this approximate Jacobian A_k, the quasi Newton equations are

$$0 = f(x_k) + A_k z_k \; . \tag{6.6}$$

The simplest such method is the *secant method* for one-dimensional root finding. Using the current x_k and $f(x_k)$, and the previous x_{k-1} and $f(x_{k-1})$, We use the slope of the line connecting the current $(x_k, f(x_k))$ to the previous $(x_{k-1}, f(x_{k-1}))$ to estimate the slope of the tangent line at x_k. The result is

$$A_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \; , \quad x_{k+1} = x_k - f(x_k)/A_k \; . \tag{6.7}$$

The local convergence rate of the secant method (6.7) is better than linear ($|x_{k+1} - x_*| \le C\,|x_k - x_*|$) and worse than quadratic.

Most multidimensional quasi-Newton methods work by updating A_k at each iteration so that $A_{k+1} z_k = f(x_{k+1}) - f(x_k)$. In multidimensions, this does not determine A_{k+1} completely because it represents n equations for the n^2 elements of A_{k+1}. The references give several suggestions for update formulas. The good ones have the property that if you you apply them to linear equations, you find the exact $A = f'$ in n steps. It is not clear that such a property makes quasi Newton methods better than ordinary Newton's method with finite difference approximations to the elements of the Jacobian.

6.4 One variable optimization

Suppose $n = 1$ and we wish to find the minimum of the function of a single variable, $V(x)$. Please bear with the following long list of definitions. We say that x_* is a *local minimum* of V if there is an $R > 0$ so that $V(x_*) \le V(x)$ whenever $|x - x_*| \le R$. We say that x_* is a *strict* local minimum if $V(x) > V(x_*)$ whenever $x \ne x_*$ and $|x - x_*| \le R$. We say that x_* is a *global* minimum if $V(x_*) \le V(x)$ for all x for which $V(x)$ is defined, and a strict global minimum if $V(x_*) < V(x)$ for all $x \ne x_*$ for which V is defined. Finally, x_* is a *nondegenerate* local minimum if $V''(x_*) > 0$. The Taylor series remainder theorem implies that if $V'(x_*) = 0$ and $V''(x_*) > 0$, then x_* is at least a strict local minimum. The function $V(x)$ is *convex* if [8] $\alpha V(x) + \beta V(y) > V(\alpha x + \beta y)$ whenever $\alpha \ge 0$, $\beta \ge 0$, and $\alpha + \beta = 1$. The function is *strictly convex* if $V''(x) > 0$ for all x. A strictly convex function is convex, but the function $V(x) = x^4$ is not strictly convex, because $V''(0) = 0$. This function has a strict but degenerate global minimum at $x_* = 0$.

[8] The reader should check that this is the same as the geometric condition that the line segment connecting the points $(x, V(x))$ and $(y, V(y))$ lies above the graph of V.

For the curious, there is an analogue of bisection search in one variable optimization called *golden section search*. It applies to any continuous function that is *unimodal*, meaning that V has a single global minimum and no local minima. The *golden mean*[9] is $r = (1+\sqrt{5})/2 \approx 1.62$. At each stage of bisection search we have an interval $[\overline{a}, \overline{b}]$ in which there must be at least one root. At each stage of golden section search we have an interval $[\overline{a}, \overline{c}]$ and a third point $\overline{b} \in [\overline{a}, \overline{c}]$ with

$$|\overline{a} - \overline{c}| = r \left|\overline{a} - \overline{b}\right| \ . \tag{6.8}$$

As with our discussion of bisection search, the notation $[\overline{a}, \overline{c}]$ does not imply that $\overline{a} < \overline{c}$. In bisection, we assume that $f(\overline{a}) \cdot f(\overline{b}) < 0$. Here, we assume that $f(\overline{b}) < f(\overline{a})$ and $f(\overline{b}) < f(\overline{c})$, so that there must be a local minimum within $[\overline{a}, \overline{c}]$. Now (this is the clever part), consider a fourth point in the larger subinterval, $d = (1 - \frac{1}{r})\overline{a} + \frac{1}{r}\overline{b}$. Evaluate $f(\overline{d})$. If $f(\overline{d}) < f(\overline{b})$, take $a' = \overline{a}$, $b' = \overline{d}$, and $c' = \overline{b}$. Otherwise, take $a' = \overline{c}$, $b' = \overline{b}$, and $c' = \overline{d}$, reversing the sense of the interval. In either case, $|a' - c'| = r\,|a' - b'|$, and $f(b') < f(a')$ and $f(b') < f(c')$, so the iteration can continue. Each stage reduces the uncertainty in the minimizer by a factor of $\frac{1}{r}$, since $|a' - c'| = \frac{1}{r}\,|\overline{a} - \overline{c}|$.

6.5 Newton's method for local optimization

Most of the properties listed in Section 6.4 are the same for multi-variable optimization. We denote the gradient as $g(x) = \nabla V(x)$, and the Hessian matrix of second partials as $H(x)$. An x_* with $g(x_*) = 0$ and $H(x_*)$ positive definite (see Section 5.3 and Exercise 2) is called non-degenerate, a natural generalization of the condition V'' for one variable problems. Such a point is at least a local minimum because the Taylor series with error bound is

$$V(x_* + z) - V(x_*) = \frac{1}{2} z^* H(x_*) z + O(\|z\|^3) \ .$$

Exercise 2 shows that the first term on the right is positive and larger than the second if $H(x_*)$ is positive definite and $\|z\|$ is small enough. If $H(x_*)$ is negative definite (obvious definition), the same argument shows that x_* is at least a local maximum. If $H(x_*)$ has some positive and some negative eigenvalues (and $g(x_*) = 0$) then x_* is neither a local minimum nor a local maximum, but is called a *saddle point*. In any case, a local minimum must satisfy $g(x_*) = 0$ if V is differentiable.

We can use Newton's method from Section 6.3 to seek a local minimum by solving the equations $g(x) = 0$, but we must pay attention to the difference between row and column vectors. We have been considering x, the Newton step, z, etc. to be column vectors while $\nabla V(x) = g(x)$ is a row vector. For

[9]This number comes up in many ways. From fibonacci numbers it is $r = \lim_{k\to\infty} f_{k+1}/f_k$. If $(\alpha+\beta)/\alpha = \alpha/\beta$ and $\alpha > \beta$, then $\alpha/\beta = r$. This has the geometric intertretation that if we remove an $\alpha \times \alpha$ square from one end of an $\alpha \times (\alpha+\beta)$ rectangle, then the remaining smaller $\beta \times \alpha$ rectangle has the same aspect ratio as the original $\alpha \times (\alpha+\beta)$ rectangle. Either of these leads to the equation $r^2 = r + 1$.

this reason, we consider applying Newton's method to the column vector of equations $g^*(x) = 0$. The Jacobian matrix of the column vector function $g^*(x)$ is the Hessian H (check this). Therefore, the locally convergent Newton method is

$$x' = \overline{x} + z \ ,$$

where the step z is given by the Newton equations

$$H(\overline{x})z = -g^*(\overline{x}) \ . \tag{6.9}$$

Because it is a special case of Newton's method, it has local quadratic convergence to x_* if x_* is a local non-degenerate local minimum.

Another point of view for the local Newton method is that each iteration minimizes a *quadratic model* of the function $V(\overline{x} + z)$. The three term Taylor series approximation to V about $\overline{x}$ is

$$V(\overline{x} + z) \approx V^{(2)}(\overline{x}, z) = V(\overline{x}) + \nabla V(\overline{x})z + \frac{1}{2}z^* H(\overline{x})z \ . \tag{6.10}$$

If we minimize $V^{(2)}(\overline{x}, z)$ over z, the result is $z = -H(\overline{x})^{-1}\nabla V(\overline{x})^*$, which is the same as (6.9). As for Newton's method for nonlinear equations, the intuition is that $V^{(2)}(\overline{x}, z)$ will be close to $V(\overline{x} + z)$ for small z. This should make the minimum of $V^{(2)}(\overline{x}, z)$ close to the minimizer of V, which is x_*.

Unfortunately, this simple local method cannot distinguish between between a local minimum, a local maximum, or even a saddle point. If x_* has $\nabla V(x_*) = 0$ (so x_* is a *stationary point*) and $H(x_*)$ is nonsingular, then the iterates $x_{k+1} = x_k - H(x_k)^{-1}g^*(x_k)$ will happily converge to to x_* if $\|x_0 - x_*\|$ is small enough. This could be a local maximum or a saddle point. Moreover, if $\|x_0 - x_*\|$ is not small, we have no idea whether the iterates will converge to anything at all.

The main difference between the unsafeguarded Newton method optimization problem and general systems of nonlinear equations is that the Hessian is symmetric and (close enough to a non-degenerate local minimum) positive definite. The Jacobian f' need not be symmetric. The Choleski decomposition requires about storage for the roughly $\frac{1}{2}n^2$ distinct elements of H and about $\frac{1}{6}n^3$ floating points to compute L. This is about half the storage and work required for a general non-symmetric linear system using the LU factorization.

6.6 Safeguards and global optimization

The real difference between minimization and general systems of equations comes from the possibility of evaluating $V(x)$ and forcing it to decrease from iteration to iteration. It is remarkable that two simple *safeguards* turn the unreliable Newton's method into a much more robust (though not perfect) method that converges to a local minimum from almost any initial guess. These are (i) finding a *descent direction* by modifying $H(\overline{x})$ if necessary, and (ii) using a one dimensional *line search* to prevent wild steps. Both of the safeguards have the purpose of guaranteeing descent, that $V(x') < V(\overline{x})$.

In principle, this would allow the x_k to converge to a saddle point, but this is extremely unlikely in practice because saddle points are unstable for this process.

The safeguarded methods use the formulation of the *search directions*, p and the *step size*, $t > 0$. One iteration will take the form $x' = \overline{x} + z$, where the step is $z = tp$. We define the search direction to be a *descent direction* if

$$\frac{d}{dt} V(\overline{x} + tp)\Big|_{t=0} = \quad g(\overline{x}) \cdot p < 0 \; . \tag{6.11}$$

This guarantees that if $t > 0$ is small enough, then $V(\overline{x} + tp) < V(\overline{x})$. Then we find a step size, t, that actually achieves this property. If we prevent t from becoming too small, it will be impossible for the iterates to converge except to a stationary point.

We find the search direction by solving a modified Newton equation

$$\widetilde{H} p = -g^*(\overline{x}) \; . \tag{6.12}$$

Putting this into (6.11) gives

$$\frac{d}{dt} V(\overline{x} + tp)\Big|_{t=0} = -g(\overline{x}) \widetilde{H} g(\overline{x})^* \; .$$

This is negative if $\widetilde{H}$ is positive definite (the right hand side is a 1×1 matrix (a number) because g is a row vector). One algorithm for finding a descent direction would be to apply the Choleski decomposition algorithm (see Section 5.3). If the algorithm finds L with $LL^* = H(\overline{x})$, use this L to solve the Newton equation (6.12) with $\widetilde{H} = H(\overline{x}) = LL^*$. If the Choleski algorithm fails to find L, then $H(\overline{x})$ is not positive definite. A possible substitute (but poor in practice, see below) is $\widetilde{H} = I$, which turns Newton's method into *gradient descent*.

A better choice for $\widetilde{H}$ comes from the *modified Choleski* algorithm. This simply replaces the equation

$$l_{kk} = \left(H_{kk} - l_{k1}^2 + \cdots + l_{k,k-1}^2\right)^{1/2}$$

with the modified equation using the absolute value

$$l_{kk} = \left|H_{kk} - l_{k1}^2 + \cdots + l_{k,k-1}^2\right|^{1/2} \; . \tag{6.13}$$

Here, H_{kk} is the (k,k) entry of $H(\overline{x})$. This modified Choleski algorithm produces L with $LL^* = H(\overline{x})$ if and only if $H(\overline{x})$ is positive definite. In any case, we take $\widetilde{H} = LL^*$, which is positive definite. Using these non-Choleski factors, the Newton equations become:

$$LL^* p = -g(\overline{x})^* \; . \tag{6.14}$$

It is not entirely clear why the more complicated modified Choleski algorithm is more effective than simply taking $\widetilde{H} = I$ when $H(\overline{x})$ is not positive definite.

One possible explanation has to do with units. Let us suppose that U_k represents the units of x_k, such as seconds, dollars, kilograms, etc. Let us also suppose that $V(x)$ is dimensionless. In this case the units of $H_{jk} = \partial_{x_j}\partial_{x_k}V$ are $[H_{jk}] = 1/U_jU_k$. We can verify by studying the Choleski decomposition equations from Section **??** that the entries of L have units $[l_{jk}] = 1/U_j$, whether we use the actual equations or the modification (6.13). We solve (**??**) in two stages, first $Lq = -\nabla V^*$, then $L^*p = q$. Looking at units, it is clear that all the elements of q are dimensionless and that the elements of p have units $[p_k] = U_k$. Thus, the modified Choleski algorithm produces a search direction that component by component has the same units as x. This allows the update formula $x' = \overline{x} + tp$ to make sense with a dimensionless t. The reader should check that the choice $\widetilde{H} = I$ does not have this property in general, even if we allow t to have units, if the U_k are different.

The second safeguard is a limited *line search*. In general, line search means minimizing the function $\phi(t) = V(x + tp)$ over the single variable t. This could be done using golden section search, but a much more rudimentary binary search process suffices as a safeguard. In this binary search, we evaluate $\phi(0) = V(\overline{x})$ and $\phi(1) = V(\overline{x} + p)$. If $\phi(1) > \phi(0)$, the step size is too large. In that case, we keep reducing t by a factor of 2 (`t = t/2;`) until $\phi(t) < \phi(0)$, or we give up. If p is a search direction, we will have $\phi(t) < \phi(0)$ for small enough t and this bisection process will halt after finitely many reductions of t. If $\phi(1) < \phi(0)$, we enter a greedy process of increasing t by factors of 2 until $\phi(2t) > \phi(t)$. This process will halt after finitely many doublings if the set of x with $V(x) < V(\overline{x})$ is bounded.

A desirable feature is that the safeguarded algorithm gives the ordinary Newton step, and rapid (quadratic) local convergence, if $\overline{x}$ is close enough to a nondegenerate local minimum. The modified Hessian will correspond to the actual Hessian if $H(x_*)$ is positive definite and $\overline{x}$ is close enough to x_*. The step size will be the default $t = 1$ if $\overline{x}$ is close enough to x_* because the quadratic model (6.10) will be accurate. The quadratic model has $V^{(2)}(\overline{x}, 2z) > V^{(2)}(\overline{x}, z)$, because z is the minimizer of $V^{(2)}$.

6.7 Gradient descent and iterative methods

The *gradient descent* optimization algorithm uses the identity matrix as the approximate Hessian, $\widetilde{H} = I$, so the (negative of the) gradient becomes the search direction: $p = -\nabla V(\overline{x})^*$. This seems to make sense geometrically, as the negative gradient is the steepest downhill direction (leading to the name *method of steepest descent*). With a proper line search, gradient descent has the theoretical global robustness properties of the more sophisticated Newton method with the modified Choleski approximate Hessian. But much of the research in sophisticated optimization methods is motivated by the fact that simple gradient descent converges slowly in many applications.

One indication of trouble with gradient descent is that the formula,

$$x' = \overline{x} - t\nabla V(\overline{x})^* \; , \tag{6.15}$$

does not make dimensional sense in general, see Section 6.6. Written in components, (6.15) is $x'_k = \overline{x}_k - t\partial_{x_k} V(\overline{x})$. Applied for $k = 1$, this makes dimensional sense if the units of t satisfy $[t] = [x_1^2]/[V]$. If the units of x_2 are different from those of x_1, the x_2 equation forces units of t inconsistent with those from the x_1 equation.

We can understand the slow convergence of gradient descent by studying how it works on the model problem $V(x) = \frac{1}{2}x^* Hx$, with a symmetric and positive definite H. This is the same as assuming that the local minimum is nondegenerate and the local approximation (6.10) is exact[10]. In this case the gradient satisfies $g(x)^* = \nabla V(x)^* = Hx$, so solving the Newton equations (6.9) gives the exact solution in one iteration. We study the gradient method with a fixed step size[11], t, which implies $x_{k+1} = x_k - tHx_k$. We write this as

$$x_{k+1} = Mx_k \ , \tag{6.16}$$

where

$$M = I - tH \ . \tag{6.17}$$

The convergence rate of gradient descent in this case is the rate at which $x_k \to 0$ in the iteration (6.16).

This, in turn, is related to the eigenvalues of M. Since H and M are symmetric, we may choose an orthonormal basis in which both are diagonal: $H = \text{diag}(\lambda_1, \ldots \lambda_n)$, and $M = \text{diag}(\mu_1, \ldots \mu_n)$. The λ_j and positive, so we may assume that $0 < \lambda_{\min} = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n = \lambda_{\max}$ The formula (6.17) implies that

$$\mu_j = 1 - t\lambda_j \ . \tag{6.18}$$

After k iterations of (6.16), we have $x_{kj} = \mu_j^k x_{0j}$, where x_{kj} component j of the iterate x_k. Clearly, the rate at which $x_k \to 0$ depends on the *spectral gap*,

$$\rho = 1 - \max_j |\mu_j\| \ ,$$

in the sense that the estimate

$$\|x_k\| \leq (1-\rho)^k \|x_0\|$$

is sharp (take $x_0 = e_1$ or $x_0 = e_n$). The optimal step size, t is the one that maximizes ρ, which leads to (see (6.18)

$$\begin{aligned} 1 - \rho &= \mu_{\max} = 1 - t\lambda_{\min} \\ \rho - 1 &= \mu_{\min} = 1 - t\lambda_{\max} \ . \end{aligned}$$

Solving these gives the optimizing value $t = 2/(\lambda_{\min} + \lambda_{\min})$ and

$$\rho = 2 \cdot \frac{\lambda_{\min}}{\lambda_{\max}} \approx \frac{2}{\kappa(H)} \ . \tag{6.19}$$

[10]We simplified the problem but not lost generality by taking $x_* = 0$ here.

[11]See Exercise 7 for an example showing that line search does not improve the situation very much.

If we take $k = 2/\rho \approx \kappa(H)$ iterations, and H is ill conditioned so that k is large, the error is reduced roughly by a factor of

$$(1-\rho)^k = \left(1 - \frac{2}{k}\right)^k \approx e^{-2} \; .$$

This justifies what we said in the Introduction, that it takes $k = \kappa(H)$ iterations to reduce the error by a fixed factor.

6.7.1 Gauss Seidel iteration

The Gauss-Seidel iteration strategy makes use of the fact that optimizing over one variable is easier than optimizing over several. It goes from $\overline{x}$ to x' in n steps. Step j optimizes $V(x)$ over the single component x_j with all other components fixed. We write the n intermediate stages as $x^{(j)}$ with components $x^{(j)} = (x_1^{(j)}, \ldots, x_n^{(j)})$. Starting with $x^{(0)} = \overline{x}$, we go from $x^{(j-1)}$ to $x^{(j)}$ by optimizing over component j. That is $x_m^{(j-1)} = x_m^{(j)}$ if $m \neq j$, and we get $x_j^{(j)}$ by solving

$$\min_{\xi} V\big((x_1^{(j-1)}, \ldots, x_{j-1}^{(j-1)}, \xi, x_{j+1}^{(j-1)}, \ldots, x_n^{(j-1)})\big) \; .$$

6.8 Resources and further reading

The book by Ortega and Reinbolt has a more detailed discussion of Newton's method for solving systems of nonlinear equations. The book*Practical Optimization* by Phillip Gill, Walter Murray, and my colleague Margaret Wright, has much more on nonlinear optimization including methods for constrained optimization problems. There is much public domain software for smooth optimization problems, but I don't think much of it is useful.

The set of initial guesses x_0 so that $k_k \to x_*$ as $t \to \infty$ is the *basin of attraction* of x_*. If the method is locally convergent, the basin of attraction contains a ball of radius R about x_*. The boundary of the basin of attraction can be a beatiful fractal set. The picture book *Fractals* by Benoit Mandelbrot, some of the most attractive fractals arise in this way.

6.9 Exercises

1. Study the convergence of Newton's method applied to solving the equation $f(x) = x^2 = 0$. Show that the root $x_* = 0$ is *degenerate* in that $f'(x_*) = 0$. The Newton iterates are x_k satisfying $x_{k+1} = x_k - f(x_k)/f'(x_k)$. Show that the local convergence in this case is *linear*, which means that there is an $\alpha < 1$ with $|x_{k+1} - x_*| \approx \alpha\, |x_k - x_*|$. Note that so called linear convergence still implies that $x_k - x_* \to 0$ exponentially. Nevertheless, contrast this linear local convergence with the quadratic local convergence for a nondegenerate problem.

2. Use the Taylor expansion to second order to derive the approximation

$$f(x') \approx C(\overline{x})f(\overline{x})^2 = \frac{1}{2}\frac{f''(\overline{x})}{f'(\overline{x})^2} \cdot f(\overline{x})^2 \ . \tag{6.20}$$

   Derive a similar expression that shows that $(x' - x_*)$ is approproximately proportional to $(\overline{x} - x_*)^2$. Use (6.20) to predict that applying Newton's method to finding solving the equation $\sin(x) = 0$ will have superquadratic convergence. What makes $C(x)$ large, and the convergence slow, is (i) small $f'(x)$ (a nearly degenerate problem), and (ii) large $f''(x)$ (a highly nonlinear problem).

3. The function $f(x) = x/\sqrt{1+x^2}$ has a unique root: $f(x) = 0$ only for $x = 0$. Show that the unsafeguarded Newton method gives $x_{k+1} = x_k^3$. Conclude that the method succeeds if and only if $|x_0| < 1$. Draw graphs to illustrate the first few iterates when $x_0 = .5$ and $x_0 = 1.5$. Note that Newton's method for this problem has local cubic convergence, which is even faster than the more typical local quadratic convergence. The formula (6.20) explains why.

4. Suppose $n = 2$ and x_1 has units of (electric) charge, x_2 has units of mass, $f_1(x_1, x_2)$ has units of length, and $f_2(x_1, x_2)$ has units of time. Find the units of each of the four entries of $(f')^{-1}$. Verify the claims about the units of the step, z, at the end of Section 6.3.

5. Suppose x_* satisfies $f(x_*) = 0$. The *basin of attraction* of x_* is the set of x so that if $x_0 = x$ then $x_k \to x_*$ as $k \to \infty$. If $f'(x_*)$ is non-singular, the basin of attraction of x_* under unsafeguarded Newton's method includes at least a neighborhood of x_*, because Newton's method is locally convergent. Exercise 3 has an example in one dimension where the basin of attraction of $x_* = 0$ is the open interval (endpoints not included) $(-1, 1)$. Now consider the two dimensional problem of finding roots of $f(z) = z^2 - 1$, where $z = x+iy$. Written out in its real components, $f(x, y) = (x^2 - y^2 - 1, 2xy)$. The basin of attraction of the solution $z_* = 1$ $((x_*, y_*) = (1, 0))$ includes a neighborhood of $z = 1$ but surprisingly many many other points in the complex plane. This *Mandlebrot set* is one of the most beautiful examples of a two dimensional fractal. The purpose of this exercise is to make a pretty picture, not to learn about scientific computing.

   (a) Show that Newton iteration is $z_{k+1} = z_k - \frac{z_k^2 - 1}{2z_k}$.

   (b) Set $z_k = 1 + w_k$ and show that $w_{k+1} = \frac{3}{2}w_k^2/(1 + w_k)$.

   (c) Use this to show that if $|w_k| < \frac{1}{4}$, then $|w_{k+1}| < \frac{1}{2}\,|w_k|$. Hint: Show $|1 + w_k| > \frac{3}{4}$. Argue that this implies that the basin of attraction of $z_* = 1$ includes at least a disk of radius $\frac{1}{4}$ about z_*, which is a quantitative form of local convergence.

   (d) Show that if $|z_k - 1| < \frac{1}{4}$ for some k, then z_0 is in the basin of attraction of $z_* = 1$. (This is the point of parts (b) and (c).)

(e) Use part (d) to make a picture of the Mandlebrot set. Hint: Divide the rectangle $|x| < R_x$, $0 \leq y \leq R_y$ into a regular grid of small cells of size $\Delta x \times \Delta y$. Start Newton's method from the center of each cell. Color the cell if $|z_k - 1| < \frac{1}{4}$ for some $k \leq N$. See how the picture depends on the parameters Δx, Δy, R_x, R_y, and N.

6. A *saddle point*[12] is an x so that $\nabla V(x) = 0$ and the Hessian, $H(x)$, is nonsingular and has at least one negative eigenvalue. We do not want the iterates to converge to a saddle point, but most Newton type optimization algorithms seem to have that potential. All the safeguarded optimization methods we discussed have $\Phi(x) = x$ if x is a saddle point because they all find the search direction by solving $\widetilde{H}p = -\nabla V(x)$.

   (a) Let $V(x) = x_1^2 - x_2^2$ and suppose $\overline{x}$ is on the x_1 axis. Show that with the modified Choleski, x' also is on the x_1 axis, so the iterates converge to the saddle point, $x = 0$. Hint: $\widetilde{H}$ has a simple form in this case.

   (b) Show that if $\overline{x}_2 \neq 0$, and $t > 0$ is the step size, and we use the bisection search that increases the step size until $\phi(t) = V(\overline{(x)} + tp)$ satisfies $\phi(2t) > \phi(t)$, then one of the following occurs:

      i. The bisection search does not terminate, $t \to \infty$, and $\phi(t) \to -\infty$. This would be considered good, since the minimum of V is $-\infty$.

      ii. The line search terminates with t satisfying $\phi(t) = V(x') < 0$. In this case, subsequent iterates cannot converge to $x = 0$ because that would force V to converge to zero, while our modified Newton strategy guarantees that V decreases at each iteration.

   (c) *Nonlinear least squares* means finding $x \in R^m$ to minimize $V(x) = \|f(x) - b\|_{l^2}^2$, where $f(x) = (f_1(x), \ldots, f_n(x))^*$ is a column vector of n nonlinear functions of the m unknowns, and $b \in R^n$ is a vector we are trying to approximate. If $f(x)$ is linear (there is an $n \times m$ matrix A with $f(x) = Ax$), then minimizing $V(x)$ is a linear least squares problem. The *Marquart Levenberg* iterative algorithm solves a linear least squares problem at each iteration. If the current iterate is $\overline{x}$, let the linearization of f be the $n \times m$ Jacobian matrix A with entries $a_{ij} = \partial_{x_j} f_i(\overline{x})$. Calculate the step, p, by solving

$$\min_p \|Ap - (b - f(\overline{x}))\|_{l^2} \ . \tag{6.21}$$

   Then take the next iterate to be $x' = \overline{x} + p$.

      i. Show that this algorithm has local quadratic convergence if the residual at the solution has zero residual: $r(x_*) = f(x_*) - b = 0$, but not otherwise (in general).

---

[12] The usual definition of saddle point is that H should have at least one positive and one negative eigenvalue and no zero eigenvalues. The simpler criterion here suffices for this application.

ii. Show that p is a descent direction.

iii. Describe a safeguarded algorithm that probably will converge to at least a local minimum or diverge.

7. This exercise shows a connection between the slowness of gradient descent and the condition number of H in one very special case. Consider minimizing the model quadratic function in two dimensions $V(x) = \frac{1}{2}\left(\lambda_1 x_1^2 + \lambda_2 x_2^2\right)$ using gradient descent. Suppose the line search is done exactly, choosing t to minimize $\phi(t) = V(\overline{x} + tp)$, where $p = \nabla V(\overline{x})$. In general it is hard to describe the effect of many minimization steps because the iteration $\overline{x} \to x'$ is nonlinear. Nevertheless, there is one case we can understand.

   (a) Show that for any V, gradient descent with exact line search has p_{k+1} orthogonal to p_k. Hint: otherwise, the step size t_k was not optimal.

   (b) In the two dimensional quadratic optimization problem at hand, show that if p_k is in the direction of $(-1,-1)^*$, then p_{k+1} is in the direction of $(-1,1)^*$.

   (c) Show that p_k is in the direction of $(-1,-1)^*$ if and only if $(x_1,x_2) = r(\lambda_2,\lambda_1)$, for some r,

   (d) Since the optimum is $x_* = (0,0)^*$, the error is $\|(x_1,x_2)\|$. Show that if p_0 is in the direction of $(-1,-1)$, then the error decreases exactly by a factor of $\rho = (\lambda_1 - \lambda_2)/(\lambda_1 + \lambda_2)$ if $\lambda_1 \geq \lambda_2$ (including the case $\lambda_1 = \lambda_2$).

   (e) Show that if $\lambda_1 \gg \lambda_2$, then $\rho \approx 1 - 2\lambda_2/\lambda_1 = 1 - 2/\kappa(H)$, where $\kappa(H)$ is the linear systems condition number of H.

   (f) Still supposing $\lambda_1 \gg \lambda_2$, show that it takes roughly $n = 1/\kappa(H)$ iterations to reduce the error by a factor of e^2.

8. This exercise walks you through construction of a robust optimizer. It is as much an exercise in constructing scientific software as in optimization techniques. You will apply it to finding the minimum of the two variable function

$$V(x,y) = \frac{\psi(x,y)}{\sqrt{1+\psi(x,y)^2}} \quad , \quad \psi(x,y) = \psi_0 + wx^2 + \bigl(y - a\sin(x)\bigr)^2 \; .$$

   Hand in output documenting what you for each of the of the parts below.

   (a) Write procedures that evaluate $V(x,y)$, $g(x,y)$, and $H(x,y)$ analytically. Write a tester that uses finite differences to verify that g and H are correct.

   (b) Implement a local Newton's method without safeguards as in Section 6.5. Use the Choleski decomposition code from Exercise 3. Report failure if H is not positive definite. Include a stoping criterion and a maximum iteration count, neither hard wired. Verify local quadratic convergence starting from initial guess $(x_0, y_0) = (.3, .3)$ with parameters $\psi_0 = .5$, $w = .5$, and $a = .5$. Find an initial condition from which the unsafeguarded method fails.

(c) Modify the Choleski decomposition code Exercise 5.3 to do the modified Choleski decomposition described in Section 6.6. This should require you to change a single line of code.

(d) Write a procedure that implements the limited line search strategy described in Section 6.6. This also should have a maximum iteration count that is not hard wired. Write the procedure so that it sees only a scalar function $\phi(t)$. Test on:

i. $\phi(t) = (t - .9)^2$ (should succeed with $t = 1$).

ii. $\phi(t) = (t - .01)^2$ (should succeed after several step size reductions).

iii. $\phi(t) = (t - 100)^2$ (should succeed after several step size doublings).

iv. $\phi(t) = t$ (should fail after too many step size reductions).

v. $\phi(t) = -t$ (should fail after too many doublings).

(e) Combine the procedures from parts (c) and (d) to create a robust global optimization code. Try the code on our test problem with $(x_0, y_0) = (10, 10)$ and parameters $\psi_0 = .5$, $w = .02$, and $a = 1$. Make plot that shows contour lines of V and all the iterates.

Chapter 7

Approximating Functions

Scientific computing often calls for representing or approximating a general function, $f(x)$. That is, we seek a $\widetilde{f}$ in a certain class of functions so that[1] $\widetilde{f} \approx f$ in some sense. For example, we might know the values $f_k = f(x_k)$ (for some set of points x_0, x_1, $\ldots$) and wish to find an *interpolating function* so that $f(x_k) = f_k$. In general there

This chapter discusses two related problems. One is finding simple approximate representations for known functions. The other is interpolation and extrapolation, estimating unknown function values from known values at nearby points. On one hand, interpolation of smooth functions gives accurate approximations. On the other hand, we can interpolate and extrapolate using our approximating functions.

Some useful interpolating functions are polynomials, splines, and trigonometric polynomials. Interpolation by low order polynomials is simple and ubiquitous in scientific computing. Ideas from Chapters 3 and 4 will let us understand its accuracy. Some simple tricks for polynomial interpolation are the Newton form of the interpolating polynomial and Horner's rule for evaluating polynomials.

Local polynomial interpolation gives different approximating functions in different intervals. A spline interpolant is a single globally defined function that has many of the approximation properties of local polynomial interpolation. Computing the interpolating spline from n data points requires us to solve a linear system of equations involving a symmetric banded matrix, so the work is proportional to n.

The order of accuracy of polynomial or spline interpolation is $p+1$, where p is the degree of polynomials used. This suggests that we could get very accurate approximations using high degree polynomials. Unfortunately, high degree polynomial interpolation on uniformly spaced points leads to linear systems of equations that are exponentially ill conditioned, $\kappa \sim e^{cp}$, where p is the degree and κ is the condition number. The condition number grows moderately as $p \to \infty$ only if the interpolation points cluster at the ends of the interval in a very specific way.

High accuracy interpolation on uniformly spaced points can by done using trigonometric polynomial interpolation, also called Fourier interpolation. More generally, Fourier analysis for functions defined at n uniformly spaced points can be done using the discrete Fourier transform, or DFT. The fast Fourier transform, or FFT, is an algorithm that computes the DFT of n values in $O(n\log(n))$ time. Besides trigonometric interpolation, the FFT gives highly accurate solutions to certain linear partial differential equations and allows us to compute large discrete convolutions, including the convolution that defines the time lag covariance function for a time series.

[1] In Chapters 2 and 3 we used the *hat* or *caret* notation of statisticians to denote approximation: $\widehat{Q} \approx Q$. In this chapter, the hat refers to Fourier coefficients and the tilde represents approximation.

7.1 Polynomial interpolation

Given points $x_0, \ldots, x_d$ and values $f_0, \ldots, f_d$, there is a unique *interpolating* polynomial of degree d,

$$p(x) = p_0 + p_1 x + \cdots + p_d x^d \;, \tag{7.1}$$

so that

$$p(x_k) = f_k \quad \text{for } k = 0, 1, \ldots, d. \tag{7.2}$$

We give three proofs of this, each one illustrating a different aspect of polynomial interpolation.

We distinguish between *low order* or *local* interpolation and *high order* or *global* interpolation. In low order interpolation, we have a small number (at least two and probably not much more than five) of nearby points and we seek an interpolating polynomial that should be valid near these points. This leads to approximations of order $d+1$ for degree d interpolation. For example, local linear interpolation (degree $d = 1$) is second order accurate. See Exercise **??**.

7.1.1 Vandermonde theory

The most direct approach to interpolation uses the Vandermonde matrix. The equations (7.22) and (7.2) form a set of linear equations that determine the $d+1$ unknown coefficients, p_j, from the $d+1$ given function values, f_k. The k^{th} equation is

$$p_0 + x_k p_1 + x_k^2 p_2 + \cdots + x_k^d p_d = f_k \;,$$

which we write abstractly as

$$Vp = f \;, \tag{7.3}$$

where

$$V = \begin{pmatrix} 1 & x_0 & . & . & . & x_0^d \\ 1 & x_1 & . & . & . & x_1^d \\ \vdots & \vdots & & & & \vdots \\ 1 & x_d & . & . & . & x_d^d \end{pmatrix} \;, \tag{7.4}$$

$p = (p_0, \ldots, p_d)^*$, and $f = (f_0, \ldots, f_d)^*$. The equations (7.3) have a unique solution if and only if $\det(V) \neq 0$. We show $\det(V) \neq 0$ using the following famous formula:

Theorem 2 *Define $D(x_0, \ldots, x_d) = \det(V)$ as in (7.4). Then*

$$D(x_0, \ldots, x_d) = \prod_{j<k} (x_k - x_j) \;. \tag{7.5}$$

The reader should verify directly that $D(x_0, x_1, x_2) = (x_2 - x_0)(x_1 - x_0)(x_2 - x_1)$. It is clear that $D = 0$ whenever $x_j = x_k$ for some $j \neq k$ because $x_j = x_k$ makes row j and row k equal to each other. The formula (7.5) says that D is a product of factors coming from these facts.

Proof: The proof uses three basic properties of determinants. The first is that the determinant does not change if we perform an elimination operation on rows or columns. If we subtract a multiple of row j from row k or of column j from column k, the determinant does not change. The second is that if row k or column k has a common factor, we can pull that factor out of the determinant. The third is that if the first column is $(1, 0 \ldots, 0)^*$, then the determinant is the determinant of the $d \times d$ matrix got by deleting the top row and first column.

We work by induction on the number of points. For $d = 1$ (7.5) is $D(x_0, x_1) = x_1 - x_0$, which is easy to verify. The induction step is the formula

$$D(x_0, \ldots, x_d) = \left(\prod_{k=1}^{d} (x_k - x_0) \right) \cdot D(x_1, \ldots, x_d) \; . \tag{7.6}$$

We use the easily checked formula

$$x^k - y^k = (x - y)(x^{k-1} + x^{k-2}y + \cdots + y^{k-1}) \; . \tag{7.7}$$

To compute the determinant of V in (7.4), we use Gauss elimination to set all but the top entry of the first column of V to zero. This means that we replace row j by row j minus row 1. Next we find common factors in the columns. Finally we perform column operations to put the $d \times d$ matrix back into the form of a vanderMonde matrix for $x_1, \ldots, x_d$, which will prove (7.6).

Rather than giving the argument in general, we give it for $d = 2$ and $d = 3$. The general case will be clear from this. For $d = 2$ we have

$$\begin{aligned}
\det \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} &= \det \begin{pmatrix} 1 & x_0 & x_0^2 \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 \end{pmatrix} \\
&= \det \begin{pmatrix} x_1 - x_0 & x_1^2 - x_0^2 \\ x_2 - x_0 & x_2^2 - x_0^2 \end{pmatrix} \; .
\end{aligned}$$

The formula (7.7) with $k = 2$ gives $x_1^2 - x_0^2 = (x_1 - x_0)(x_1 + x_0)$, so $(x_1 - x_0)$ is a common factor in the top row. Similarly, $(x_2 - x_0)$ is a common factor of the bottom row. Thus:

$$\begin{aligned}
\det \begin{pmatrix} x_1 - x_0 & x_1^2 - x_0^2 \\ x_2 - x_0 & x_2^2 - x_0^2 \end{pmatrix} &= \det \begin{pmatrix} x_1 - x_0 & (x_1 - x_0)(x_1 + x_0) \\ x_2 - x_0 & x_2^2 - x_0^2 \end{pmatrix} \\
&= (x_1 - x_0) \det \begin{pmatrix} 1 & (x_1 + x_0) \\ x_2 - x_0 & x_2^2 - x_0^2 \end{pmatrix} \\
&= (x_1 - x_0)(x_2 - x_0) \det \begin{pmatrix} 1 & x_1 + x_0 \\ 1 & x_2 + x_0 \end{pmatrix} \; .
\end{aligned}$$

The final step is to subtract x_0 times the first column from the second column, which does not change the determinant:

$$\det \begin{pmatrix} 1 & x_1 + x_0 \\ 1 & x_2 + x_0 \end{pmatrix} = \det \begin{pmatrix} 1 & x_1 + x_0 - x_0 * 1 \\ 1 & x_2 + x_0 - x_0 * 1 \end{pmatrix}$$

$$= \det \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix}$$
$$= D(x_1, x_2) \ .$$

This proves (7.6) for $d = 2$.

For $d = 3$ there is one more step. If we subtract row 1 from row k for $k > 1$ and do the factoring using (7.7) for $k = 2$ and $x = 3$, we get

$$\det \begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \end{pmatrix} =$$
$$(x_1 - x_0)(x_2 - x_0)(x_3 - x_0) \det \begin{pmatrix} 1 & x_1 + x_0 & x_1^2 + x_1x_0 + x_0^2 \\ 1 & x_2 + x_0 & x_2^2 + x_2x_0 + x_0^2 \\ 1 & x_3 + x_0 & x_3^2 + x_3x_0 + x_0^2 \end{pmatrix} \ .$$

We complete the proof of (7.6) in this case by showing that

$$\det \begin{pmatrix} 1 & x_1 + x_0 & x_1^2 + x_1x_0 + x_0^2 \\ 1 & x_2 + x_0 & x_2^2 + x_2x_0 + x_0^2 \\ 1 & x_3 + x_0 & x_3^2 + x_3x_0 + x_0^2 \end{pmatrix} = \det \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \ .$$

For this we first subtract x_0 times the first column from the second column, then subtract x_0^2 times the first column from the third column, then subtract x_0 times the second column from the third column. This completes the proof of Theorem 2 and shows that one can find the coefficients of the interpolating polynomial by solving a linear system of equations involving the vanderMonde matrix.

7.1.2 Newton interpolation formula

The Newton interpolation formula is a simple and insightful way to express the interpolating polynomial. It is based on repeated divided differences, done in a way to expose the leading terms of polynomials. These are combined with a specific basis for the vector space of polynomials of degree k so that in the end the interpolation property is obvious. In some sense, the Newton interpolation formula provides a formula for the inverse of the vanderMonde matrix.

We begin with the problem of estimating derivatives of $f(x)$ using a number of function values. Given nearby points, x_1 and x_0, we have

$$f' \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0} \ .$$

We know that the divided difference is particularly close to the derivative at the center of the interval, so we write

$$f'\left(\frac{x_1 + x_0}{2}\right) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0} \ , \tag{7.8}$$

with an error that is $O\left(|x_1 - x_0|^2\right)$. If we have three points that might not be uniformly spaced, the second derivative estimate using (7.8) could be

$$
\begin{aligned}
f'' &\approx \frac{f'\left(\frac{x_2+x_1}{2}\right) - f'\left(\frac{x_1+x_0}{2}\right)}{\dfrac{x_2+x_1}{2} - \dfrac{x_1+x_0}{2}} \\
f'' &\approx \frac{\dfrac{f(x_2)-f(x_1)}{x_2-x_1} - \dfrac{f(x_1)-f(x_0)}{x_1-x_0}}{\frac{1}{2}\left(x_2-x_0\right)} \quad . \qquad (7.9)
\end{aligned}
$$

As we saw in Chapter 3, the approximation (7.9) is consistent (converges to the exact answer as $x_2 \to x$, $x_1 \to x$, and $x_0 \to x$) if it is exact for quadratics, $f(x) = ax^2 + bx + c$. Some algebra shows that both sides of (7.9) are equal to $2a$.

The formula (7.9) suggests the right way to do repeated divided differences. Suppose we have $d+1$ points[2] $x_0, \ldots, x_d$, we define $f[x_k] = f(x_k)$ (exchange round parentheses for square brackets), and the first order divided difference is:

$$
f[x_k, x_{k+1}] = \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k} \quad .
$$

More generally, the *Newton divided difference* of order $k+1$ is a divided difference of divided differences of order k:

$$
f[x_j, \cdots, x_{k+1}] = \frac{f[x_{j+1}, \cdots, x_{k+1}] - f[x_j, \cdots, x_k]}{x_{k+1} - x_j} \quad . \qquad (7.10)
$$

The denominator in (7.10) is the difference between the extremal x values, as (7.9) suggests it should be. If instead of a function $f(x)$ we just have values $f_0, \ldots, f_d$, we define

$$
[f_j, \cdots, f_{k+1}] = \frac{[f_{j+1}, \cdots, f_{k+1}] - [f_j, \cdots, f_k]}{x_{k+1} - x_j} \quad . \qquad (7.11)
$$

It may be convenient to use the alternative notation

$$
D_k(f) = f[x_0, \cdots, x_k] \; .
$$

If $r(x) = r_k x^k + \cdots + r_0$ is a polynomial of degree k, we will see that $D_k r = k! \cdot r_k$. We verified this already for $k = 1$ and $k = 2$.

The interpolation problem is to find a polynomial of degree d that satisfies the interpolation conditions (7.2). The formula (7.1) expresses the interpolating polynomial as a linear combination of pure monomials x^k. Using the monomials as a basis for the vector space of polynomials of degree d leads to the vander-Monde matrix (7.4). Here we use a different basis, which might be called the

[2]The x_k must be distinct but they need not be in order. Nevertheless, it helps the intuition to think that $x_0 < x_1 < \cdots < x_d$.

Newton monomials of degree k (although they strictly speaking are not monomials), $q_0(x) = 1$, $q_1(x) = x - x_0$, $q_2(x) = (x - x_1)(x - x_0)$, and generally,

$$q_k(x) = (x - x_{k-1}) \cdot \cdots \cdot (x - x_0) \; . \tag{7.12}$$

It is easy to see that $q_k(x)$ is a polynomial of degree k in x with leading coefficient equal to one:

$$q_k(x) = x^k + a_{k-1}x^{k-1} + \cdots \quad .$$

Since this also holds for q_{k-1}, we may subtract to get:

$$q_k(x) - a_{k-1}q_{k-1}(x) = x^k + b_{k-2}x^{k-2} + \cdots \; .$$

Continuing in this way, we express x^k in terms of Newton monomials:

$$x^k = q_k(x) - a_{k,k-1}q_{k-1}(x) - b_{k,k-2} - \cdots \; . \tag{7.13}$$

This shows that the $q_k(x)$ are linearly independent and span the same space as the monomial basis.

The connection between repeated divided differences (7.10) and Newton monomials (7.12) is

$$D_k q_j = \delta_{kj} \; . \tag{7.14}$$

The intuition is that $D_k f$ plays the role $\frac{1}{k!}\partial_x^k f(0)$ and $q_j(x)$ plays the role of x_j. For $k > j$, $\partial_k x^j = 0$ because differentiation lowers the order of a monomial. For $k < j$, $\partial_x^k x^j = 0$ when evaluated at $x = 0$ because monomials vanish when $x = 0$. The remaining case is the interesting one, $\frac{1}{k!}\partial_x^k x^k = 1$.

We verify (7.14) by induction on k. We suppose that (7.14) holds for all $k < d$ and all j and use that to prove it for $k = d$ and all j, treating the cases $j = d$, $j < d$, and $j > d$ separately. The base case $k = 1$ explains the ideas. For $j = 1$ we have

$$D_1 q_1(x) = \frac{q_1(x_1) - q_1(x_0)}{x_1 - x_0} = \frac{(x_1 - x_0) - (x_0 - x_0)}{x_1 - x_0} = 1 \; , \tag{7.15}$$

as claimed. More generally, any first order divided difference of q_1 is equal to one,

$$q_1[x_{k+1}, x_k] = \frac{q_1(x_{k+1}) - q_1(x_k)}{x_{k+1} - x_k} = 1 \; ,$$

which implies that higher order divided differences of q_1 are zero. For example,

$$q_1[x_2, x_3, x_4] = \frac{q_1[x_3, x_4] - q_1[x_2, x_3]}{x_4 - x_2} = \frac{1 - 1}{x_4 - x_2} = 0 \; .$$

This proves the base case, $k = 1$ and all j.

The induction step has the same three cases. For $j > d$ it is clear that $D_d q_j = q_j[x_0, \ldots, x_d] = 0$ because $q_j(x_k) = 0$ for all the x_k that are used in $q_j[x_0, \ldots, x_d]$. The interesting case is $q_k[x_0, \ldots, x_k] = 1$. From (7.10) we have that

$$q_k[x_0, \ldots, x_k] = \frac{q_k[x_1, \ldots, x_k] - q_k[x_0, \ldots, x_{k-1}]}{x_k - x_0} = \frac{q_k[x_1, \ldots, x_k]}{x_k - x_0} \; ,$$

because $q_k[x_0, \ldots, x_{k-1}] = 0$ (it involves all zeros). The same reasoning gives $q_k[x_1, \ldots, x_{k-1}] = 0$ and

$$q_k[x_1, \ldots, x_k] = \frac{q_k[x_2, \ldots, x_k] - q_k[x_1, \ldots, x_{k-1}]}{x_k - x_1} = \frac{q_k[x_2, \ldots, x_k]}{x_k - x_1} \; .$$

Combining these gives

$$q_k[x_0, \ldots, x_k] = \frac{q_k[x_1, \ldots, x_k]}{x_k - x_0} = \frac{q_k[x_2, \ldots, x_k]}{(x_k - x_0)(x_k - x_1)} \; ,$$

and eventually, using the definition (7.12), to

$$q_k[x_0, \ldots, x_k] = \frac{q_k(x_k)}{(x_k - x_0) \cdot \cdots \cdot (x_k - x_{k-1})} = \frac{(x_k - x_0) \cdot \cdots \cdot (x_k - x_{k-1})}{(x_k - x_0) \cdot \cdots \cdot (x_k - x_{k-1})} = 1 \; ,$$

as claimed. Now, using (7.13) we see that

$$D_k x^k = D_k \big(q_k - a_{k,k-1} q_{k-1} - \cdots \big) = D_k q_k = 1 \; ,$$

for any collection of distinct points x_j. This, in turn, implies that

$$q_k[x_{m+k}, \ldots, x_m] = 1 \; .$$

which, as in (7.15), implies that $D_{k+1} q_k = 0$. This completes the induction step.

The formula (7.14) allows us to verify the *Newton interpolation formula*, which states that

$$p(x) = \sum_{k=0}^{d} [f_0, \ldots, f_k] q_k(x) \; , \tag{7.16}$$

satisfies the interpolation conditions (7.2). We see that $p(x_0) = f_0$ because each term on the right $k = 0$ vanishes when $x = x_0$. The formula (7.14) also implies that $D_1 p = D_1 f$. This involves the values $p(x_1)$ and $p(x_0)$. Since we already know $p(x_0)$ is correct, this implies that $p(x_1)$ also is correct. Continuing in this way verifies all the interpolation conditions.

7.1.3 Lagrange interpolation formula

The Lagrange approach to polynomial interpolation is simpler mathematically but less useful than the others. For each k, define the polynomial[3] of degree d

$$l_k(x) = \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)} \; . \tag{7.17}$$

[3]We do not call these *Lagrange polynomials* because that term means something else.

For example, for $d = 2$, $x_0 = 0$, $x_1 = 2$, $x_2 = 3$ we have

$$\begin{aligned}
l_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-2)(x-3)}{(-2)(-3)} = \frac{1}{6}\left(x^2-5x+6\right) \\
l_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x-0)(x-3)}{(1)(-1)} = x^2-3x \\
l_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{(x-0)(x-2)}{(3)(1)} = \frac{1}{3}\left(x^2-2x\right) \ ,
\end{aligned}$$

If $j = k$, the numerator and denominator in (7.17) are equal. If $j \neq k$, then $l(k(x_j) = 0$ because $(x_j - x_j) = 0$ is one of the factors in the numerator. Therefore

$$l_k(x_j) = \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \tag{7.18}$$

The *Lagrange interpolation formula* is

$$p(x) = \sum_{k=0}^{d} f_k l_k(x) \ . \tag{7.19}$$

The right side is a polynomial of degree d. This satisfies the interpolation conditions (7.2) because of (7.18).

7.2 Discrete Fourier transform

The Fourier transform is one of the most powerful methods of applied mathematics. Its finite dimensional anologue, the *discrete Fourier transform*, or *DFT*, is just as useful in scientific computing. The DFT allows direct algebraic solution of certain differential and integral equations. It is the basis of computations with time series data and for digital signal processing and control. It leads to computational methods that have an infinite order of accuracy (which is not the same as being exact).

The drawback of DFT based methods is their geometric inflexibility. They can be hard to apply to data that are not sampled at uniformly spaced points. The multidimensional DFT is essentially a product of one dimensional DFTs. Therefore is it hard to apply DFT methods to problems in more than one dimension unless the computational domain has a simple shape. Even in one dimension, applying the DFT depends on boundary conditions.

7.2.1 Fourier modes

The simplest Fourier analysis is for periodic functions of one variable. We say $f(x)$ is *periodic* with *period* p if $f(x+p) = f(x)$ for all x. If α is an integer, then the *Fourier mode*

$$w_\alpha(x) = e^{2\pi i \alpha x/p} \tag{7.20}$$

is such a periodic function. Fourier analysis starts with the fact that Fourier modes form a basis for the vector space of periodic functions. That means that if f has period p, then there are *Fourier coefficients*, $\widehat{f}_\alpha$, so that

$$f(x) = \sum_{\alpha=-\infty}^{\infty} \widehat{f}_\alpha e^{2\pi i\alpha x/p} \quad . \tag{7.21}$$

More precisely, let V_p be the vector space of complex valued periodic functions so that

$$\|f\|_{L^2}^2 = \int_0^p |f(x)|^2 \, dx < \infty \; .$$

This vector space has an *inner product* that has the properties discussed in Section 4.2.2. The definition is

$$\langle f, g\rangle = \int_0^p \overline{f}(x) g(x) dx \; . \tag{7.22}$$

Clearly, $\|f\|_{L^2}^2 = \langle f, f\rangle > 0$ unless $f \equiv 0$. The inner product is linear in the g variable:

$$\langle f, ag_1 + bg_2\rangle = a\langle f, g_1\rangle + b\langle f, g_2\rangle \; ,$$

and *antilinear* in the f variable:

$$\langle af_1 + bf_2, g\rangle = \overline{a}\langle f_1, g\rangle + \overline{b}\langle f_2, g\rangle \; .$$

Functions f and g are *orthogonal* if $\langle f, g\rangle = 0$. A set of functions is orthogonal if any two of them are orthogonal and none of them is zero. The Fourier modes (7.20) have this property: if $\alpha \neq \beta$ are two integers, then $\langle w_\alpha, w_\beta\rangle = 0$ (check this).

Any orthogonal system of functions is linearly independent. Linear independence means that if

$$f(x) = \sum_\alpha \widehat{f}_\alpha w_\alpha(x) = \sum_{\alpha=-\infty}^{\infty} \widehat{f}_\alpha e^{2\pi i\alpha x/p} \; , \tag{7.23}$$

then the $\widehat{f}_\alpha$ are uniquely determined. For orthogonal functions, we show this by taking the inner product of w_β with f and use the linearity of the inner product to get

$$\langle w_\beta, f\rangle = \sum_\alpha \widehat{f}_\alpha \langle w_\beta, w_\alpha\rangle = \widehat{f}_\beta \langle w_\beta, w_\beta\rangle \; ,$$

so

$$\widehat{f}_\beta = \frac{\langle w_\beta, f\rangle}{\|w_\beta\|^2} \; . \tag{7.24}$$

This formula shows that the coefficients are uniquely determined. Written more explicitly for the Fourier modes, (7.24) is

$$\widehat{f}_\alpha = \frac{1}{p} \int_{x=0}^{p} e^{-2\pi i\alpha x/p} f(x) dx \; . \tag{7.25}$$

An orthogonal family, w_α, is *complete* if any f has a representation in terms of them as a linear combination (7.23). An orthogonal family need not be complete. Fourier conjectured that the Fourier modes (7.20) are complete, but this was first proven several decades later.

Much of the usefulness of Fourier series comes from the relationship between the series for f and for derivatives f', f'', etc. When we differentiate (7.23) with respect to x and differentiate under the summation on the right side, we get

$$f'(x) = \frac{2\pi i}{p} \sum_\alpha \alpha \widehat{f}_\alpha e^{2\pi i \alpha x/p} .$$

This shows that the α Fourier coefficient of f' is

$$\widehat{f'}_\alpha = \frac{2\pi i \alpha}{p} \widehat{f}_\alpha . \tag{7.26}$$

Formulas like these allow us to express the solutions to certain ordinary and partial differential equations in terms of Fourier series. See Exercise 4 for one example.

We will see that the differentiation formula (7.26) also contains important information about the Fourier coefficients of smooth functions, that they are very small for large α. This implies that approximations

$$f(x) \approx \sum_{|\alpha| \leq R} \widehat{f}_\alpha w_\alpha(x)$$

are very accurate if f is smooth. It also implies that the DFT coefficients (see Section 7.2.2) are very accurate approximations of $\widehat{f}_\alpha$. Both of these make DFT based computational methods very attractive (when they apply) for application to problems with smooth solutions.

We start to see this decay by rewriting (7.26) as

$$\widehat{f}_\alpha = \frac{p\widehat{f'}_\alpha}{2\pi i} \cdot \frac{1}{\alpha} . \tag{7.27}$$

The integral formula

$$\widehat{f'}_\alpha = \frac{1}{p} \int_0^p \overline{w_\alpha(x)} f'(x) dx$$

shows that the Fourier coefficients $\widehat{f'}_\alpha$ are bounded if f' is bounded, since (for some real θ) $|w_\alpha(x)| = \left|e^{i\theta}\right| = 1$. This, and (7.27) shows that

$$\left|\widehat{f}_\alpha\right| \leq C \cdot \frac{1}{|\alpha|} .$$

We can go further by applying (7.27) to f' and f'' to get

$$\widehat{f}_\alpha = \frac{p^2 \widehat{f''}_\alpha}{-4\pi^2} \cdot \frac{1}{\alpha^2} ,$$

so that if f'' is bounded, then

$$\left|\widehat{f}_\alpha\right| \leq C \cdot \frac{1}{|\alpha^2|} ,$$

which is faster decay ($1/\alpha^2 \ll 1/\alpha$ for large α). Continuing in this way, we can see that if f has N bounded derivatives then

$$\left|\widehat{f}_\alpha\right| \leq C_N \cdot \frac{1}{|\alpha^N|} . \tag{7.28}$$

This shows, as we said, that the Fourier coefficients of smooth functions decay rapidly.

It is helpful in real applications to use real Fourier modes, particularly when $f(x)$ is real. The real modes are sines and cosines:

$$u_\alpha(x) = \cos(2\pi\alpha x/p) , \quad v_\alpha(x) = \sin(2\pi\alpha x/p) . \tag{7.29}$$

The u_α are defined for $\alpha \geq 0$ and the v_α for $\alpha \geq 1$. The special value $\alpha = 0$ corresponds to $u_0(x) = 1$ (and $v_0(x) = 0$). Exercise 3 shows that the u_α for $\alpha = 0, 1, \ldots$ and v_α for $\alpha = 1, 2, \ldots$ form an orthogonal family. The real Fourier series representation expresses f as a superposition of these functions:

$$f(x) = \sum_{\alpha=0}^{\infty} a_\alpha \cos(2\pi\alpha x/p) + \sum_{\alpha=1}^{\infty} b_\alpha \sin(2\pi\alpha x/p) . \tag{7.30}$$

The reasoning that led to (7.25), and the normalization formulas of Exercise **??** below, (7.41), gives

$$\left.\begin{aligned}
a_\alpha &= \frac{2}{p}\int_0^p \cos(2\pi\alpha x/p) f(x)dx \;(\alpha \geq 1), \\
b_\alpha &= \frac{2}{p}\int_0^p \sin(2\pi\alpha x/p) f(x)dx \;(\alpha \geq 1), \\
a_0 &= \frac{1}{p}\int_0^p f(x)dx .
\end{aligned}\right\} \tag{7.31}$$

This real Fourier (7.30) is basically the same as the complex Fourier series (7.23) when $f(x)$ is real. If f is real, then (7.25) shows that[4] $\widehat{f}_{-\alpha} = \overline{\widehat{f}_\alpha}$. This determines the $\alpha < 0$ Fourier coefficients from the $\alpha \geq 0$ coefficients. If $\alpha > 0$ and $\widehat{f}_\alpha = g_\alpha + ih_\alpha$ (g_α and h_α being real and imaginary parts), then (using $e^{i\theta} = \cos(\theta) + i\sin(\theta)$), we have

$$\begin{aligned}
&\widehat{f}_\alpha e^{2\pi i\alpha x/p} + \widehat{f}_{-\alpha} e^{-2\pi i\alpha x/p} \\
&= (g_\alpha + ih_\alpha)\big(\cos(\cdot) + i\sin(\cdot)\big) + (g_\alpha - ih_\alpha)\big(\cos(\cdot) - i\sin(\cdot)\big) \\
&= 2g_\alpha\cos(\cdot) - 2h_\alpha\sin(\cdot) .
\end{aligned}$$

[4]This also shows that the full Fourier series sum over positive and negative α is somewhat redundant for real f. This is another motivation for using the version with cosines and sines.

Moreover, when f is real,

$$\begin{aligned} g_\alpha &= \frac{1}{p}\mathrm{Re}\left[\int_0^p e^{-2\pi i\alpha x/p} f(x)dx\right] \\ &= \frac{1}{p}\int_0^p \cos(2\pi i\alpha x/p) f(x)dx \\ &= \frac{1}{2}a_\alpha \ . \end{aligned}$$

Similarly, $h_\alpha = \frac{-1}{2}b_\alpha$. This shows that the real Fourier series relations (7.23) and (7.25) directly follow from the complex Fourier series relations (7.30) and (7.31) without using Exercise 3.

7.2.2 The DFT

The DFT is a discrete anologue of Fourier analysis. The vector space V_p is replaced by sequences with period n: $f_{j+n} = f_j$. A periodic sequence is determined by the n entries[5]: $f = (f_0, \ldots, f_{n-1})^*$, and the vector space of such sequences is C^n. *Sampling* a periodic function of x is one way to create an element of C^n. Choose $\Delta x = p/n$, take *sample points*, $x_j = j\Delta x$, then the samples are $f_j = f(x_j)$. If the continuous f has period p, then the discrete sampled f has period n, because $f_{j+n} = f(x_{j+n})$, and $x_{j+n} = (j+n)\Delta x$, and $n\Delta x = p$.

The DFT modes come from sampling the continuous Fourier modes (7.20) in this way. That is

$$w_{\alpha,j} = w_\alpha(x_j) = \exp(2\pi i\alpha x_j/p) \ .$$

Since $x_j = jp/n$, this gives

$$w_{\alpha,j} = \exp(2\pi i\alpha j/n) = w^{\alpha j} \ , \tag{7.32}$$

where w is a *primitive root of unity*[6]

$$w = e^{2\pi i/n} \ . \tag{7.33}$$

Aliasing is a difference between continuous and discrete Fourier analysis. If $\beta = \alpha + n$ then the samplings of w_β and w_α are the same, $w_\beta(x_j) = w_\alpha(x_j)$ for all j, even though w_β and w_α are different functions of x. Figure 7.1 illustrates this with sine functions instead of complex exponentials. Aliasing implies that the discrete sampled vectors $w_\alpha \in C^n$ with components given by (7.32) are not all different. In fact, the n vectors w_α for $0 \le \alpha < n$ are distinct. After that they repeat: $w_{\alpha+n} = w_\alpha$ (this being an equality between two vectors in C^n).

These n discrete sampled modes form a basis, the DFT basis, for C^n. If f and g are two elements of C^n, the discrete inner product is

$$\langle f, g\rangle = \sum_{j=0}^{n-1} \overline{f}_j g_j \ .$$

[5] The * in $(f_0, \ldots, f_{n-1})^*$ indicates that we think of f as a column vector in C^n.

[6] *Unity* means the number 1. An n^{th} root of x is a y with $y^n = x$. An n^{th} root of unity is *primitive* if $w^n = 1$ but $w^k \neq 1$ for $0 \le k \le n$.

Figure 7.1: *An illustration of aliasing with* $n = 7$ *sampling points and period* $p = 2\pi$, *with modes* $\alpha = 2$ *and* $\beta = \alpha + n = 9$. *The continuous curves are* $\sin(2x)$ *and* $\sin(9x)$. *The circles are these curves sampled at the sample points* $x_j = 2\pi j/n$. *The sampled values are identical even though the functions* $\sin(2x)$ *and* $\sin(9x)$ *are not.*

This is the usual inner product on C^n and leads to the usual l^2 norm $\langle f, f\rangle = \|f\|_{l^2}^2$. We show that the discrete modes form an orthogonal family, but only as far as is allowed by aliasing. That is, if $0 \leq \alpha < n$ and $0 \leq \beta < n$, and $\alpha \neq \beta$, then $\langle w_\alpha, w_\beta\rangle = 0$.

Recall that for any complex number, z, $S(z) = \sum_{j=0}^{n-1} z^j$ has $S = n$ if $z = 1$ and $S = (z^n - 1)/(z-1)$ if $z \neq 1$. Also,

$$\overline{w}_{\alpha,j} = \overline{e^{2\pi i\alpha j/n}} = e^{-2\pi i\alpha j/n} = w^{-\alpha j} \ ,$$

so we can calculate

$$\begin{aligned}\langle w_\alpha, w_\beta\rangle &= \sum_{j=0}^{n-1} w^{-\alpha j} w^{\beta j} \\ &= \sum_{j=0}^{n-1} w^{(\beta-\alpha)j} \\ &= \sum_{j=0}^{n-1} \left(w^{\beta-\alpha}\right)^j\end{aligned}$$

Under our assumptions ($\alpha \neq \beta$ but $0 \leq \alpha < n$, $0 \leq \beta < n$) we have $0 < |\beta - \alpha| < n$, and $z = w^{\beta-\alpha} \neq 1$ (using the fact that w is a *primitive* root of unity). This gives

$$\langle w_\alpha, w_\beta\rangle = \frac{w^{n(\beta-\alpha)} - 1}{w^{\beta-\alpha} - 1} \ .$$

Also,

$$w^{n(\beta-\alpha)} = \left(w^n\right)^{\beta-\alpha} = 1^{\beta-\alpha} = 1 \ ,$$

because w is an n^{th} root of unity. This shows $\langle w_\alpha, w_\beta\rangle = 0$. We also can calculate that $\|w_\alpha\|^2 = \sum_{j=1}^{n} |w_{\alpha,j}|^2 = n$.

Since the n vectors w_α for $0 \leq \alpha < n$ are linearly independent, they form a basis of C^n. That is, any $f \in C^n$ may be written as a linear combination of the w_α:

$$f = \sum_{\alpha=0}^{n-1} \widehat{f}_\alpha w_\alpha \ .$$

By the arguments we gave for Fourier series above, the DFT coefficients are

$$\widehat{f}_\alpha = \frac{1}{n}\langle w_\alpha, f\rangle \ .$$

Expressed explicitly in terms of sums, these relations are

$$\widehat{f}_\alpha = \frac{1}{n}\sum_{j=0}^{n-1} w^{-\alpha j} f_j \ , \tag{7.34}$$

and

$$f_j = \sum_{\alpha=0}^{n-1} \widehat{f}_\alpha w^{\alpha j} \ . \tag{7.35}$$

These are the (*forward*) DFT and *inverse* DFT respectively. Either formula implies the other. If we start with the f_j and calculate the $\widehat{f}_\alpha$ using (7.34), then we can use (7.35) to recover the f_j from the $\widehat{f}_\alpha$, and the other way around. These formulas are more similar to each other than are the corresponding formulas (7.25) and (7.21). Both are sums rather than one being a sum and the other an integral.

There are many other slightly different definitions of the DFT relations. Some people define the discrete Fourier coefficients using a variant of (7.34), such as

$$\widehat{f}_\alpha = \sum_{j=0}^{n-1} w^{\alpha j} f_j \ .$$

This particular version changes (7.35) to

$$f_j = \frac{1}{n}\sum_{\alpha=0}^{n-1} w^{-\alpha j} \widehat{f}_\alpha \ .$$

Still another way to express these relations is to use the *DFT matrix*, W, which is an orthogonal matrix whose (α, j) entry is

$$w_{\alpha,j} = \frac{1}{\sqrt{n}} w^{-\alpha j} \ .$$

The adjoint of W has entries

$$w^*_{j,\alpha} = \overline{w}_{\alpha,j} = \frac{1}{\sqrt{n}} w^{\alpha j} \ .$$

The DFT relations are equivalent to $W^*W = I$. In vector notation, this $\widehat{f} = Wf$ transform differs from (7.34) by a factor of $\sqrt{n}$:

$$\widehat{f}_\alpha = \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} w^{-\alpha j} f_j \ .$$

It has the advantage of making the direct and inverse DFT as similar as possible, with a factor $1/\sqrt{n}$ in both.

As for continuous Fourier series, there is a real discrete cosine and sine transform for real f. The complex coefficients $\widehat{f}_\alpha$, determine the real coefficients

a_α and b_α as before. Aliasing is more complicated for the discrete sine and cosine transform, and depends on whether n is even or odd. The cosine and sine sums corresponding to (7.30) run from $\alpha = 0$ or $\alpha = 1$ roughly to $\alpha = n/2$.

We can estimate the Fourier coefficients of a continuous function by taking the DFT of its sampled values. We call the vector of samples $f^{(n)}$. It has components $f_j^{(n)} = f(x_j)$, where $x_j = j\Delta x$ and $\Delta x = p/n$. The DFT coefficients $\widehat{f}_\alpha^{(n)}$ defined by (7.34) are rectangle rule approximations to the Fourier series coefficients (7.25). This follows from (7.32) and $\frac{1}{p}\Delta x = \frac{1}{n}$, since $\alpha j/n = \alpha x_j/p$ and

$$
\begin{aligned}
\widehat{f}_\alpha^{(n)} &= \frac{1}{n}\sum_{j=0}^{n-1} w^{-\alpha j} f_j^{(n)} \qquad (7.36)\\
&= \frac{1}{p}\Delta x \sum_{j=0}^{n-1} e^{-2\pi i \alpha x_j/n} f(x_j) \; .
\end{aligned}
$$

There is a simple *aliasing formula* for the Fourier coefficients of $f^{(n)}$ in terms of those of f. It depends on aliasing when we put the continuous Fourier representation (7.25) into the discrete Fourier coefficient formula (7.37). If we rewrite (7.25) with β instead of α as the summation index, substitute into (7.37), and change the order of summation, we find

$$
\widehat{f}_\alpha^{(n)} = \sum_{\beta=-\infty}^{\infty} \widehat{f}_\beta \frac{\Delta x}{p} \sum_{j=0}^{n-1} \exp\left[2\pi i \left(\beta - \alpha\right) x_j/n\right] \; .
$$

We have shown that the inner sum is equal to zero unless mode β aliases to mode α on the grid, which means $\beta = \alpha + kn$ for some integer k. We also showed that if mode β does alias to mode α on the grid, then each of the summands is equal to one. Since $\Delta x/p = 1/n$, this implies that

$$
\widehat{f}_\alpha^{(n)} = \sum_{k=-\infty}^{\infty} \widehat{f}_{\alpha+kn} \; . \qquad (7.37)
$$

This shows that the discrete Fourier coefficient is equal to the continuous Fourier coefficient (the $k = 0$ term on the right) plus all the coefficients that alias to it (the terms $k \neq 0$).

You might worry that the continuous function has Fourier coefficients with both positive and negative α while the DFT computes coefficients for $\alpha = 0, 1, \ldots, n-1$. The answer to this is aliasing. We find approximations to the negative α Fourier coefficients using $\widehat{f}_{-\alpha}^{(n)} = \widehat{f}_{n-\alpha}^{(n)}$. It may be more helpful to think of the DFT coefficients as being defined for $\alpha \approx -\frac{n}{2}$ to $\alpha \approx \frac{n}{2}$ (the exact range depending on whether n is even or odd) rather than from $\alpha = 0$ to $\alpha = n-1$. The aliasing formula shows that if f is smooth, then $\widehat{f}_\alpha^{(n)}$ is a very good approximation to $\widehat{f}_\alpha$.

7.2.3 FFT algorithm

It takes n^2 multiplies to carry out all the sums in (7.34) directly (n terms in each of n sums). The *Fast Fourier Transform*, or *FFT*, is an algorithm that calculates the n components of $\widehat{f}$ from the n components of f using $O(n\log(n))$ operations, which is much less for large n.

The idea behind FFT algorithms is clearest when $n = 2m$. A single DFT of size $n = 2m$ is reduced to two DFT calculations of size $m = n/2$ followed by $O(n)$ work. If $m = 2r$, this process can be continued to reduct the two size m DFT operations to four size r DFT operations followed by $2 \cdot O(m) = O(n)$ operations. If $n = 2^p$, this process can be continued p times, where we arrive at $2^p = n$ trivial DFT calculations of size one each. The total work for the p levels is $p \cdot O(n) = O(n\log_2(n))$.

There is a variety of related methods to handle cases where n is not a power of 2, and the $O(n\log(n))$ work count holds for all n. The algorithm is simpler and faster for $n = 2^p$. For example, an FFT with $n = 2^{20} = 1,048,576$ should be significantly faster than with $n = 1,048,573$, which is a prime number.

Let $W_{n\times n}$ be the complex $n \times n$ matrix[7] whose (α, j) entry is $w_{\alpha,j} = w^{-\alpha j}$, where w is a primitive n^{th} root of unity. The DFT (7.34), but for the factor of $\frac{1}{n}$, is the matrix product $\widetilde{f} = W_{n\times n}f$. If $n = 2m$, then w^2 is a primitive m^{th} root of unity and an $m \times m$ DFT involves the matrix product $\widetilde{g} = W_{m\times m}g$, where the (α, k) entry of $W_{m\times m}$ is $(w^2)^{-\alpha k} = w^{-2\alpha k}$. The reduction splits $f \in C^n$ into $g \in C^m$ and $h \in C^m$, then computes $\widetilde{g} = W_{m\times m}g$ and $\widetilde{h} = W_{m\times m}h$, then combines $\widetilde{g}$ and $\widetilde{h}$ to form $\widetilde{f}$.

The elements of $\widetilde{f}$ are given by the sums

$$\widetilde{f}_\alpha = \sum_{j=0}^{n-1} w^{-\alpha j} f_j \; .$$

We split these into even and odd parts, with $j = 2k$ and $j = 2k+1$ respectively. Both of these have k ranging from 0 to $\frac{n}{2} - 1 = m - 1$. For these sums, $-\alpha j = -\alpha(2k) = -2\alpha k$ (even), and $-\alpha j = -\alpha(2k+1) = -2\alpha k - \alpha$ (odd) respectively. Thus

$$\widetilde{f}_\alpha = \sum_{k=0}^{m-1} w^{-2\alpha k} f_{2k} + w^{-\alpha} \sum_{k=0}^{m-1} w^{-2\alpha k} f_{2k+1} \; . \tag{7.38}$$

Now define $g \in C^m$ and $h \in C^m$ to have the even and odd components of f respectively:

$$g_k = f_{2k} \; , \quad h_k = f_{2k+1} \; .$$

The $m \times m$ operations $\widetilde{g} = W_{m\times m}g$ and $\widetilde{h} = W_{m\times m}h$, written out, are

$$\widetilde{g}_\alpha = \sum_{k=0}^{n-1} \left(w^2\right)^{-\alpha k} g_k \; ,$$

[7] This definition of W differs from that of Section 7.2 by a factor of $\sqrt{n}$.

and

$$\widetilde{h}_\alpha = \sum_{k=0}^{n-1} \left(w^2\right)^{-\alpha k} h_k \; .$$

Then (7.38) may be written

$$\widetilde{f}_\alpha = \widetilde{g}_\alpha + w^{-\alpha} \widetilde{h}_\alpha \; . \tag{7.39}$$

This is the last step, which reassembles $\widetilde{f}$ from $\widetilde{g}$ and $\widetilde{h}$. We must apply (7.39) for n values of α ranging from $\alpha = 0$ to $\alpha = n - 1$. The computed $\widetilde{g}$ and $\widetilde{h}$ have period m ($\widetilde{g}_{\alpha+m} = \widetilde{g}_\alpha$, etc.), but the factor $w^{-\alpha}$ in front of $\widetilde{h}$ makes $\widetilde{f}$ have period $n = 2m$ instead.

To summarize, an order n FFT requires first n copying to form g and h, then two order $n/2$ FFT operations, then order n copying, adding, and multiplying. Of course, the order $n/2$ FFT operations themselves boil down to copying and simple arithmetic. As explained in Section 5.7, the copying and memory accessing can take more computer time than the arithmetic. High performance FFT software needs to be chip specific to take full advantage of cache.

7.2.4 Trigonometric interpolation

Interpolating $f(x)$ at points x_j, $j = 1, \ldots, n$, means finding another function $F(x)$ so that $F(x_j) = f(x_j)$ for $j = 1, \ldots, n$. In polynomial interpolation, $F(x)$ is a polynomial of degree $n - 1$. In *trigonometric interpolation*, $F(x)$ is a *trigonometric polynomial* with n terms. Because we are interested in values of $F(x)$ off the grid, do should not take advantage of aliasing to simplify notation. Instead, we let Z_n be a set of integers as symmetric about $\alpha = 0$ as possible. This depends on whether n is even or odd

$$Z_n = \begin{cases} \{-m, -m+1, \ldots, m\} & \text{if } n = 2m+1 \text{ (i.e. } n \text{ is odd)} \\ \{-m+1, \ldots, m\} & \text{if } n = 2m \text{ (i.e. } n \text{ is even)} \end{cases} \tag{7.40}$$

With this notation, an n term trigonometric polynomial may be written

$$F(x) = \sum_{\alpha \in Z_n} c_\alpha e^{2\pi i \alpha x / p} \; .$$

The DFT provides coefficients c_α of the trigonometric interpolating polynomial at n uniformly spaced points.

The high accuracy of DFT based methods comes from the fact that trigonometric polynomials are very efficient approximations for smooth periodic functions. This, in turn, follows from the rapid decay of Fourier coefficients of smooth periodic functions. Suppose, for example, that $f(x)$ is a periodic function that has N bounded derivatives. Let be the n term trigonometric polynomial consisting of n terms of the Fourier sum:

$$F^{(n)}(x) = \sum_{\alpha \in Z_n} \widehat{f}_\alpha e^{2\pi i \alpha x / p} \; .$$

Note that for large m,

$$\sum_{\alpha \geq m} \frac{1}{\alpha^N} \approx \int_{\alpha=m}^{\infty} \frac{1}{\alpha^N} d\alpha = \frac{1}{N-1} \frac{1}{\alpha^{N-1}} \ .$$

The rapid decay inequality (7.28) gives a simple error bound

$$\begin{aligned} \left| f(x) - F^{(n)}(x) \right| &= \left| \sum_{\alpha \notin Z_n} \widehat{f}_\alpha e^{2\pi i \alpha x/p} \right| \\ &\leq C \sum_{\alpha \geq n/2} \frac{1}{\alpha^N} + C \sum_{\alpha \leq -n/2} \frac{1}{|\alpha|^N} \\ &\leq C \cdot \frac{1}{n^{N-1}} \end{aligned}$$

Thus, the smoother f is, the more accurate is the partial Fourier sum approximation.

7.3 Software

Performance tools.

7.4 References and Resources

The classical books on numerical analysis (Dahlquist and Björck, Isaacson and Keller, etc.) discuss the various facts and forms of polynomial interpolation. There are many good books on Fourier analysis. One of my favorites is by Harry Dym and my colleague Henry McKean[8] I learned the aliasing formula (7.37) from a paper by Heinz Kreiss and Joseph Oliger.

7.5 Exercises

1. Verify that both sides of (7.9) are equal to $2a$ when $f(x) = ax^2 + bx + c$.

2. One way to estimate the derivative or integral from function values is to differentiate or integrate the interpolating polynomial.

3. Show that the real Fourier modes (7.29) form an orthogonal family. This means that

   (a) $\langle u_\alpha, u_\beta \rangle = 0$ if $\alpha \neq \beta$.

   (b) $\langle v_\alpha, v_\beta \rangle = 0$ if $\alpha \neq \beta$.

   (c) $\langle u_\alpha, v_\beta \rangle = 0$ for any α and β.

[8]He pronounces McKean as one would Senator McCain.

In (a), α or β may be zero. In (b), α and β start with one. In (c), α starts with zero and β starts with one. It is easy to verify these relations using complex exponentials and the formula $e^{i\theta} = \cos(\theta) + i\sin(\theta)$. For example, we can write $\cos(\theta) = \frac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right)$, so that $u_\alpha = \frac{1}{2}\left(w_\alpha + w_{-\alpha}\right)$. Therefore

$$\langle u_\alpha, u_\beta \rangle = \frac{1}{4}\Big(\langle w_\alpha, w_\beta \rangle + \langle w_{-\alpha}, w_\beta \rangle + \langle w_\alpha, w_{-\beta} \rangle + \langle w_{-\alpha}, w_{-\beta} \rangle \Big) \; .$$

You can check that if $\alpha \geq 0$ and $\beta \geq 0$ and $\alpha \neq \beta$, then all four terms on the right side are zero because the w_α are an orthogonal family. Also check this way that

$$\left\| u_\alpha \right\|_{L^2}^2 = \langle u_\alpha, u_\alpha \rangle = \left\| v_\alpha \right\|_{L^2}^2 = \frac{p}{2} \; , \tag{7.41}$$

if $\alpha \geq 1$, and $\left\| u_0 \right\|_{L^2}^2 = p$.

4. We wish to use Fourier series to solve the boundary value problem from 4.11.

   (a) Show that the solution to (4.44) and (4.45) can be written as a Fourier *sine series*

   $$u(x) = \sum_{\alpha=1}^{n-1} c_\alpha \sin(\pi \alpha x) \; . \tag{7.42}$$

   One way to do this is to define new functions also called u and f, that *extend* the given ones in a special way to satisfy the boundary conditions. For $1 \leq x \leq 2$, we define $f(x) = -f(x-1)$. This defines f in the interval $[0,2]$ in a way that makes it antisymmetric about $x = 1$. Next if $x \notin [0,2]$ there is a k so that $x - 2k \in [0,2]$. Define $f(x) = f(x - 2k)$ in that case. This makes f a periodic function with period $p = 2$ that is antisymmetric about any of the integer points $x = 0$, $x = 1$, etc. Draw a picture to make this two step extension clear. If we express this extended f as a real cosine and sine series, the coefficients of all the cosine terms are zero (why?), so $f(x) = \sum_{\alpha>0} b_\alpha \sin(\pi\alpha x)$. (Why π instead of 2π?) Now determine the c_α in terms of the b_α so that u satisfies (4.44). Use the sine series for u to show that $u = 0$ for $x = 0$ and $x = 1$.

   (b) Write a program in Matlab that uses the Matlab `fft` function to calculate the discrete sine coefficients b_α for a sampled function $f^{(n)}$. The simplest way to program this is to extend f to the interval $[0,2]$ as described, sample this extended f at $2n+1$ uniformly spaced points in the interval $[0,2]$, compute the complex DFT of these samples, then extract the sine coefficients as described in Section 7.2.1.

   (c) Write a Matlab program that takes the b_α and computes $\widetilde{f}(x) = \sum_{\alpha=1}^{n-1} b_\alpha \sin(\pi\alpha x)$ for an arbitrary x value. On one graph, make a plot of $f(x)$ and $\widetilde{f}(x)$ for $x \in [0,1]$. On another plot the error

$f(x) - \widetilde{f}(x)$. Check that the error is zero at the sampling points, as it should be. For plotting, you will have to evaluate $f(x)$ and $\widetilde{f}(x)$ at many more than the sampling points. Do this for $n = 5, 20, 100$ and $f(x) = x^2$ and $f(x) = \sin(3\sin(x))$. Comment on the accuracy in the two cases.

(d) Use the relation between b_α and c_α to calculate the solution to (4.44) and (4.45) for the two functions f in part (c). Comment on the difference in accuracy.

(e) Show that the eigenvectors of the matrix A in Exercise 4.11 are discrete sine modes. Use this to describe an algorithm to express any vector, F, in terms as a linear combination of eigenvectors of A. This is more or less what part (a) does, only stated in a different way.

(f) Use part (e) to develop a *fast algorithm* (i.e. as fast as the FFT rather than the direct method) to solve $Au = F$. Write a Matlab code to do this. Compare the accuracy of the second order method from Exercise 4.11 to the DFT based algorithm of part (c).

Chapter 8

Dynamics and Differential Equations

Many dynamical systems are modeled by first order systems of differential equations. An n component vector $x(t) = (x_1(t), \ldots, x_n(t))$, models the state of the system at time t. The dynamics are modelled by

$$\frac{dx}{dt} = \dot{x}(t) = f(x(t), t) \ , \tag{8.1}$$

where $f(x,t)$ is an n component function, $f(x,t) = f_1(x,t), \ldots, f_n(x,t))$. The system is *autonomous* if $f = f(x)$, i.e., if f is independent of t. A *trajectory* is a function, $x(t)$, that satisfies (8.1). The *initial value problem* is to find a trajectory that satisfies the *initial condition*[1]

$$x(0) = x_0 \ , \tag{8.2}$$

with x_0 being the *initial data*. In practice we often want to specify initial data at some time other than $t_0 = 0$. We set $t_0 = 0$ for convenience of discussion. If $f(x,t)$ is a differentiable function of x and t, then the initial value problem has a solution trajectory defined at least for t close enough to t_0. The solution is unique as long as it exists.[2].

Some problems can be reformulated into the form (8.1), (8.2). For example, suppose $F(r)$ is the force on an object of mass m if the position of the object is $r \in R^3$. Newton's equation of motion: $F = ma$ is

$$m\frac{d^2r}{dt^2} = F(r) \ . \tag{8.3}$$

This is a system of three second order differential equations. The velocity at time t is $v(t) = \dot{r}(t)$. The trajectory, $r(t)$, is determined by the initial position, $r_0 = r(0)$, and the initial velocity, $v_0 = v(0)$.

We can reformulate this as a system of six first order differential equations for the position and velocity, $x(t) = (r(t), v(t))$. In components, this is

$$\begin{pmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ x_5(t) \\ x_6(t) \end{pmatrix} = \begin{pmatrix} r_1(t) \\ r_2(t) \\ r_3(t) \\ v_1(t) \\ v_2(t) \\ v_3(t) \end{pmatrix} \ .$$

[1] There is a conflict of notation that we hope causes little confusion. Sometimes, as here, x_k refers to component k of the vector x. More often x_k refers to an approximation of the vector x at time t_k.

[2] This is the existence and uniqueness theorem for ordinary differential equations. See any good book on ordinary differential equations for details.

The dynamics are given by $\dot{r} = v$ and $\dot{v} = \frac{1}{m}F(r)$. This puts the equations (8.3) into the form (8.1) where

$$f = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} = \begin{pmatrix} x_4 \\ x_5 \\ x_6 \\ \frac{1}{m}F_1(x_1, x_2, x_3) \\ \frac{1}{m}F_2(x_1, x_2, x_3) \\ \frac{1}{m}F_3(x_1, x_2, x_3) \end{pmatrix} .$$

There are variants of this scheme, such as taking $x_1 = r_1$, $x_2 = \dot{r}_1$, $x_3 = r_2$, etc., or using the momentum, $p = m\dot{r}$ rather than the velocity, $v = \dot{r}$. The initial values for the six compenents $x_0 = x(t_0)$ are given by the initial position and velocity components.

8.1 Time stepping and the forward Euler method

For simplicity this section supposes f does not depend on t, so that (8.1) is just $\dot{x} = f(x)$. *Time stepping*, or *marching*, means computing approximate values of $x(t)$ by advancing time in a large number of small steps. For example, if we know $x(t)$, then we can estimate $x(t + \Delta t)$ using

$$x(t + \Delta t) \approx x(t) + \Delta t \dot{x}(t) = x(t) + \Delta t f(x(t)) . \tag{8.4}$$

If we have an approximate value of $x(t)$, then we can use (8.4) to get an approximate value of $x(t + \Delta t)$.

This can be organized into a method for approximating the whole trajectory $x(t)$ for $0 \leq t \leq T$. Choose a *time step*, Δt, and define discrete times $t_k = k\Delta t$ for $k = 0, 1, 2, \ldots$. We compute a sequence of approximate values $x_k \approx x(t_k)$. The approximation (8.4) gives

$$x_{k+1} \approx x(t_{k+1}) = x(t_k + \Delta t) \approx x(t_k) + \Delta t f(x_k) \approx x_k + \Delta t f(x_k) .$$

The *forward Euler* method takes this approximation as the definition of x_{k+1}:

$$x_{k+1} = x_k + \Delta t f(x_k) \ . \tag{8.5}$$

Starting with initial condition x_0 (8.5) allows us to calculate x_1, then x_2, and so on as far as we like.

Solving differential equations sometimes is called integration. This is because of the fact that if $f(x,t)$ is independent of x, then $\dot{x}(t) = f(t)$ and the solution of the initial value problem (8.1) (8.2) is given by

$$x(t) = x(0) + \int_0^t f(s)ds .$$

If we solve this using the rectangle rule with time step Δt, we get

$$x(t_k) \approx x_k = x(0) + \Delta t \sum_{j=0}^{k-1} f(t_j) .$$

We see from this that $x_{k+1} = x_k + \Delta t f(t_k)$, which is the forward Euler method in this case. We know that the rectangle rule for integration is first order accurate. This is a hint that the forward Euler method is first order accurate more generally.

We can estimate the accuracy of the forward Euler method using an informal *error propagation equation*. The error, as well as the solution, evolves (or propagates) from one time step to the next. We write the value of the exact solution at time t_k as $\widetilde{x}_k = x(t_k)$. The error at time t_k is $e_k = x_k - \widetilde{x}_k$. The *residual* is the amount by which $\widetilde{x}_k$ fails to satisfy the forward Euler equations[3] (8.5):

$$\widetilde{x}_{k+1} = \widetilde{x}_k + \Delta t f(\widetilde{x}_k) + \Delta t r_k \ , \tag{8.6}$$

which can be rewritten as

$$r_k = \frac{x(t_k + \Delta t) - x(t_k)}{\Delta t} - f(x(t_k)) \ . \tag{8.7}$$

In Section 3.2 we showed that

$$\frac{x(t_k + \Delta t) - x(t_k)}{\Delta t} = \dot{x}(t_k) + \frac{\Delta t}{2}\ddot{x}(t_k) + O(\Delta t^2) \ .$$

Together with $\dot{x} = f(x)$, this shows that

$$r_k = \frac{\Delta t}{2}\ddot{x}(t_k) + O(\Delta t^2) \ , \tag{8.8}$$

which shows that $r_k = O(\Delta t)$.

The error propagation equation, (8.10) below, estimates e in terms of the residual. We can estimate $e_k = x_k - \widetilde{x}_k = x_k - x(t_k)$ by comparing (8.5) to (8.6)

$$e_{k+1} = e_k + \Delta t \left(f(x(k)) - f(\widetilde{x}_k) \right) - \Delta t r_k \ .$$

This resembles the forward Euler method applied to approximating some function $e(t)$. Being optimistic, we suppose that x_k and $x(t_k)$ are close enough to use the approximation (f' is the Jacobian matrix of f as in Section **??**)

$$f(x_k) = f(\widetilde{x}_k + e_k) \approx f(\widetilde{x}_k) + f'(\widetilde{x}_k)e_k \ ,$$

and then

$$e_{k+1} \approx e_k + \Delta t \left(f'(\widetilde{x}_k)e_k - r_k \right) \ . \tag{8.9}$$

If this were an equality, it would imply that the e_k satisfy the forward Euler approximation to the differential equation

$$\dot{e} = f'\left(x\left(t\right)\right)e - r(t) \ , \tag{8.10}$$

where $x(t)$ satisfies (8.1), e has initial condition $e(0) = 0$, and $r(t)$ is given by (8.8):

$$r(t) = \frac{\Delta t}{2}\ddot{x}(t) \ . \tag{8.11}$$

[3] We take out one factor of Δt so that the order of magnitude of r is the order of magnitude of the error e.

The *error propagation equation* (8.10) is linear, so $e(t)$ should be proportional to[4] r, which is proportional to Δt. If the approximate $e(t)$ satisfies $\|e(t)\| = C(t)\Delta t$, then the exact $e(t)$ should satisfy $\|e(t)\| = O(\Delta t)$, which means there is a $C(t)$ with

$$\|e(t)\| \leq C(t)\Delta t \, . \tag{8.12}$$

This is the first order accuracy of the forward Euler method.

It is important to note that this argument does not prove that the forward Euler method converges to the correct answer as $\Delta t \to 0$. Instead, it assumes the convergence and uses it to get a quantitative estimate of the error. The formal proof of convergence may be found in any good book on numerical methods for ODEs, such as the book by Isaiah Iserles.

If this analysis is done a little more carefully, it shows that there is an asymptotic error expansion

$$x_k \sim x(t_k) + \Delta t u_1(t_k) + \Delta t^2 u_2(t_k) + \cdots \, . \tag{8.13}$$

The leading error coefficient, $u_1(t)$, is the solution of (8.10). The higher order coefficients, $u_2(t)$, etc. are found by solving higher order error propagation equations.

The *modified equation* is a different approach to error analysis that allows us to determine the long time behavior of the error. The idea is to modify the differential equaion (8.1) so that the solution is closer to the forward Euler sequence. We know that $x_k = x(t_k) + O(\Delta t)$. We seek a differential equation $\dot{y} = g(y)$ so that $x_k = y(t_k) + O(\Delta t^2)$ We construct an error expansion for the equation itself rather than the solution.

It is simpler to require $y(t)$ so satisfy the forward Euler equation at each t, not just the discrete times t_k:

$$y(t + \Delta t) = y(t) + \Delta t f(y(t)) \, . \tag{8.14}$$

We seek

$$g(y, \Delta t) = g_0(y) + \Delta t g_1(y) + \cdots \tag{8.15}$$

so that the solution of (8.14) satisfies $\dot{y} = g(y) + O(\Delta t^2)$. We combine the expansion (8.15) with the Taylor series

$$y(t + \Delta t) = y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + O(\Delta t^3) \, ,$$

to get (dividing both sides by Δt, $O(\Delta t^3)/\Delta t = O(\Delta t^2)$.):

$$\begin{aligned} y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + O(\Delta t^3) &= y(t) + \Delta t f(y(t)) \\ g_0(y(t)) + \Delta t g_1(y(t)) + \frac{\Delta}{2}\ddot{y}(t) &= f(y(t)) + O(\Delta t^2) \end{aligned}$$

[4] The value of $e(t)$ depends on the values of $r(s)$ for $0 \leq s \leq t$. We can solve $\dot{u} = f'(x)u - w$, where $w = \frac{1}{2}\ddot{x}$, then solve (8.10) by setting $e = \Delta t u$. This shows that $\|e(t)\| = \Delta t \, \|u(t)\|$, which is what we want, with $C(t) = \|u(t)\|$.

Equating the leading order terms gives the unsurprising result

$$g_0(y) = f(y) \; ,$$

and leaves us with

$$g_1(y(t)) + \frac{1}{2}\ddot{y}(t) = O(\Delta t) \; . \tag{8.16}$$

We differentiate $\dot{y} = f(y) + O(\Delta t)$ and use the chain rule, giving

$$\begin{aligned} \ddot{y} = \frac{d}{dt}\dot{y} &= \frac{d}{dt}\Big(f(y(t)) + O(\Delta t)\Big) \\ &= f'(y)\dot{y}(t) + O(\Delta t) \\ \ddot{y} &= f'(y)f(y) + O(\Delta t) \end{aligned}$$

Substituting this into (8.16) gives

$$g_1(y) = -\frac{1}{2}f'(y)f(y) \; .$$

so the modified equation, with the first correction term, is

$$\dot{y} = f(y) - \frac{\Delta t}{2}f'(y)f(y) \; . \tag{8.17}$$

A simple example illustrates these points. The nondimensional harmonic oscillator equation is $\ddot{r} = -r$. The solution is $r(t) = a\sin(t) + b\cos(t)$, which oscillates but does not grow or decay. We write this in first order as $\dot{x}_1 = x_2$, $\dot{x}_2 = -x_1$, or

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_2 \\ -x_1 \end{pmatrix} \; . \tag{8.18}$$

Therefore, $f(x) = \begin{pmatrix} x_2 \\ -x_1 \end{pmatrix}$, $f' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$, and $f'f = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} x_2 \\ -x_1 \end{pmatrix} = \begin{pmatrix} -x_1 \\ -x_2 \end{pmatrix}$, so (8.17) becomes

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix} + \frac{\Delta t}{2}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \frac{\Delta t}{2}t & 1 \\ -1 & \frac{\Delta t}{2} \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \; .$$

We can solve this by finding eigenvalues and eigenvectors, but a simpler trick is to use a partial integrating factor and set $y(t) = e^{\frac{1}{2}\Delta t \cdot t}z(t)$, where $\dot{z} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}z$. Since $z_1(t) = a\sin(t) + b\cos(t)$, we have our approximate numerical solution $y_1(t) = e^{\frac{1}{2}\Delta t \cdot t}\left(a\sin(t) + b\cos(t)\right)$. Therefore

$$\|e(t)\| \approx \left(e^{\frac{1}{2}\Delta t \cdot t} - 1\right) \; . \tag{8.19}$$

This modified equation analysis confirms that forward Euler is first order accurate. For small Δt, we write $e^{\frac{1}{2}\Delta t \cdot t} - 1 \approx \frac{1}{2}\Delta t \cdot t$ so the error is about $\Delta t \cdot t\,(a\sin(t) + b\cos(t))$. Moreover, it shows that the error grows with t. For each fixed t, the error satisfies $\|e(t)\| = O(\Delta t)$ but the implied constant $C(t)$ (in $\|e(t)\| \leq C(t)\Delta t$) is a growing function of t, at least as large as $C(t) \geq \frac{t}{2}$.

8.2 Runge Kutta methods

Runge Kutta[5] methods are a general way to achieve higher order approximate solutions of the initial value problem (8.1), (8.2). Each time step consists of m stages, each stage involving a single evaluation of f. The relatively simple four stage fourth order method is in wide use today. Like the forward Euler method, but unlike multistep methods, Runge Kutta time stepping computes x_{k+1} from x_k without using values x_j for $j < k$. This simplifies error estimation and adaptive time step selection.

The simplest Runge Kutta method is forward Euler (8.5). Among the second order methods is Heun's[6]

$$\xi_1 = \Delta t f(x_k, t_k) \tag{8.20}$$
$$\xi_2 = \Delta t f(x_k + \xi_1, t_k + \Delta t) \tag{8.21}$$
$$x_{k+1} = x_k + \frac{1}{2}(\xi_1 + \xi_2) \ . \tag{8.22}$$

The calculations of ξ_1 and ξ_2 are the two *stages* of Heun's method. Clearly they depend on k, though that is left out of the notation.

To calculate x_k from x_0 using a Runge Kutta method, we apply take k time steps. Each time step is a transformation that may be written

$$x_{k+1} = \widehat{S}(x_k, t_k, \Delta t) \ .$$

As in Chapter 6, we express the general time step as[7] $x' = \widehat{S}(\overline{x}, t, \Delta t)$. This $\widehat{S}$ approximates the exact *solution operator*, $S(\overline{x}, t, \Delta t)$. We say that $x' = S(\overline{x}, t, \Delta t)$ if there is a trajectory satisfying the differential equation (8.1) so that $x(t) = \overline{x}$ and $x' = x(t + \Delta t)$. In this notation, we would give Heun's method as $x' = \widehat{S}(\overline{x}, \Delta t) = \overline{x} + \frac{1}{2}(\xi_1 + \xi_2)$, where $\xi_1 = f(\overline{x}, t, \Delta t)$, and $\xi_2 = f(\overline{x} + \xi_1, t, \Delta t)$.

The best known and most used Runge Kutta method, which often is called *the* Runge Kutta method, has four stages and is fourth order accurate

$$\xi_1 = \Delta t f(\overline{x}, t) \tag{8.23}$$
$$\xi_2 = \Delta t f(\overline{x} + \frac{1}{2}\xi_1, t + \frac{1}{2}\Delta t) \tag{8.24}$$
$$\xi_3 = \Delta t f(\overline{x} + \frac{1}{2}\xi_2, t + \frac{1}{2}\Delta t) \tag{8.25}$$
$$\xi_4 = \Delta t f(\overline{x} + \xi_3, t + \Delta t) \tag{8.26}$$
$$x' = \overline{x} + \frac{1}{6}(\xi_1 + 2\xi_2 + 2\xi_3 + \xi_4) \ . \tag{8.27}$$

---

[5]Carl Runge was Professor of applied mathematics at the turn of the 20^{th} century in Göttingen, Germany. Among his colleagues were David Hilbert (of Hilbert space) and Richard Courant. But Courant was forced to leave Germany and came to New York to found the Courant Institute. Kutta was a student of Runge.

[6]Heun, whose name rhymes with "coin", was another student of Runge.

[7]The notation x' here does not mean the derivative of x with respect to t (or any other variable) as it does in some books on differential equations.

Understanding the accuracy of Runge Kutta methods comes down to Taylor series. The reasoning of Section 8.1 suggests that the method has error $O(\Delta t^p)$ if

$$\widehat{S}(\overline{x}, t, \Delta t) = S(\overline{x}, t, \Delta t) + \Delta t \; r \;, \tag{8.28}$$

where $\|r\| = O(\Delta t^p)$. The reader should verify that this definition of the residual, r, agrees with the definition in Section 8.1. The analysis consists of expanding both $S(\overline{x}, t, \Delta t)$ and $\widehat{S}(\overline{x}, t, \Delta t)$ in powers of Δt. If the terms agree up to order Δt^p but disagree at order Δt^{p+1}, then p is the order of accuracy of the overall method.

We do this for Heun's method, allowing f to depend on t as well as x. The calculations resemble the derivation of the modified equation (8.17). To make the expansion of S, we have $x(t) = \overline{x}$, so

$$x(t + \Delta t) = \overline{x} + \Delta t \dot{x}(t) + \frac{\Delta t^2}{2} \ddot{x}(t) + O(\Delta t^3) \;.$$

Differentiating with respect to t and using the chain rule gives:

$$\ddot{x} = \frac{d}{dt}\dot{x} = \frac{d}{dt} f(x(t), t) = f'(x(t), t)\dot{x}(t) + \partial_t f(x(t), t) \;,$$

so

$$\ddot{x}(t) = f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \;.$$

This gives

$$S(\overline{x}, t, \Delta t) = \overline{x} + \Delta t f(\overline{x}, t) + \frac{\Delta t^2}{2} \left(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \right) + O(\Delta t^3) \;. \tag{8.29}$$

To make the expansion of $\widehat{S}$ for Heun's method, we first have $\xi_1 = \Delta t f(\overline{x}, t)$, which needs no expansion. Then

$$\begin{aligned}
\xi_2 &= \Delta t f(\overline{x} + \xi_1, t + \Delta t) \\
&= \Delta t \big(f(\overline{x}, t) + f'(\overline{x}, t)\xi_1 + \partial_t f(\overline{x}, t)\Delta t + O(\Delta t^2) \big) \\
&= \Delta t f(\overline{x}, t) + \Delta t^2 \big(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \big) + O(\Delta t^3) \;.
\end{aligned}$$

Finally, (8.22) gives

$$\begin{aligned}
x' &= \overline{x} + \frac{1}{2}(\xi_1 + \xi_2) \\
&= \overline{x} + \frac{1}{2}\Big\{ \Delta t f(\overline{x}, t) + \Big[\Delta t f(\overline{x}, t) + \Delta t^2 \big(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \big) \Big] \Big\} + O(\Delta t^3)
\end{aligned}$$

Comparing this to (8.29) shows that

$$\widehat{S}(\overline{x}, t, \Delta t) = S(\overline{x}, t, \Delta t) + O(\Delta t^3) \;.$$

which is the second order accuracy of Heun's method. The same kind of analysis shows that the four stage Runge Kutta method is fourth order accurate, but it would take a full time week. It was Kutta's thesis.

8.3 Linear systems and stiff equations

A good way to learn about the behavior of a numerical method is to ask what it would do on a properly chosen *model problem*. In particular, we can ask how an initial value problem solver would perform on a linear system of differential equations

$$\dot{x} = Ax \; . \tag{8.30}$$

We can do this using the eigenvalues and eigenvectors of A if the eigenvectors are not too ill conditioned. If[8] $Ar_\alpha = \lambda_\alpha r_\alpha$ and $x(t) = \sum_{\alpha=1}^{n} u_\alpha(t) r_\alpha$, then the components u_α satisfy the scalar differential equations

$$\dot{u}_\alpha = \lambda_\alpha u_\alpha \; . \tag{8.31}$$

Suppose $x_k \approx x(t_k)$ is the approximate solution at time t_k. Write $x_k = \sum_{\alpha=1}^{n} u_{\alpha k} r_\alpha$. For a majority of methods, including Runge Kutta methods and linear multistep methods, the $u_{\alpha k}$ (as functions of k) are what you would get by applying the same time step approximation to the scalar equation (8.31). The eigenvector matrix, R, (see Section **??**), that diagonalizes the differential equation (8.30) also diagonalizes the computational method. The reader should check that this is true of the Runge Kutta methods of Section 8.2.

One question this answers, at least for linear equations (8.30), is how small the time step should be. From (8.31) we see that the λ_α have units of 1/time, so the $1/\left|\lambda_\alpha\right|$ have units of time and may be called *time scales*. Since Δt has units of time, it does not make sense to say that Δt is small in an absolute sense, but only relative to other time scales in the problem. This leads to the following:
Possibility: *A time stepping approximation to (8.30) will be accurate only if*

$$\max_\alpha \Delta t \left|\lambda_\alpha\right| \ll 1 \; . \tag{8.32}$$

Although this *possibility* is not true in every case, it is a dominant technical consideration in most practical computations involving differential equations. The *possibility* suggests that the time step should be considerably smaller than the smallest time scale in the problem, which is to say that Δt should *resolve* even the fastest time scales in the problem.

A problem is called *stiff* if it has two characteristics: (i) there is a wide range of time scales, and (ii) the fastest time scale modes have almost no energy. The second condition states that if $\left|\lambda_\alpha\right|$ is large (relative to other eigenvalues), then $\left|u_\alpha\right|$ is small. Most time stepping problems for partial differential equations are stiff in this sense. For a stiff problem, we would like to take larger time steps than (8.32):

$$\Delta t \left|\lambda_\alpha\right| \ll 1 \qquad \begin{cases} \text{for all } \alpha \text{ with } u_\alpha \text{ signifi-} \\ \text{cantly different from zero.} \end{cases} \tag{8.33}$$

What can go wrong if we ignore (8.32) and choose a time step using (8.33) is *numerical instability*. If mode u_α is one of the large $\left|\lambda_\alpha\right|$ small $\left|u_\alpha\right|$ modes,

[8] We call the eigenvalue index α to avoid conflict with k, which we use to denote the time step.

it is natural to assume that the real part satisfies $\mathrm{Re}(\lambda_\alpha) \leq 0$. In this case we say the mode is *stable* because $|u_\alpha(t)| = |u_\alpha(0)| \, e^{\lambda_\alpha t}$ does not increase as t increases. However, if $\Delta t \lambda_\alpha$ is not small, it can happen that the time step approximation to (8.31) is unstable. This can cause the $u_{\alpha k}$ to grow exponentially while the actual u_α decays or does not grow. Exercise 8 illustrates this. Time step restrictions arising from stability are called *CFL* conditions because the first systematic discussion of this possibility in the numerical solution of partial differential equations was given in 1929 by Richard Courant, Kurt Friedrichs, and Hans Levy.

8.4 Adaptive methods

Adaptive means that the computational steps are not fixed in advance but are determined as the computation proceeds. Section 3.6, discussed an integration algorithm that chooses the number of integration points adaptively to achieve a specified overall accuracy. More sophisticated adaptive strategies choose the distribution of points to maximize the accuracy from a given amount of work. For example, suppose we want an $\widehat{I}$ for $I = \int_0^2 f(x)dx$ so that $\left|\widehat{I} - I\right| < .06$. It might be that we can calculate $I_1 = \int_0^1 f(x)dx$ to within .03 using $\Delta x = .1$ (10 points), but that calculating $I_2 = \int_1^2 f(x)dx$ to within .03 takes $\Delta x = .02$ (50 points). It would be better to use $\Delta x = .1$ for I_1 and $\Delta x = .02$ for I_2 (60 points total) rather than using $\Delta x = .02$ for all of I (100 points).

Adaptive methods can use *local error estimates* to concentrate computational resources where they are most needed. If we are solving a differential equation to compute $x(t)$, we can use a smaller time step in regions where x has large accelleration. There is an active community of researchers developing systematic ways to choose the time steps in a way that is close to optimal without having the overhead in choosing the time step become larger than the work in solving the problem. In many cases they conclude, and simple model problems show, that a good strategy is to *equidistribute* the *local truncation error*. That is, to choose time steps Δt_k so that the the local truncation error $\rho_k = \Delta t_k r_k$ is nearly constant.

If we have a variable time step Δt_k, then the times $t_{k+1} = t_k + \Delta t_k$ form an irregular *adapted mesh* (or adapted *grid*). Informally, we want to choose a mesh that *resolves* the solution, $x(t)$ being calculated. This means that knowing the $x_k \approx x(t_k)$ allows you make an accurate reconstruction of the function $x(t)$, say, by interpolation. If the points t_k are too far apart then the solution is *underresolved*. If the t_k are so close that $x(t_k)$ is predicted accurately by a few neighboring values ($x(t_j)$ for $j = k \pm 1, k \pm 2$, etc.) then $x(t)$ is *overresolved*, we have computed it accurately but paid too high a price. An efficient adaptive mesh avoids both underresolution and overresolution.

Figure 8.1 illustrates an adapted mesh with equidistributed interpolation error. The top graph shows a curve we want to resolve and a set of points that concentrates where the curvature is high. Also also shown is the piecewise linear

Figure 8.1: *A nonuniform mesh for a function that needs different resolution in different places. The top graph shows the function and the mesh used to interpolate it. The bottom graph is the difference between the function and the piecewise linear approximation. Note that the interpolation error equidistributed though the mesh is much finer near $x = 0$.*

curve that connects the interpolation points. On the graph is looks as though the piecewise linear graph is closer to the curve near the center than in the smoother regions at the ends, but the error graph in the lower frame shows this is not so. The reason probably is that what is plotted in the bottom frame is the vertical distance between the two curves, while what we see in the picture is the two dimensional distance, which is less if the curves are steep. The bottom frame illustrates equidistribution of errer. The interpolation error is zero at the grid points and gets to be as large as about -6.3×10^{-3} in each interpolation interval. If the points were uniformly spaced, the interpolation error would be smaller near the ends and larger in the center. If the points were bunched even more than they are here, the interpolation error would be smaller in the center than near the ends. We would not expect such perfect equidistribution in real problems, but we might have errors of the same order of magnitude everywhere.

For a Runge Kutta method, the local truncation error is $\rho(x, t, \Delta t) = \widehat{S}(x, t, \Delta t) - S(x, t, \Delta t)$. We want a way to estimate ρ and to choose Δt so that $|\rho| = e$, where e is the value of the equidistributed local truncation error. We suggest a method related to Richardson extrapolation (see Section 3.3), comparing the result of one time step of size Δt to two time steps of size $\Delta t/2$. The best adaptive Runge Kutta differential equation solvers do not use this generic method, but instead use ingenious schemes such as the Runge Kutta Fehlberg five stage scheme that simultaneously gives a fifth order $\widehat{S}_5$, but also gives an estimate of the difference between a fourth order and a fifth order method, $\widehat{S}_5 - \widehat{S}_4$. The method described here does the same thing with eight function evaluations instead of five.

The Taylor series analysis of Runge Kutta methods indicates that $\rho(x, t, \Delta t) = \Delta t^{p+1}\sigma(x, t) + O(\Delta t^{p+2})$. We will treat σ as a constant because the all the x and t values we use are within $O(\Delta t)$ of each other, so variations in σ do not effect the principal error term we are estimating. With one time step, we get $x' = \widehat{S}(\overline{x}, t, \Delta t)$ With two half size time steps we get first $\widetilde{x}_1 = \widehat{S}(\overline{x}, t, \Delta t/2)$, then $\widetilde{x}_2 = \widehat{S}(\widetilde{x}_1, t + \Delta t/2, \Delta t/2)$.

We will show, using the Richardson method of Section 3.3, that

$$x' - \widetilde{x}_2 = \left(1 - 2^{-p}\right) \rho(\overline{x}, t, \Delta t) + O(\Delta t^{p+1}) \ . \tag{8.34}$$

We need to use the *semigroup* property of the solution operator: If we "run" the exact solution from $\overline{x}$ for time $\Delta t/2$, then run it from there for another time $\Delta t/2$, the result is the same as running it from $\overline{x}$ for time Δt. Letting x be the solution of (8.1) with $x(t) = \overline{x}$, the formula for this is

$$S(\overline{x}, t, \Delta t) \quad = \quad S(x(t + \Delta t/2), t + \Delta t/2, \Delta t/2)$$

$$= \quad S(\,S(\overline{x},t,\Delta t/2)\,,t+\Delta t/2,\Delta t/2)\;.$$

We also need to know that $S(x,t,\Delta t) = x + O(\Delta t)$ is reflected in the Jacobian matrix S' (the matrix of first partials of S with respect to the x arguments with t and Δt fixed)[9]: $S'(x,t,\Delta t) = I + O(\Delta t)$.

The actual calculation starts with

$$\begin{aligned}
\widetilde{x}_1 &= \widehat{S}(\overline{x},t,\Delta t/2) \\
&= S(\overline{x},t,\Delta t/2) + 2^{-(p+1)}\Delta t^{-(p+1)}\sigma + O(\Delta t^{-(p+2)})\;,
\end{aligned}$$

and

$$\begin{aligned}
\widetilde{x}_2 &= \widehat{S}(\widetilde{x}_1,t+\Delta t,\Delta t/2) \\
&= S(\widetilde{x}_1,t+\Delta t/2,\Delta t/2) + 2^{-(p+1)}\Delta t^{-(p+1)}\sigma + O(\Delta t^{-(p+2)})\;,
\end{aligned}$$

We simplify the notation by writing $\widetilde{x}_1 = x(t{+}\Delta t/2){+}u$ with $u = 2^{-(p+1)}\Delta t^p\sigma +$ $O(\Delta t^{-(p+2)})$. Then $\|u\| = O(\Delta t^{-(p+1)})$ and also (used below) $\Delta t\,\|u\| = O(\Delta t^{-(p+2)})$ and (since $p \geq 1$) $\|u\|^2 = O(\Delta t^{-(2p+2)}) = O(\Delta t^{-(p+2)})$. Then

$$\begin{aligned}
S(\widetilde{x}_1,t+\Delta t/2,\Delta t/2) &= S(x(t+\Delta t/2)+u,t+\Delta t/2,\Delta t/2) \\
&= S(x(t+\Delta t/2),t+\Delta t/2,\Delta t/2) + S'u + O\left(\|u\|^2\right) \\
&= S(x(t+\Delta t/2),t+\Delta t/2,\Delta t/2) + u + O\left(\|u\|^2\right) \\
&= S(\overline{x},t,\Delta t) + 2^{-(p+1)}\Delta t^p\sigma + uO\left(\Delta t^{p+2}\right)\;.
\end{aligned}$$

Altogether, since $2\cdot 2^{-(p+1)} = 2^{-p}$, this gives

$$\widetilde{x}_2 = S(\overline{x},t,\Delta t) + 2^{-p}\Delta t^{p+1}\sigma + O\left(\Delta t^{p+2}\right)\;.$$

Finally, a single size Δt time step has

$$x' = X(\overline{x},\Delta t,t) + \Delta t^{p+1}\sigma + O\left(\Delta t^{p+2}\right)\;.$$

Combining these gives (8.34). It may seem like a mess but it has a simple underpinning. The whole step produces an error of order Δt^{p+1}. Each half step produces an error smaller by a factor of 2^{p+1}, which is the main idea of Richardson extrapolation. Two half steps produce almost exactly twice the error of one half step.

There is a simple adaptive strategy based on the local truncation error estimate (8.34). We arrive at the start of time step k with an estimated time step size Δt_k. Using that time step, we compute $x' = \widehat{S}(x_k,t_k,\Delta t_k)$ and $\widetilde{x}_2$ by taking two time steps from x_k with $\Delta t_k/2$. We then estimate ρ_k using (8.34):

$$\widehat{\rho}_k = \frac{1}{1-2^{-p}}\left(x' - \widetilde{x}_2\right)\;. \tag{8.35}$$

[9] This fact is a consequence of the fact that S is twice differentiable as a function of all its arguments, which can be found in more theoretical books on differential equations. The Jacobian of $f(x) = x$ is $f'(x) = I$.

This suggests that if we adjust Δt_k by a factor of μ (taking a time step of size $\mu\Delta t_k$ instead of Δt_k), the error would have been $\mu^{p+1}\widehat{\rho}_k$. If we choose μ to exactly equidistribute the error (according to our estimated ρ, we would get

$$e = \mu^{p+1} \|\widehat{\rho}_k\| \implies \mu_k = \left(e/\|\widehat{\rho}_k\|\right)^{1/(p+1)} . \tag{8.36}$$

We could use this estimate to adjust Δt_k and calculate again, but this may lead to an infinite loop. Instead, we use $\Delta t_{k+1} = \mu_k \Delta t_k$.

Chapter 3 already mentioned the paradox of error estimation. Once we have a quantitative error estimate, we should use it to make the solution more accurate. This means taking

$$x_{k+1} = \widehat{S}(x_k, t_k, \Delta t_k) + \widehat{\rho}_k ,$$

which has order of accuracy $p+1$, instead of the order p time step $\widehat{S}$. This increases the accuracy but leaves you without an error estimate. This gives an order $p+1$ calculation with a mesh chosen to be nearly optimal for an order p calculation. Maybe the reader can find a way around this paradox. Some adaptive strategies reduce the overhead of error estimation by using the Richardson based time step adjustment, say, every fifth step.

One practical problem with this strategy is that we do not know the quantitative relationship between local truncation error and global error[10]. Therefore it is hard to know what e to give to achieve a given global error. One way to estimate global error would be to use a given e and get some time steps Δt_k, then redo the computation with each interval $[t_k, t_{k+1}]$ cut in half, taking exactly twice the number of time steps. If the method has order of accuracy p, then the global error should decrease very nearly by a factor of 2^p, which allows us to estimate that error. This is rarely done in practice. Another issue is that there can be isolated zeros of the leading order truncation error. This might happen, for example, if the local truncation error were proportional to a scalar function $\ddot{x}$. In (8.36), this could lead to an unrealistically large time step. One might avoid that, say, by replacing μ_k with $\min(\mu_k, 2)$, which would allow the time step to grow quickly, but not too much in one step. This is less of a problem for systems of equations.

8.5 Multistep methods

Linear multistep methods are the other class of methods in wide use. Rather than giving a general discussion, we focus on the two most popular kinds, methods based on difference approximations, and methods based on integrating $f(x(t))$, *Adams methods*. Hybrids of these are possible but often are unstable. For some reason, almost nothing is known about methods that both are multistage and multistep.

[10] *Adjoint* based error control methods that address this problem are still in the research stage (2006).

Multistep methods are characterized by using information from previous time steps to go from x_k to x_{k+1}. We describe them for a fixed Δt. A simple example would be to use the second order centered difference approximation $\dot{x}(t) \approx \left(x(t+\Delta t) - x(t-\Delta t)\right)/2\Delta t$ to get

$$\left(x_{k+1} - x_{k-1}\right)/2\Delta t = f(x_k) \; ,$$

or

$$x_{k+1} = x_{k-1} + 2\Delta t f(x_k) \; . \tag{8.37}$$

This is the *leapfrog*[11] method. We find that

$$\widetilde{x}_{k+1} = \widetilde{x}_{k-1} + 2\Delta t f(\widetilde{x}_k) + \Delta t O(\Delta t^2) \; ,$$

so it is second order accurate. It achieves second order accuracy with a single evaluation of f per time step. Runge Kutta methods need at least two evaluations per time step to be second order. Leapfrog uses x_{k-1} and x_k to compute x_{k+1}, while Runge Kutta methods forget x_{k-1} when computing x_{k+1} from x_k.

The next method of this class illustrates the subtletlies of multistep methods. It is based on the four point one sided difference approximation

$$\dot{x}(t) = \frac{1}{\Delta t}\left(\frac{1}{3}x(t+\Delta t) + \frac{1}{2}x(t) - x(t-\Delta t) + \frac{1}{6}x(t-2\Delta t)\right) + O\left(\Delta t^3\right) \; .$$

This suggests the time stepping method

$$f(x_k) = \frac{1}{\Delta t}\left(\frac{1}{3}x_{k+1} + \frac{1}{2}x_k - x_{k-1} + \frac{1}{6}x_{k-2}\right) \; , \tag{8.38}$$

which leads to

$$x_{k+1} = 3\Delta t f(x_k) - \frac{3}{2}x_k + 3x_{k-1} - \frac{1}{2}x_{k-2} \; . \tag{8.39}$$

This method never is used in practice because it is *unstable* in a way that Runge Kutta methods cannot be. If we set $f \equiv 0$ (to solve the model problem $\dot{x} = 0$), (8.38) becomes the *recurrence relation*

$$x_{k+1} + \frac{3}{2}x_k - 3x_{k-1} + \frac{1}{2}x_{k-2} = 0 \; , \tag{8.40}$$

which has *characteristic polynomial*[12] $p(z) = z^3 + \frac{3}{2}z^2 - 3z + \frac{1}{2}$. Since one of the roots of this polynomial has $|z| > 1$, general solutions of (8.40) grow exponentially on a Δt time scale, which generally prevents approximate solutions from converging as $\Delta t \to 0$. This cannot happen for Runge Kutta methods because setting $f \equiv 0$ always gives $x_{k+1} = x_k$, which is the exact answer in this case.

[11] Leapfrog is a game in which two or more children move forward in a line by taking turns jumping over each other, as (8.37) jumps from x_{k-1} to x_{k+1} using only $f(x_k)$.

[12] If $p(z) = 0$ then $x_k = z^k$ is a solution of (8.40).

Adams methods use old values of f but not old values of x. We can integrate (8.1) to get

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(t))dt \; . \tag{8.41}$$

An accurate estimate of the integral on the right leads to an accurate time step. *Adams Bashforth* methods estimate the integral using polynomial extrapolation from earlier f values. At its very simplest we could use $f(x(t)) \approx f(x(t_k))$, which gives

$$\int_{t_k}^{t_{k+1}} f(x(t))dt \approx (t_{k+1} - t_k) f(x(t_k)) \; .$$

Using this approximation on the right side of (8.41) gives forward Euler.

The next order comes from linear rather than constant extrapolation:

$$f(x(t)) \approx f(x(t_k)) + (t - t_k) \frac{f(x(t_k)) - f(x(t_{k-1}))}{t_k - t_{k-1}} \; .$$

With this, the integral is estimated as (the generalization to non constant Δt is Exercise **??**):

$$\begin{aligned} \int_{t_k}^{t_{k+1}} f(x(t))dt &\approx \Delta t f(x(t_k)) + \frac{\Delta t^2}{2} \frac{f(x(t_k)) - f(x(t_{k-1}))}{\Delta t} \\ &= \Delta t \left[\frac{3}{2} f(x(t_k)) - \frac{1}{2} f(x(t_{k-1})) \right] \; . \end{aligned}$$

The second order Adams Bashforth method for constant Δt is

$$x_{k+1} = x_k + \Delta t \left[\frac{3}{2} f(x_k) - \frac{1}{2} f(x_{k-1}) \right] \; . \tag{8.42}$$

To program higher order Adams Bashforth methods we need to evaluate the integral of the interpolating polynomial. The techniques of polynomial interpolation from Chapter 7 make this simpler.

Adams Bashforth methods are attractive for high accuracy computations where stiffness is not an issue. They cannot be unstable in the way (8.39) is because setting $f \equiv 0$ results (in (8.42), for example) in $x_{k+1} = x_k$, as for Runge Kutta methods. Adams Bashforth methods of any order or accuracy require one evaluation of f per time step, as opposed to four per time step for the fourth order Runge Kutta method.

8.6 Implicit methods

Implicit methods use $f(x_{k+1})$ in the formula for x_{k+1}. They are used for stiff problems because they can be stable with large $\lambda \Delta t$ (see Section 8.3) in ways explicit methods, all the ones discussed up to now, cannot. An implicit method must solve a system of equations to compute x_{k+1}.

The simplest implicit method is *backward Euler*:

$$x_{k+1} = x_k + \Delta t f(x_{k+1}) \; . \tag{8.43}$$

This is only first order accurate, but it is stable for any λ and Δt if $\mathrm{Re}(\lambda) \leq 0$. This makes it suitable for solving stiff problems. It is called implicit because x_{k+1} is determined implicitly by (8.43), which we rewrite as

$$F(x_{k+1}, \Delta t) = 0 \;\; , \quad \text{where} \;\; F(y, \Delta t) = y - \Delta t f(y) - x_k \; , \tag{8.44}$$

To find x_{k+1}, we have to solve this system of equations for y.

Applied to the linear scalar problem (8.31) (dropping the α index), the method (8.43) becomes $u_{k+1} = u_k + \Delta t \lambda u_{k+1}$, or

$$u_{k+1} = \frac{1}{1 - \Delta t \lambda} u_k \; .$$

This shows that $|u_{k+1}| < |u_k|$ if $\Delta t > 0$ and λ is any complex number with $\mathrm{Re}(\lambda) \leq 0$. This is in partial agreement with the qualitative behavior of the exact solution of (8.31), $u(t) = e^{\lambda t} u(0)$. The exact solution decreases in time if $\mathrm{Re}(\lambda) < 0$ but not if $\mathrm{Re}(\lambda) = 0$. The backward Euler approximation decreases in time even when $\mathrm{Re}(\lambda) = 0$. The backward Euler method artificially stabilizes a neutrally stable system, just as the forward Euler method artificially destabilizes it (see the modified equation discussion leading to (8.19)).

Most likely the equations (8.44) would be solved using an iterative method as discussed in Chapter 6. This leads to *inner iterations*, with the *outer* iteration being the time step. If we use the unsafeguarded local Newton method, and let j index the inner iteration, we get $F' = I - \Delta t f'$ and

$$y_{j+1} = y_j - \Big(I - \Delta t f'(y_j)\Big)^{-1} \big(y_j - \Delta t f(y_j) - x_k\big) \; , \tag{8.45}$$

hoping that $y_j \to x_{k+1}$ as $j \to \infty$. We can take initial guess $y_0 = x_k$, or, even better, an extrapolation such as $y_0 = x_k + \Delta t (x_k - x_{k-1})/\Delta t = 2x_k - x_{k-1}$. With a good initial guess, just one Newton iteration should give x_{k+1} accurately enough.

Can we use the approximation $J \approx I$ for small Δt? If we could, the Newton iteration would become the simpler *functional iteration* (check this)

$$y_{j+1} = x_k + \Delta t f(y_j) \; . \tag{8.46}$$

The problem with this is that it does not work precisely for the stiff systems we use implicit methods for. For example, applied to $\dot{u} = \lambda u$, the functional iteration diverges ($|y_j| \to \infty$ as $j \to \infty$) for $\Delta t \lambda < -1$.

Most of the explicit methods above have implicit anologues. Among implicit Runge Kutta methods we mention the *trapezoid rule*

$$\frac{x_{k+1} - x_k}{\Delta t} = \frac{1}{2}\big(f(x_{k+1}) + f(x_k)\big) \; . \tag{8.47}$$

There are *backward differentiation formula*, or *BDF* methods based on higher order one sided approximations of $\dot{x}(t_{k+1})$. The second order BDF method uses (**??**):

$$\dot{x}(t) = \frac{1}{\Delta t}\left(\frac{3}{2}x(t) - 2x(t-\Delta t) + \frac{1}{2}x(t-2\Delta t)\right) + O\left(\Delta t^2\right) \ ,$$

to get

$$f(x(t_{k+1})) = \dot{x}(t_{k+1}) = \left(\frac{3}{2}x(t_{k+1}) - 2x(t_k) + \frac{1}{2}x(t_{k-1})\right) + O\left(\Delta t^2\right) \ ,$$

and, neglecting the $O\left(\Delta t^2\right)$ error term,

$$x_{k+1} - \frac{2\Delta t}{3} f(x_{k+1}) = \frac{4}{3}x_k - \frac{1}{3}x_{k-1} \ . \tag{8.48}$$

The *Adams Molton* methods estimate $\int_{t_k}^{t_{k+1}} f(x(t))dt$ using polynomial interpolation using the values $f(x_{k+1})$, $f(x_k)$, and possibly $f(x_{k-1})$, etc. The second order Adams Molton method uses $f(x_{k+1})$ and $f(x_k)$. It is the same as the trapezoid rule (8.47). The third order Adams Molton method also uses $f(x_{k-1})$. Both the trapezoid rule (8.47) and the second order BDF method (8.48) both have the property of being *A-stable*, which means being stable for (8.31) with any λ and Δt as long as $\mathrm{Re}(\lambda) \leq 0$. The higher order implicit methods are more stable than their explicit counterparts but are not A stable, which is a constant frustration to people looking for high order solutions to stiff systems.

8.7 Computing chaos, can it be done?

In many applications, the solutions to the differential equation (8.1) are *chaotic*.[13] The informal definition is that for large t (not so large in real applications) $x(t)$ is an unpredictable function of $x(0)$. In the terminology of Section 8.5, this means that the solution operator, $S(x_0, 0, t)$, is an ill conditioned function of x_0.

The dogma of Section 2.7 is that a floating point computation cannot give an accurate approximation to S if the condition number of S is larger than $1/\epsilon_{mach} \sim 10^{16}$. But practical calculations ranging from weather forcasting to molecular dynamics violate this rule routinely. In the computations below, the condition number of $S(x,t)$ increases with t and crosses 10^{16} by $t = 3$ (see Figure 8.3). Still, a calculation up to time $t = 60$ (Figure 8.4, bottom), shows the beautiful butterfly shaped *Lorentz attractor*, which looks just as it should.

On the other hand, in this and other computations, it truly is impossible to calculate details correctly. This is illustrated in Figure 8.2. The top picture

[13] James Glick has written a nice popular book on chaos. Neil Strogatz has a more technical introduction that does not avoid the beautiful parts.

plots two trajectories, one computed with $\Delta t = 4 \times 10^{-4}$ (dashed line), and the other with the time step reduced by a factor of 2 (solid line). The difference between the trajectories is an estimate of the accuracy of the computations. The computation seems somewhat accurate (curves close) up to time $t \approx 5$, at which time the dashed line goes up to $x \approx 15$ and the solid line goes down to $x \approx -15$. At least one of these is completely wrong. Beyond $t \approx 5$, the two "approximate" solutions have similar qualitative behavior but seem to be independent of each other. The bottom picture shows the same experiment with Δt a hundred times smaller than in the top picture. With a hundred times more accuracy, the approximate solution loses accuracy at $t \approx 10$ instead of $t \approx 5$. If a factor of 100 increase in accuracy only extends the validity of the solution by 5 time units, it should be hopeless to compute the solution out to $t = 60$.

The present numerical experiments are on the *Lorentz equations*, which are a system of three nonlinear ordinary differential equations

$$
\begin{aligned}
\dot{x} &= \sigma(y - x) \\
\dot{y} &= x(\rho - z) - y \\
\dot{z} &= xy - \beta z
\end{aligned}
$$

with[14] $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$. The C/C++ program outputs (x, y, z) for plotting every $t = .02$ units of time, though there many time steps in each plotting interval. The solution first finds its way to the butterfly shaped *Lorentz attractor* then stays on it, travelling around the right and left wings in a seemingly random (technically, *chaotic*) way. The initial data $x = y = z = 0$ are not close to the attractor, so we ran the differential equations for some time before time $t = 0$ in order that $(x(0), y(0), z(0))$ should be a typical point on the attractor. Figure 8.2 shows the chaotic sequence of wing choice. A trip around the left wing corresponds to a dip of $x(t)$ down to $x \approx -15$ and a trip around the right wing corresponds to x going up to $x \approx 15$.

Sections 2.7 and 4.3 explain that the condition number of the problem of calculating $S(x, t)$ depends on the Jacobian matrix $A(x, t) = \partial_x S(x, t)$. This represents the sensitivity of the solution at time t to small perturbations of the initial data. Adapting notation from (4.29), we find that the condition number of calculating $x(t)$ from initial conditions $x(0) = x$ is

$$
\kappa(S(x, t)) = \|A(x(0), t)\| \frac{\|x(0)\|}{\|x(t)\|} . \tag{8.49}
$$

We can calculate $A(x, t)$ using ideas of perturbation theory similar to those we used for linear algebra problems in Section 4.2.6. Since $S(x, t)$ is the value of a solution at time t, it satisfies the differential equation

$$
f(S(x, t)) = \frac{d}{dt} S(x, t) .
$$

[14]These can be found, for example, in http://wikipedia.org by searching on "Lorentz attractor".

We differentiate both sides with respect to x and interchange the order of differentiation,

$$\frac{\partial}{\partial x} f((S(x,t)) = \frac{\partial}{\partial x}\frac{d}{dt} S(x,t) = \frac{d}{dt}\frac{\partial}{\partial x} S(x,t) = \frac{d}{dt} A(x,t) \ ,$$

to get (with the chain rule)

$$\begin{aligned} \frac{d}{dt} A(x,t) &= \frac{\partial}{\partial x} f(S(x,t)) \\ &= f'(S(x,t)) \cdot \partial_x S \\ \dot{A} &= f'(S(x,t)) A(x,t) \ . \end{aligned} \tag{8.50}$$

Thus, if we have an initial value x and calculate the trajectory $S(x,t)$, then we can calculate the *first variation*, $A(x,t)$, by solving the linear initial value problem (8.50) with initial condition $A(x,0) = I$ (why?). In the present experiment, we solved the Lorentz equations and the perturbation equation using forward Euler with the same time step.

In typical chaotic problems, the first variation grows exponentially in time. If $\sigma_1(t) \geq \sigma_2(t) \geq \cdots \geq \sigma_n(t)$ are the singular values of $A(x,t)$, then there typically are *Lyapounov exponents*, μ_k, so that

$$\sigma_k(t) \sim e^{\mu_k t} \ ,$$

More precisely,

$$\lim_{t\to\infty} \frac{\ln(\sigma_k(t))}{t} = \mu_k \ .$$

In our problem, $\|A(x,t)\| = \sigma_1(t)$ seems to grow exponentially because $\mu_1 > 0$. Since $\|x = x(0)\|$ and $\|x(t)\|$ are both of the order of about ten, this, with (8.49), implies that $\kappa(S(x,t))$ grows exponentially in time. That explains why it is impossible to compute the values of $x(t)$ with any precision at all beyond $t = 20$.

It is interesting to study the condition number of A itself. If $\mu_1 > \mu_n$, the l^2 this also grows exponentially,

$$\kappa_{l^2}(A(x,t)) = \frac{\sigma_1(t)}{\sigma_n(t)} \sim e^{(\mu_1 - \mu_n)t} \ .$$

Figure 8.3 gives some indication that our Lorentz system has differing Lyapoounov exponents. The top figure shows computations of the three singular values for $A(x,t)$. For $0 \leq t <\approx 2$, it seems that σ_3 is decaying exponentially, making a downward sloping straight line on this log plot. When σ_3 gets to about 10^{-15}, the decay halts. This is because it is nearly impossible for a full matrix in double precision floating point to have a condition number larger than $1/\epsilon_{mach} \approx 10^{16}$. When σ_3 hits 10^{-15}, we have $\sigma_1 \sim 10^2$, so this limit has been reached. These trends are clearer in the top frame of Figure 8.4, which is the same calculation carried to a larger time. Here $\sigma_1(t)$ seems to be growing exponentially with a

Figure 8.2: *Two convergence studies for the Lorentz system. The time steps in the bottom figure are 100 times smaller than the time steps in the to figure. The more accurate calculation loses accuracy at $t \approx 10$, as opposed to $t \approx 5$ with a larger time step. The qualitative behavior is similar in all computations.*

Figure 8.3: *Computed singluar values of the sensitivity matrix $A(x,t) = \partial_x S(x,t)$ with large time step (top) and ten times smaller time step (bottom). Top and bottom curves are similar qualitatively though the fine details differ. Theory predicts that middle singular value should be not grow or decay with t. The times from Figure 8.2 at which the numerical solution loses accuracy are not apparent here. In higher precision arithmetic, $\sigma_3(t)$ would have continued to decay exponentially. It is unlikely that computed singular values of any full matrix would differ by less than a factor of $1/\epsilon_{mach} \approx 10^{16}$.*

gap between σ_1 and σ_3 of about $1/\epsilon_{mach}$. Theory says that σ_2 should be close to one, and the computations are consistent with this until the condition number bound makes $\sigma_2 \sim 1$ impossible.

To summarize, some results are quantitatively right, including the butterfly shape of the attractor and the exponential growth rate of $\sigma_1(t)$. Some results are qualitatively right but quantitatively wrong, including the values of $x(t)$ for $t >\approx 10$. Convergence analyses (comparing Δt results to $\Delta t/2$ results) distinguishes right from wrong in these cases. Other features of the computed solution are consistent over a range of Δt and consistently wrong. There is no reason to think the condition number of $A(x,t)$ grows exponentially until $t \sim 2$ then levels off at about 10^{16}. Much more sophisticated computational methods using the semigroup property show this is not so.

8.8 Software: Scientific visualization

Visualization of data is indispensable in scientific computing and computational science. Anomolies in data that seem to jump off the page in a plot are easy to overlook in numerical data. It can be easier to interpret data by looking at pictures than by examining columns of numbers. For example, here are entries

Figure 8.4: *Top: Singular values from Figure 8.3 computed for longer time. The $\sigma_1(t)$ grows exponentially, making a straight line on this log plot. The computed $\sigma_2(t)$ starts growing with the same exponential rate as σ_1 when roundoff takes over. A correct computation would show $\sigma_3(t)$ decreasing exponentially and $\sigma_2(t)$ neither growing nor decaying. Bottom: A beautiful picture of the butterfly shaped Lorentz attractor. It is just a three dimensional plot of the solution curve.*

500 to 535 from the time series that made the top curve in the top frame of Figure 8.4 (multiplied by 10^{-5}).

0.1028	0.1020	0.1000	0.0963	0.0914	0.0864	0.0820
0.0790	0.0775	0.0776	0.0790	0.0818	0.0857	0.0910
0.0978	0.1062	0.1165	0.1291	0.1443	0.1625	0.1844
0.2104	0.2414	0.2780	0.3213	0.3720	0.4313	0.4998
0.5778	0.6649	0.7595	0.8580	0.9542	1.0395	1.1034

Looking at the numbers, we get the overall impression that they are growing in an irregular way. The graph shows that the numbers have simple exponential growth with fine scale irregularities superposed. It could take hours to get that information directly from the numbers.

It can be a challenge to make visual sense of higher dimensional data. For example, we could make graphs of $x(t)$ (Figure 8.2), $y(t)$ and $z(t)$ as functions of t, but the single three dimensional plot in the lower frame of Figure 8.4 makes is clearer that the solution goes sometimes around the left wing and sometimes around the right. The three dimensional plot (`plot3` in Matlab) illuminates the structure of the solution better than three one dimensional plots.

There are several ways to visualize functions of two variables. A *contour plot* draws several *contour lines*, or *level lines*, of a function $u(x,y)$. A contour line for level u_k is the set of points (x,y) with $u(x,y) = u_k$. It is common to take about ten uniformly spaced values u_k, but most contour plot programs, including the Matlab program `contour`, allow the user to specify the u_k. Most contour lines are smooth curves or collections of curves. For example, if $u(x,y) = x^2 - y^2$, the contour line $u = u_k$ with $u_k \neq 0$ is a hyperbola with two components. An exception is $u_k = 0$, the contour line is an $\times$.

A *grid plot* represents a two dimensional rectangular array of numbers by colors. A *color map* assigns a color to each numerical value, that we call $c(u)$. In practice, usually we specify $c(u)$ by giving RGB values, $c(u) = (r(u), g(u), b(u))$, where r, g, and b are the intensities for red, green and blue respectively. These may be integers in a range (e.g. 0 to 255) or, as in Matlab, floating point numbers in the range from 0 to 1. Matlab uses the commands `colormap` and `image` to establish the color map and display the array of colors. The image is an array of boxes. Box (i,j) is filled with the color $c(u(i,j))$.

Surface plots visualize two dimensional surfaces in three dimensional space. The surface may be the graph of $u(x,y)$. The Matlab commands `surf` and `surfc` create surface plots of graphs. These look nice but often are harder to interpret than contour plots or grid plots. It also is possible to plot *contour surfaces* of a function of three variables. This is the set of (x,y,z) so that $u(x,y,z) = u_k$. Unfortunately, it is hard to plot more than one contour surface at a time.

Movies help visualize time dependent data. A movie is a sequence of frames, with each frame being one of the plots above. For example, we could visualize the Lorentz attractor with a movie that has the three dimensional butterfly together with a dot representing the position at time t.

The default in Matlab, and most other quality visualization packages, is to render the user's data as explicitly as possible. For example, the Matlab command `plot(u)` will create a piecewise linear "curve" that simply connects successive data points with straight lines. The plot will show the granularity of the data as well as the values. Similarly, the grid lines will be clearly visible in a color grid plot. This is good most of the time. For example, the bottom frame of Figure 8.4 clearly shows the granularity of the data in the wing tips. Since the curve is sampled at uniform time increments, this shows that the trajectory is moving faster at the wing tips than near the body where the wings meet.

Some plot packages offer the user the option of smoothing the data using spline interpolation before plotting. This might make the picture less angular, but it can obscure features in the data and introduce artifacts, such as overshoots, not present in the actual data.

8.9 Resources and further reading

There is a beautiful discussion of computational methods for ordinary differential equations in *Numerical Methods* by øA]ke Bjöork and Germund Dahlquist. It was Dahlquist who created much of our modern understanding of the subject. A more recent book is *A First Course in Numerical Analysis of Differential Equations* by Arieh Iserles. The book *Numerical Solution of Ordinary Differential Equations* by Lawrence Shampine has a more practical orientation.

There is good public domain software for solving ordinary differential equations. A particularly good package is LSODE (google it).

The book by Sans-Serna explains symplectic integrators and their application to large scale problems such as the dynamics of large scape biological molecules. It is an active research area to understand the quantitative relationship between long time simulations of such systems and the long time behavior of the systems themselves. Andrew Stuart has written some thoughtful oapers on the subject.

The numerical solution of partial differential equations is a vast subject with many deep specialized methods for different kinds of problems. For computing stresses in structures, the current method of choice seems to be *finite element* methods. For fluid flow and wave propagation (particularly nonlinear), the majority relies on finite difference and finite volume methods. For finite differences, the old book by Richtmeyer and Morton still merit though there are more up to date books by Randy LeVeque and by Bertil Gustavson, Heinz Kreiss, and Joe Oliger.

8.10 Exercises

1. We compute the second error correction $u_2(t)$ n (8.13). For simplicity, consider only the scalar equation ($n = 1$). Assuming the error expansion, we have

$$\begin{aligned} f(x_k) &= f(\widetilde{x}_k + \Delta t u_1(t_k) + \Delta t^2 u_2(t_k) + O(\Delta t^3)) \\ &\approx f(\widetilde{x}_k) + f'(\widetilde{x}_k)\left(\Delta t u_1(t_k) + \Delta t^2 u_2(t_k)\right) \end{aligned}$$

$$+\frac{1}{2}f''(\widetilde{x}_k)\Delta t^2 u_1(t_k)^2 + O\left(\Delta t^3\right) \ .$$

Also

$$\frac{x(t_k+\Delta t)-x(t_k)}{\Delta t} = \dot{x}(t_k) + \frac{\Delta t}{2}\ddot{x}(t_k) + \frac{\Delta t^2}{6}x^{(3)}(t_k) + O\left(\Delta t^3\right) \ ,$$

and

$$\Delta t \frac{u_1(t_k+\Delta t)-u_1(t_k)}{\Delta t} = \Delta t \dot{u}_1(t_k) + \frac{\Delta t^2}{2}\ddot{u}_1(t_k) + O\left(\Delta t^3\right) \ .$$

Now plug in (8.13) on both sides of (8.5) and collect terms proportional to Δt^2 to get

$$\dot{u}_2 = f'(x(t))u_2 + \frac{1}{6}x^{(3)}(t) + \frac{1}{2}f''(x(t))u_1(t)^2 + ??? \ .$$

2. This exercise confirms what was hinted at in Section 8.1, that (8.19) correctly predicts error growth even for t so large that the solution has lost all accuracy. Suppose $k = R/\Delta t^2$, so that $t_k = R/\Delta t$. The error equation (8.19) predicts that the forward Euler approximation x_k has grown by a factor of $e^R/2$ although the exact solution has not grown at all. We can confirm this by direct calculation. Write the forward Euler approximation to (8.18) in the form $x_{k+1} = Ax_k$, where A is a 2×2 matrix that depends on Δt. Calculate the eigenvalues of A up to second order in Δt: $\lambda_1 = 1 + i\Delta t + a\Delta t^2 + O(\Delta t^3)$, and $\lambda_2 = 1 - i\Delta t + b\Delta t^2 + O(\Delta t^3)$. Find the constants a and b. Calculate $\mu_1 = \ln(\lambda_1) = i\Delta t + c\Delta t^2 + O(\Delta t^3)$ so that $\lambda_1 = \exp(i\Delta t + c\Delta t^2 + O(\Delta t^3))$. Conclude that for $k = R/\Delta t^2$, $\lambda_1^k = \exp(k\mu_1) = e^{iR/\Delta t}e^{R/2+O(\Delta t)}$, which shows that the solution has grown by a factor of nearly $e^{R/2}$ as predicted by (8.19). This s**t is good for something!

3. Another two stage second order Runge Kutta method sometimes is called the *modified Euler* method. The first stage uses forward Euler to predict the x value at the middle of the time step: $\xi_1 = \frac{\Delta t}{2}f(x_k, t_k)$ (so that $x(t_k + \Delta t/2) \approx x_k + \xi_1$). The second stage uses the midpoint rule with that estimate of $x(t_k + \Delta t/2)$ to step to time t_{k+1}: $x_{k+1} = x_k + \Delta t f(t_k + \frac{\Delta t}{2}, x_k + \xi_1)$. Show that this method is second order accurate.

4. Show that applying the four stage Runge Kutta method to the linear system (8.30) is equivalent to approximating the fundamental solution $S(\Delta t) = \exp(\Delta t A)$ by its Taylor series in Δt up to terms of order Δt^4 (see Exercise **??**). Use this to verify that it is fourth order for linear problems.

5. Write a C/C++ program that solves the initial value problem for (8.1), with f independent of t, using a constant time step, Δt. The arguments to the initial value problem solver should be T (the final time), Δt (the time step), $f(x)$ (specifying the differential equations), n (the number of

components of x), and x_0 (the initial condition). The output should be the apaproximation to $x(T)$. The code must do something to preserve the overall order of accuracy of the method in case T is not an integer multiple of Δt. The code should be able to switch between the three methods, forward Euler, second order Adams Bashforth, forth order four state Runge Kutta, with a minimum of code editing. Hand in output for each of the parts below showing that the code meets the specifications.

(a) The procedure should return an error flag or notify the calling routine in some way if the number of time steps called for is negative or impossibly large.

(b) For each of the three methods, verify that the coding is correct by testing that it gets the right answer, $x(.5) = 2$, for the scalar equation $\dot{x} = x^2$, $x(0) = 1$.

(c) For each of the three methods and the test problem of part b, do a convergence study to verify that the method achieves the theoretical order of accuracy. For this to work, it is necessary that T should be an integer multiple of Δt.

(d) Apply your code to problem (8.18) with initial data $x_0 = (1, 0)^*$. Repeat the convergence study with $T = 10$.

6. Verify that the recurrence relation (8.39) is unstable.

(a) Let z be a complex number. Show that the sequence $x_k = z^k$ satisfies (8.39) if and only if z satisfies $0 = p(z) = z^3 + \frac{3}{2}z^2 - 3z + \frac{1}{2}$.

(b) Show that $x_k = 1$ for all k is a solution of the recurrence relation. Conclude that $z = 1$ satisfies $p(1) = 0$. Verify that this is true.

(c) Using polynomial division (or another method) to factor out the known root $z = 1$ from $p(z)$. That is, find a quadratic polynomial, $q(z)$, so that $p(z) = (z - 1)q(z)$.

(d) Use the quadratic formula and a calculator to find the roots of q as $z = \frac{-5}{4} \pm \sqrt{\frac{41}{16}} \approx -2.85,\ .351$.

(e) Show that the general formula $x_k = az_1^k + bz_2^k + cz_3^k$ is a solution to (8.39) if z_1, z_2, and z_3 are three roots $z_1 = 1$, $z_2 \approx -2.85$, $z_3 \approx .351$, and, conversely, the general solution has this form. Hint: we can find a, b, c by solving a vanderMonde system (Section 7.4) using x_0, x_1, and x_2.

(f) Assume that $|x_0| \leq 1$, $|x_1| \leq 1$, and $|x_2| \leq 1$, and that b is on the order of double precision floating point roundoff (ϵ_{mach}) relative to a and c. Show that for $k > 80$, x_k is within ϵ_{mach} of bz_2^k. Conclude that for $k > 80$, the numerical solution has nothing in common with the actual solution $x(t)$.

7. Applying the implicit trapezoid rule (8.47) to the scalar model problem (8.31) results in $u_{k+1} = m(\lambda \Delta t) u_k$. Find the formula for m and show that $|m| \leq 1$ if $\mathrm{Re}(\lambda) \leq 0$, so that $|u_{k+1}| \leq |u_k|$. What does this say about the applicibility of the trapezoid rule to stiff problems?

8. Exercise violating time step constraint.

9. Write an adaptive code in C/C++ for the initial value problem (8.1) (8.2) using the method described in Section 8.4 and the four stage fourth order Runge Kutta method. The procedure that does the solving should have arguments describing the problem, as in Exercise 5, and also the local truncation error level, e. Apply the method to compute the trajectory of a comet. In nondimensionalized variables, the equations of motion are given by the inverse square law:

$$\frac{d^2}{dt^2} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} = \frac{-1}{(r_1^2 + r_2^2)^{3/2}} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \quad .$$

We always will suppose that the comet starts at $t = 0$ with $r_1 = 10$, $r_2 = 0$, $\dot{r}_1 = 0$, and $\dot{r}_2 = v_0$. If v_0 is not too large, the point $r(t)$ traces out an ellipse in the plane[15]. The shape of the ellipse depends on v_0. The *period*, $P(v_0)$, is the first time for which $r(P) = r(0)$ and $\dot{r}(P) = \dot{r}(0)$. The solution $r(t)$ is *periodic* because it has a period.

(a) Verify the correctness of this code by comparing the results to those from the fixed time step code from Exercise 5 with $T = 30$ and $v_0 = .2$.

(b) Use this program, with a small modification to compute $P(v_0)$ in the range $.01 \leq v_0 \leq .5$. You will need a criterion for telling when the comet has completed one orbit since it will not happen that $r(P) = r(0)$ exactly. Make sure your code tests for and reports failure[16].

(c) Choose a value of v_0 for which the orbit is rather but not extremely elliptical (width about ten times height). Choose a value of e for which the solution is rather but not extremely accurate – the error is small but shows up easily on a plot. If you set up your environment correctly, it should be quick and easy to find suitable paramaters by trial and error using Matlab graphics. Make a plot of a single period with two curves on one graph. One should be a solid line representing a highly accurate solution (so accurate that the error is smaller than the line width – *plotting accuracy*), and the other being the modestly accurate solution, plotted with a little "o" for each time step. Comment on the distribution of the time step points.

[15] Isaac Newton formulated these equations and found the explicit solution. Many aspects of planetary motion – elliptical orbits, sun at one focus, $|r|\,\dot{\theta} = const$ – had beed discovered observationally by Johannes Keppler. Newton's inverse square law *theory* fit Keppler's *data*.

[16] This is not a drill.

(d) For the same parameters as part b, make a single plot of that contains three curves, an accurate computation of $r_1(t)$ as a function of t (solid line), a modestly accurate computation of r_1 as a function of t ("o" for each time step), and Δt as a function of t. You will need to use a different scale for Δt if for no other reason than that it has different units. Matlab can put one scale in the left and a different scale on the right. It may be necessary to plot Δt on a log scale if it varies over too wide a range.

(e) Determine the number of adaptive time stages it takes to compute $P(.01)$ to .1% accuracy (error one part in a thousand) and how many fixed Δt time step stages it takes to do the same. The counting will be easier if you do it within the function f.

10. The vibrations of a two dimensional crystal lattice may be modelled in a crude way using the differential equations[17]

$$\ddot{r}_{jk} = r_{j-1,k} + r_{j+1,k} + r_{j,k-1} + r_{j,k+1} - 4r_{jk} \; . \tag{8.51}$$

Here $r_{jk}(t)$ represents the displacement (in the vertical direction) of an atom at the (j,k) location of a square crystal lattice of atoms. Each atom is *bonded* to its four neighbors and is pulled toward them with a linear force. A lattice of size L has $1 \leq j \leq L$ and $1 \leq k \leq L$. Apply *reflecting* boundary conditions along the four boundaries. For example, the equation for $r_{1,k}$ should use $r_{0,k} = r_{1,k}$. This is easy to implement using a *ghost cell* strategy. Create ghost cells along the boundary and copy appropriate values from the actual cells to the ghost cells before each evaluation of f. This allows you to use the formula (8.51) at every point in the lattice. Start with initial data $r_{jk}(0) = 0$ and $\dot{r}_{jk}(0) = 0$ for all j, k except $\dot{r}_{1,1} = 1$. Compute for $L = 100$ and $T = 300$. Use the fourth order Runge Kutta method with a fixed time step $\Delta t = .01$. Write the solution to a file every .5 time units then use Matlab to make a movie of the results, with a 2D color plot of the solution in each frame. The movie should be a circular wave moving out from the bottom left corner and bouncing off the top and right boundaries. There should be some beautiful wave patterns inside the circle that will be hard to see far beyond time $t = 100$. Hand in a few of your favorite stills from the movie. If you have a web site, post your movie for others to enjoy.

[17] This is one of Einstein's contributions to science.

Chapter 9

Monte Carlo methods

Monte Carlo means using random numbers in scientific computing. More precisely, it means using random numbers as a tool to compute something that is not random. For example[1], let X be a random variable and write its expected value as $A = E[X]$. If we can generate $X_1, \ldots, X_n$, n independent random variables with the same distribution, then we can make the approximation

$$A \approx \widehat{A}_n = \frac{1}{n} \sum_{k=1}^{n} X_k \; .$$

The *strong law of large numbers* states that $\widehat{A}_n \to A$ as $n \to \infty$. The X_k and $\widehat{A}_n$ are random and (depending on the seed, see Section 9.2) could be different each time we run the program. Still, the target number, A, is not random.

We emphasize this point by distinguishing between Monte Carlo and *simulation*. Simulation means producing random variables with a certain distribution just to look at them. For example, we might have a model of a random process that produces clouds. We could simulate the model to generate cloud pictures, either out of scientific interest or for computer graphics. As soon as we start asking quantitative questions about, say, the average size of a cloud or the probability that it will rain, we move from pure simulation to Monte Carlo.

The reason for this distinction is that there may be other ways to define A that make it easier to estimate. This process is called *variance reduction*, since most of the error in $\widehat{A}$ is *statistical*. Reducing the variance of $\widehat{A}$ reduces the statistical error.

We often have a choice between Monte Carlo and deterministic methods. For example, if X is a one dimensional random variable with probability density $f(x)$, we can estimate $E[X]$ using a panel integration method, see Section 3.4. This probably would be more accurate than Monte Carlo because the Monte Carlo error is roughly proportional to $1/\sqrt{n}$ for large n, which gives it order of accuracy roughly $\frac{1}{2}$. The worst panel method given in Section 3.4 is first order accurate. The general rule is that deterministic methods are better than Monte Carlo in any situation where the determinist method is practical.

We are driven to resort to Monte Carlo by the "curse of dimensionality". The curse is that the work to solve a problem in many dimensions may grow exponentially with the dimension. Suppose, for example, that we want to compute an integral over ten variables, an integration in ten dimensional space. If we approximate the integral using twenty points in each coordinate direction, the total number of integration points is $20^{10} \approx 10^{13}$, which is on the edge of what a computer can do in a day. A Monte Carlo computation might reach the same accuracy with only, say, 10^6 points. People often say that the number of points needed for a given accuracy in Monte Carlo does not depend on the dimension, and there is some truth to this.

One favorable feature of Monte Carlo is that it is possible to estimate the order of magnitude of statistical error, which is the dominant error in most Monte Carlo computations. These estimates are often called "error bars" because of

[1] Section **??** has a quick review of the probability we use here.

the way they are indicated on plots of Monte Carlo results. Monte Carlo error bars are essentially statistical confidence intervals. Monte Carlo practitioners are among the avid consumers of statistical analysis techniques.

Another feature of Monte Carlo that makes academics happy is that simple clever ideas can lead to enormous practical improvements in efficiency and accuracy (which are basically the same thing). This is the main reason I emphasize so strongly that, while A is given, the algorithm for estimating it is not. The search for more accurate alternative algorithms is often called "variance reduction". Common variance reduction techniques are importance sampling, antithetic variates, and control variates.

Many of the examples below are somewhat artificial because I have chosen not to explain specific applications. The techniques and issues raised here in the context of toy problems are the main technical points in many real applications. In many cases, we will start with the probabilistic definition of A, while in practice, finding this is part of the problem. There are some examples in later sections of choosing alternate definitions of A to improve the Monte Carlo calculation.

9.1 Quick review of probability

This is a quick review of the parts of probability needed for the Monte Carlo material discussed here. Please skim it to see what notation we are using and check Section 9.6 for references if something is unfamiliar.

Probability theory begins with the assumption that there are *probabilities* associated to *events*. Event B has probability $\Pr(B)$, which is a number between 0 and 1 describing what fraction of the time event B would happen if we could repeat the *experiment* many times. The exact meaning of probabilities is debated at length elsewhere.

An event is a set of possible outcomes. The set of all possible outcomes is called Ω and particular outcomes are called ω. Thus, an event is a subset of the set of all possible outcomes, $B \subseteq \Omega$. For example, suppose the experiment is to toss four coins (a penny, a nickle, a dime, and a quarter) on a table and record whether they were face up (*heads*) or face down (*tails*). There are 16 possible outcomes. The notation $THTT$ means that the penny was face down (tails), the nickle was up, and the dime and quareter were down. The event "all heads" consists of a single outcome, $HHHH$. The event $B =$ "more heads than tails" consists of the five outcomes: $B = \{HHHH, THHH, HTHH, HHTH, HHHT\}$.

The basic set operations apply to events. For example the intersection of events B and c is the set of outcomes both in B and in C: $\omega \in B \cap C$ means $\omega \in B$ and $\omega \in C$. For that reason, $B \cap C$ represents the event "B and C". For example, if B is the event "more heads than tails" above and C is the event "then dime was heads", then C has 8 outcomes in it, and $B \cap C = \{HHHH, THHH, HTHH, HHHT\}$. The set union, $B \cup C$ is $\omega \in B \cup C$ if $\omega \in B$ or $\omega \in C$, so we call it "B or C". One of the axioms of probability is

$$\Pr(B \cup C) = \Pr(B) + \Pr(C) \;\;,\;\; \text{if} \quad B \cap C \;\; \text{is empty.}$$

Another axiom is that

$$0 \leq \Pr(B) \leq 1 \ , \ \text{ for any event } B.$$

These have many intuitive consequences, such as

$$B \subset C \Longrightarrow \Pr(B) \leq \Pr(C) \ .$$

A final axiom is $\Pr(\Omega) = 1$; for sure something will happen. This really means that Ω includes every possible outcome.

The *conditional* "probability of B given C" is given by Bayes' formula:

$$\Pr(B \mid C) = \frac{\Pr(B \cap C)}{\Pr(C)} = \frac{\Pr(B \text{ and } C)}{\Pr(C)} \tag{9.1}$$

Intuitively, this is the probability that a C outcome also is a B outcome. If we do an experiment and $\omega \in C$, $\Pr(B \mid C)$ is the probability that $\omega \in B$. The right side of (9.1) represents the fraction of C outcomes that also are in B. We often know $\Pr(C)$ and $\Pr(B \mid C)$, and use (9.1) to calculate $\Pr(B \cap C)$. Of course, $\Pr(C \mid C) = 1$. Events B and C are *independent* if $\Pr(B) = \Pr(B \mid C)$, which is the same as $\Pr(B \mid C) = \Pr(B)\Pr(C)$, so B being independent of C is the same as C being independent of B.

The *probability space* Ω is finite if it is possible to make a finite list[2] of all the outcomes in Ω: $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$. The space is *countable* if it is possible to make a possibly infinite list of all the elements of Ω: $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n, \ldots\}$. We call Ω *discrete* in both cases. When Ω is discrete, we can specify the probabilities of each outcome

$$f_k = \Pr(\omega = \omega_k) \ .$$

Then an event B has probability

$$\Pr(B) = \sum_{\omega_k \in B} \Pr(\omega_k) = \sum_{\omega_k \in B} f_k \ .$$

A discrete random variable[3] is a number, $X(\omega)$, that depends on the random outcome, ω. In the coin tossing example, $X(\omega)$ could be the number of heads. The *expected value* is (defining $x_k = X(\omega_k)$)

$$E[X] = \sum_{\omega \in \Omega} X(\omega)\Pr(\omega) = \sum_{\omega_k} x_k f_k \ .$$

The *probability distribution* of a *continuous* random variable is described by a *probability density function*, or *PDF*, $f(x)$. If $X \in R^n$ is an n component random vector and $B \subseteq R^n$ is an event, then

$$\Pr(B) = \int_{x \in B} f(x) dx \ .$$

[2]Warning: this list is impractically large even in common simple applications.

[3]Warning: sometimes ω is the random variable and X is a function of a random variable.

In one dimension, we also have the *cumulative distribution function*, or[4] *CDF*: $F(x) = \Pr(X \leq x) = \int_{-\infty}^{x} f(x)dx$. For $a < b$ we have $\Pr(a \leq x \leq b) = F(b) - F(a)$. Also $f(x) = \frac{d}{dx}F(x)$, so we may write, informally,

$$\Pr(x \leq X \leq x + dx) = F(x + dx) - F(x) = f(x)dx \; . \tag{9.2}$$

The expected value is

$$\mu = E[X] = \int_{R^n} xf(x)dx \; .$$

in more than one dimension, both sides are n component vectors with components

$$\mu_k = E[X_k] = \int_{R^n} x_k f(x)dx \; .$$

In one dimension, the *variance* is

$$\sigma^2 = \mathrm{var}(X) = E\left[(X - \mu)^2\right] = \int_{-\infty}^{\infty} (x - \mu)^2 f(x)dx \; .$$

It is helpful not to distinguish between discrete and continuous random variables in general discussions. For example, we write $E[X]$ for the expected value in either case. The probability distribution, or probability *law*, of X is its probability density in the continuous case or the discrete probabilities in the discrete case. We write $X \sim f$ to indicate that f is the probability density, or the discrete probabilities, of X. We also write $X \sim X'$ to say that X and X' have the same probability distribution. If X and X' also are independent, we say they are iid, for *independent and identically distributed*. The goal of a simple sampler (Section 9.3 below) is generating a sequence $X_k \sim X$. We call these *samples* of the distribution X.

In more than one dimension, there is the symmetric $n \times n$ *variance/covariance* matrix (usually called *covariance matrix*),

$$C = E\left[(X - \mu)(X - \mu)^*\right] = \int_{R^n} (x - \mu)(x - \mu)^* f(x)dx \; , \tag{9.3}$$

whose entries are the individual covariances

$$C_{jk} = E\left[(X_j - \mu_j)(X_k - \mu_k)\right] = \mathrm{cov}\left[X_j, X_k\right] \; .$$

The covariance matrix is positive semidefinite in the sense that for any $y \in R^n$, $y^*Cy \geq 0$. This follows from (9.3):

$$\begin{aligned} y^*Cy &= y^* \left(E\left[(X - \mu)(X - \mu)^*\right]\right) y \\ &= E\left[\big(y^*(X - \mu)\big)\big((X - \mu)^*y\big)\right] \\ &= E\left[W^2\right] \geq 0 \; , \end{aligned}$$

[4] A common convention is to use capital letters for random variables and the corresponding lower case letter to represent values of that variable.

where $W = y^*(X-\mu)$. In order for C not to be positive definite, there must be a $y \neq 0$ that has $E\left[W^2\right] = 0$, which means that[5] W is identically zero. This means that the random vector X always lies on the hyperplane in R^n defined by $y^*x = y^*\mu$.

Let $Z = (X, Y)$ be an $n + m$ dimensional random variable with X being the first n components and Y being the last m, with probability density $f(z) = f(x, y)$. The *marginal* density for X is $g(x) = \int_{y \in R^m} f(x, y)dy$, and the marginal for Y is $h(y) = \int_x f(x, y)dx$. The random variables X and Y are *independent* if $f(x, y) = g(x)h(y)$. This is the same as saying that any event depending only on X is independent of any event depending only on[6] Y. The *conditional probability density* is the probability density for X alone once the value of Y is known:

$$f(x \mid y) = \frac{f(x, y)}{\int_{x' \in R^n} f(x', y)dx'} \; . \tag{9.4}$$

For $n = m = 1$ the informal version of this is the same as Bayes' rule (9.1). If B is the event $x \leq X \leq x + dx$ and C is the event $y \leq Y \leq y + dy$, then

$$\begin{aligned} f(x \mid y)dx &= \Pr(B \mid Y = y) \\ &= \Pr(B \mid y \leq Y \leq y + dy) \\ &= \frac{\Pr(B \text{ and } y \leq Y \leq y + dy)}{\Pr(y \leq Y \leq y + dy)} \end{aligned}$$

The numerator in the last line is $f(x, y)dxdy$ and the denominator is $\int_{x'} f(x', y)dx'dy$, which gives (9.4).

Suppose X is an n component random variable, $Y = u(X)$, and $g(y)$ is the probability density of Y. We can compute the expected value of Y in two ways

$$E[Y] = \int_y yg(y)dy = E[u(X)] = \int_x u(x)f(x)dx \; .$$

The one dimensional informal version is simple if u is one to one (like e^x or $1/x$, but not x^2). Then $y + dy$ corresponds to $u(x + dx) = u(x) + u'(x)dx$, so

$$\begin{aligned} g(y)dy &= Pr(y \leq Y \leq y + dy) \\ &= Pr(y \leq u(X) \leq y + dy \\ &= Pr(x \leq X \leq x + dx) \\ &= f(x)dx \quad \text{where } dx = \frac{dy}{u'(x)} \\ &= \frac{f(x)}{u'(x)}dy \; . \end{aligned}$$

[5]If W is a random variable with $E\left[W^2\right] = 0$, then the $\Pr(W \neq 0) = 0$. Probabilists say that $W = 0$ *almost surely* to distinnguish between events that don't exist (like $W^2 = -1$) and events that merely never happen.

[6]If $B \subseteq R^n$ and $C \subseteq R^m$, then $\Pr(X \in B \text{ and } Y \in C) = \Pr(X \in B)\Pr(Y \in C)$.

This shows that if $y = u(x)$, then $g(y) = f(x)/u'(x)$.

Three common continuous random variables are *uniform*, *exponential*, and *gaussian* (or *normal*). In each case there is a *standard* version and a general version that is easy to express in terms of the standard version. The standard uniform random variable, U, has probability density.

$$f(u) = \begin{cases} 1 & \text{if } 0 \leq u \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{9.5}$$

Because the density is constant, U is equally likely to be anywhere within the unit interval, $[0, 1]$. From this we can create the general random variable uniformly distributed in the interval $[a, b]$ by $Y = (b - a)U + a$. The PDF for Y is

$$g(y) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq y \leq b \\ 0 & \text{otherwise.} \end{cases}$$

The exponential random variable, T, with *rate constant* $\lambda > 0$, has PDF

$$f(t) = \begin{cases} \lambda e^{-\lambda t} & \text{if } 0 \leq t \\ 0 & \text{if } t < 0. \end{cases} \tag{9.6}$$

The exponential is a model for the amount of time something (e.g. a light bulb) will work before it breaks. It is characterized by the *Markov property*, if it has not broken by time t, it is as good as new. Let λ characterize the probability density for breaking right away, which means $\lambda dt = \Pr(0 \leq T \leq dt)$. The random time T is exponential if all the conditional probabilities are equal to this:

$$\Pr(t \leq T \leq t + dt \mid T \geq t) = \Pr(T \leq dt) = \lambda dt \,.$$

Using Bayes' rule (9.1), and the observation that $\Pr(T \leq t + dt \text{ and } T \geq t) = f(t)dt$, this becomes

$$\lambda dt = \frac{f(t)dt}{1 - F(t)} \,,$$

which implies $\lambda\,(1 - F(t)) = f(t)$, and, by differentiating, $-\lambda f(t) = f'(t)$. This gives $f(t) = Ce^{-\lambda t}$ for $t > 0$. We find $C = \lambda$ using $\int_0^\infty f(t)dt = 1$. Independent exponential *inter arrival* times generate the *Poisson arrival processes*. Let T_k, for $k = 1, 2, \ldots$ be independent exponentials with rate λ. The k^{th} *arrival time* is

$$S_k = \sum_{j \leq k} T_j \,.$$

The expected number of arrivals in interval $[t_1, t_2]$ is $\lambda(t_2 - t_1)$ and all arrivals are independent. This is a fairly good model for the arrivals of telephone calls at a large phone bank.

We denote the *standard normal* by Z. The standard normal has PDF

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2} \,. \tag{9.7}$$

The general normal with mean μ and variance σ^2 is given by $X = \sigma Z + \mu$ and has PDF

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} . \tag{9.8}$$

We write $X \sim \mathcal{N}(\mu, \sigma^2)$ in this case. A standard normal has distribution $\mathcal{N}(0,1)$. An n component random variable, X, is a *multivariate normal* with mean μ and covariance C it has probability density

$$f(x) = \frac{1}{Z} \exp\left((x-\mu)^* H (x-\mu)/2\right) , \tag{9.9}$$

where $H = C^{-1}$ and the *normalization constant* is (we don't use this) $Z = 1/\sqrt{(2\pi)^n \det(C)}$. The reader should check that this is the same as (9.8) when $n = 1$.

The class of multivariate normals has the *linear transformation property.* Suppose L is an $m \times n$ matrix with rank m (L is a linear transformation from R^n to R^m that is onto R^m). If X is an n dimensional multivariate normal, then Y is an m dimensional multivariate normal. The covariance matrix for Y is given by

$$C_Y = L C_X L^* . \tag{9.10}$$

We derive this, taking $\mu = 0$ without loss of generality (why?), as follows:

$$C_Y = E\left[YY^*\right] = E\left[(LX)\left((LX)^*\right] = E\left[LXX^*L^*\right] = LE\left[XX^*\right]L^* = LC_X L^* .$$

The two important theorems for simple Monte Carlo are the *law of large numbers* and the *central limit theorem.* Suppose $A = E[X]$ and X_k for $k = 1, 2, \ldots$ is an iid sequence of samples of X. The approximation of A is

$$\widehat{A}_n = \frac{1}{n} \sum_{k=1}^{n} X_k .$$

The error is $R_n = \widehat{A}_n - A$. The law of large numbers states[7] that $\widehat{A}_n \to A$ as $n \to \infty$. Statisticians call estimators with this property *consistent.* The central limit theorem states that if $\sigma^2 = \mathrm{var}(X)$, then $R_n \approx \mathcal{N}(0, \sigma^2/n)$. It is easy to see that

$$E\left[R_n\right] = E\left[\widehat{A}_n\right] - A = 0 ,$$

and that (recall that A is not random)

$$\mathrm{var}(R_n) = \mathrm{var}(\widehat{A}_n) = \frac{1}{n}\mathrm{var}(X) .$$

The first property makes the estimator *unbiased*, the second follows from independence of the X_k. The deep part of the central limit theorem is that for large n, R_n is approximately normal, regardless of the distribution of X (as long as $E\left[X^2\right] < \infty$).

[7] The *Kolmogorov strong law* of large numbers is the theorem that $\lim_{n\to\infty} \widehat{A}_n = A$ almost surely, i.e. that the probability of the limit not existing or being the wrong answer is zero. More useful for us is the *weak law*, which states that, for any $\epsilon > 0$, $P(|R_n| > \epsilon) \to 0$ as $n \to \infty$. These are closely related but not the same.

9.2 Random number generators

The random variables used in Monte Carlo are generated by a (pseudo) random number generator. The procedure `double rng()` is a perfect random number generator if

```
for( k=0; k<n; k++ )  U[k] = rng();
```

produces an array of iid standard uniform random variables. The best available random number generators are perfect in this sense for nearly all practical purposes. The native C/C++ procedure `random()` is good enough for most Monte Carlo (I use it).

Bad ones, such as the native `rand()` in C/C++ and the procedure in *Numerical Recipies* give incorrect results in common simple cases. If there is a random number generator of unknown origin being passed from person to person in the office, do not use it (without a condom).

The computer itself is not random. A pseudo random number generator simulates randomness without actually being random. The *seed* is a collection of m integer variables: `int seed[m];`. Assuming standard C/C++ 32 bit integers, the number of bits in the seed is $32 \cdot m$. There is a *seed update* function $\Phi(s)$ and an *output* function $\Psi(s)$. The update function produces a new seed: $s' = \Phi(s)$. The output function produces a floating point number (check the precision) $u = \Psi(s) \in [0,1]$. One call `u = rng();` has the effect

$$s \longleftarrow \Phi(s)\ ; \quad \texttt{return}\ u = \Psi(s)\ ;\ .$$

The random number generator should come with procedures `s = getSeed();` and `setSeed(s);`, with obvious functions. Most random number generators set the initial seed to the value of the system clock as a default if the program has no `setSeed(s);` command. We use `setSeed(s)` and `getSeed()` for two things. If the program starts with `setSeed(s);`, then the sequence of seeds and "random" numbers will be the same each run. This is helpful in debugging and reproducing results across platforms. The other use is *checkpointing*. Some Monte Carlo runs take so long that there is a real chance the computer will crash during the run. We avoid losing too much work by storing the state of the computation to disk every so often. If the machine crashes, we restart from the most recent checkpoint. The random number generator seed is part of the checkpoint data.

The simplest random number generators use linear congruences. The seed represents an integer in the range $0 \leq s < c$ and Φ is the linear congruence (a and b positive integers) $s' = \Phi(s) = (as + b)_{\mathrm{mod}\ c}$. If $c > 2^{32}$, then we need more than one 32 bit integer variable to store s. Both `rand()` and `random()` are of this type, but `rand()` has $m = 1$ and `random()` has $m = 4$. The output is $u = \Psi(s) = s/c$. The more sophisticated random number generators are of a similar computational complexity.

9.3 Sampling

A *simple sampler* is a procedure that produces an independent sample of X each time it is called. The job of a simple sampler is to turn iid standard uniforms into samples of some other random variable. A large Monte Carlo computation may spend most of its time in the sampler and it often is possible to improve the performance by paying attention to the details of the algorithm and coding. Monte Carlo practitioners often are amply rewarded for time stpnt tuning their samplers.

9.3.1 Bernoulli coin tossing

A *Bernoulli* random variable with parameter p, or a *coin toss*, is a random variable, X, with $\Pr(X = 1) = p$ and $\Pr(X = 0) = 1 - p$. If U is a standard uniform, then $p = \Pr(U \leq p)$. Therefore we can sample X using the code fragment

```
X=0; if ( rng() <= p) X=1;
```

Similarly, we can sample a random variable with finitely many values $\Pr(X = x_k) = p_k$ (with $\sum_k p_k = 1$) by dividing the unit interval into disjoint sub intervals of length p_k. This is all you need, for example, to simulate a simple random walk or a finite state space Markov chain.

9.3.2 Exponential

If U is standard uniform, then

$$T = \frac{-1}{\lambda} \ln(U) \tag{9.11}$$

is an exponential with rate parameter λ. Before checking this, note first that $U > 0$ so $\ln(U)$ is defined, and $U < 1$ so $\ln(U)$ is negative and $T > 0$. Next, since λ is a rate, it has units of 1/Time, so (9.11) produces a positive number with the correct units. The code `T = -(1/lambda)*log( rng() );` generates the exponential.

We verify (9.11) using the informal probability method. Let $f(t)$ be the PDF of the random variable of (9.11). We want to show $f(t) = \lambda e^{-\lambda t}$ for $t > 0$. Let B be the event $t \leq T \leq t + dt$. This is the same as

$$t \leq \frac{-1}{\lambda} \ln(U) \leq t + dt \ ,$$

which is the same as

$$-\lambda t - \lambda dt \leq \ln(U) \leq -\lambda t \qquad \text{(all negative)},$$

and, because e^x is an increasing function of x,

$$e^{-\lambda t - \lambda dt} \leq U \leq e^{-\lambda t} \ .$$

Now, $e^{-\lambda t - \lambda dt} = e^{-\lambda t} e^{-\lambda dt}$, and $e^{-\lambda dt} = 1 - \lambda dt$, so this is

$$e^{-\lambda t} - \lambda dt e^{-\lambda t} \leq U \leq e^{-\lambda t} \ .$$

But this is an interval within $[0,1]$ of length $\lambda\, dt\, e^{-\lambda t}$, so

$$\begin{aligned} f(t)dt &= \Pr(t \leq T \leq t + dt) \\ &= \Pr(e^{-\lambda t} - \lambda dt e^{-\lambda t} \leq U \leq e^{-\lambda t}) \\ &= \lambda\, dt\, e^{-\lambda t} \ , \end{aligned}$$

which shows that (9.11) gives an exponential.

9.3.3 Using the distribution function

In principle, the CDF provides a simple sampler for any one dimensional probability distribution. If X is a one component random variable with probability density $f(x)$, the cumulative distrubution function is $F(x) = \Pr(X \leq x) = \int_{x' \leq x} f(x' dx'$. Of course $0 \leq F(x) \leq 1$ for all x, and for any $u \in [0,1]$, there is an x with $F(x) = u$. Moreover, if $X \sim f$ and $U = F(X)$, and $u = F(x)$, $\Pr(U \leq u) = \Pr(X \leq x) = F(x) = u$. But $\Pr(U \leq u) = u$ says that U is a standard uniform. Conversely, we can reverse this reasoning to see that if U is a standard uniform and $F(X) = U$, then X has probability density $f(x)$. The simple sampler is, first take `U  rng();`, then find X with $F(X) = U$. The second step may be difficult in applications where $F(x)$ is hard to evaluate or the equation $F(x) = u$ is hard to solve.

For the exponential random variable,

$$F(t) = \Pr(T \leq t) = \int_{t'=0}^{t} \lambda e^{-\lambda t'} dt' = 1 - e^{-\lambda t} \ .$$

Solving $F(t) = u$ gives $t = \frac{-1}{\lambda} \ln(1-u)$. This is the same as the sampler we had before, since $1-u$ also is a standard uniform.

For the standard normal we have

$$F(z) = \int_{z'=-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-z'^2/2} dz' = N(z) \ . \tag{9.12}$$

There is no *elementary*[8] formula for the *cumulative normal*, $N(z)$, but there is good software to evaluate it to nearly double precision accuracy, both for $N(z)$ and for the inverse cumulative normal $z = N^{-1}(u)$. In many applications,[9] this is the best way to make standard normals. The general $X \sim \mathcal{N}(\mu, \sigma^2)$ may be found using $X = \sigma Z + \mu$.

[8]An elementary function is one that can be expressed using exponentials, logs, trigonometric, and algebraic functions only.

[9]There are applications where the relationship between Z and U is important, not only the value of Z. These include sampling using normal copulas, and quasi (low discrepency sequence) Monte Carlo.

9.3.4 The Box Muller method

The *Box Muller* algorithm generates two independent standard normals from two independent standard uniforms. The formulas are

$$\begin{aligned} R &= \sqrt{-2\ln(U_1)} \\ \Theta &= 2\pi U_2 \\ Z_1 &= R\cos(\Theta) \\ Z_2 &= R\sin(\Theta) \; . \end{aligned}$$

We can make a thousand independent standard normals by making a thousand standard uniforms then using them in pairs to generate five hunderd pairs of independent standard normals.

9.3.5 Multivariate normals

Let $X \in R^n$ be a multivariate normal random variable with mean zero and covariance matrix C. We can sample X using the Choleski factorization of C, which is $C\ LL^T$, where L is lower triangular. Note that L exists because C is symmetric and positive definite. Let $Z \in R^n$ be a vector of n independent standard normals generated using Box Muller or any other way. The covariance matrix of Z is I (check this). Therefore, if

$$X = LZ \; , \tag{9.13}$$

then X is multivariate normal (because it is a linear transformation of a multivariate normal) and has covariance matrix (see (9.10)

$$C_X = LIL^t = C \; .$$

If we want a multivariate normal with mean $\mu \in R^n$, we simply take $X = LZ+\mu$.

9.3.6 Rejection

The *rejection* algorithm turns samples from one density into samples of another. One ingredient is a simple sampler for the *trial distribution*, or *proposal distribution*. Suppose `gSamp()` produces iid samples from the PDF $g(x)$. The other ingredient is an *acceptance probability*, $p(x)$, with $0 \leq p(x) \leq 1$ for all x. The algorithm generates a trial $X \sim g$ and accepts this trial value with probability $p(X)$. The process is repeated until the first acceptance. All this happens in

```
while ( rng() > p( X = gSamp() ) );
```
(9.14)

We accept X is $U \leq p(X)$, so $U > p(X)$ means reject and try again. Each time we generate a new X, which must be independent of all previous ones.

The X returned by (9.14) has PDF

$$f(x) = \frac{1}{Z} p(x) g(x) \; , \tag{9.15}$$

where Z is a normalization constant that insures that $\int f(x)dx = 1$:

$$Z = \int_{x \in R^n} p(x)g(x)dx \ . \tag{9.16}$$

This shows that Z is the probability that any given trial will be an acceptance. The formula (9.15) shows that rejection goes from g to f by thinning out samples in regions where $f < g$ by rejecting some of them. We verify it informally in the one dimensional setting:

$$\begin{aligned}
f(x)dx &= \Pr(\text{ accepted } X \in (x, x+dx)) \\
&= \Pr(X \in (x, x+dx) \mid \text{ acceptance }) \\
&= \frac{\Pr(X \in (x, x+dx) \text{ and accepted})}{\Pr(\text{ accepted })} \\
&= \frac{g(x)dxp(x)}{Z} \ .
\end{aligned}$$

An argument like this also shows the correctness of (9.15) also for multivariate random variables.

We can use rejection to generate normals from exponentials. Suppose $g(x) = e^{-x}$ for $x > 0$, corresponding to a standard exponential, and $f(x) = \frac{2}{\sqrt{2\pi}} e^{-x^2/2}$ for $x > 0$, corresponding to the positive half of the standard normal distribution. Then (9.15) becomes

$$\begin{aligned}
p(x) &= Z \frac{f(x)}{g(x)} \\
&= Z \cdot \frac{2}{\sqrt{2\pi}} \cdot \frac{e^{-x^2/2}}{e^{-x}} \\
p(x) &= Z \cdot \frac{2}{\sqrt{2\pi}} e^{x - x^2/2} \ .
\end{aligned} \tag{9.17}$$

This would be a formula for $p(x)$ if we know the constant, Z.

We maximize the efficiency of the algorithm by making Z, the overall probability of acceptance, as large as possible, subject to the constraint $p(x) \le 1$ for all x. Therefore, we find the x that maximizes the right side:

$$e^{x - x^2/2} = \max \quad \Longrightarrow \quad x - \frac{x^2}{2} = \max \quad \Longrightarrow \quad x_{\max} = 1 \ .$$

Choosing Z so that the maximum of $p(x)$ is one gives

$$1 = p_{\max} = Z \cdot \frac{2}{\sqrt{2\pi}} e^{x_{\max} - x_{\max}^2/2} = Z \frac{2}{\sqrt{2\pi}} e^{1/2} \ ,$$

so

$$p(x) = \frac{1}{\sqrt{e}} e^{x - x^2/2} \ . \tag{9.18}$$

It is impossible to go the other way. If we try to generate a standard exponential from a positive standard normal we get accetpance probability related to the recripocal to (9.17):

$$p(x) = Z\frac{\sqrt{2\pi}}{2}e^{x^2/2-x} \ .$$

This gives $p(x) \to \infty$ as $x \to \infty$ for any $Z > 0$. The normal has thinner[10] *tails* than the exponential. It is possible to start with an exponential and thin the tails using rejection to get a Gaussian (Note: (9.17) has $p(x) \to 0$ as $x \to \infty$.). However, rejection cannot fatten the tails by more than a factor of $\frac{1}{Z}$. In particular, rejection cannot fatten a Gaussian tail to an exponential.

The *efficiency* of a rejection sampler is the expected number of trials needed to generate a sample. Let N be the number of samples to get a success. The efficiency is

$$E[N] = 1 \cdot \Pr(N = 1) + 2 \cdot \Pr(N = 2) + \cdots$$

We already saw that $\Pr(N = 1) = Z$. To have $N = 2$, we need first a rejection then an acceptance, so $\Pr(N = 2) = (1 - Z)Z$. Similarly, $\Pr(N = k) = (1 - Z)^{k-1}Z$. Finally, we have the geometric series formulas for $0 < r < 1$:

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r} \ , \quad \sum_{k=1}^{\infty} kr^{k-1} = \sum_{k=0}^{\infty} kr^{k-1} = \frac{d}{dr}\sum_{k=0}^{\infty} r^k = \frac{1}{(1-r)^2} \ .$$

Applying these to $r = 1 - Z$ gives $E[N] = \frac{1}{Z}$. In generating a standard normal from a standatd exponential, we get

$$Z = \sqrt{\frac{\pi}{2e}} \approx .76 \ .$$

The sampler is efficient in that more than 75% of the trials are successes.

Rejection samplers for other distributions, particularly in high dimensions, can be much worse. We give a rejection algorithm for finding a random point uniformly distributed inside the unit ball n dimensions. The algorithm is correct for any n in the sense that it produces at the end of the day a random point with the desired probability density. For small n, it even works in practice and is not such a bad idea. However, for large n the algorithm is very inefficient. In fact, Z is an exponentially decreasing function of n. It would take more than a century on any present computer to generate a point uniformly distributed inside the unit ball in $n = 100$ dimensions this way. Fortunately, there are better ways.

A point in n dimensions is $x = (x_1, \ldots, x_n)$. The *unit ball* is the set of points with $\sum_{k=1}^{n} x_k^2 \leq 1$. We will use a trial density that is uniform inside the smallest (hyper)cube that contains the unit ball. This is the cube with $-1 \leq x_k \leq 1$ for each k. The uniform density in this cube is

$$g(x_1, \ldots, x_n) = \begin{cases} 2^{-n} & \text{if } |x_k| \leq 1 \text{ for all } k = 1, \ldots, n \\ 0 & \text{otherwise.} \end{cases}$$

[10] The tails of a probability density are the parts for large x, where the graph of $f(x)$ gets thinner, like the tail of a mouse.

This density is a product of the one dimensional uniform densities, so we can sample it by choosing n independent standard uniforms:

```
for( k = 0; k < n; k++) x[k] = 2*rng() - 1; // unif in [-1,1].
```

We get a random point inside the unit ball if we simply reject samples outside the ball:

```
while(1) {     // The rejection loop (possibly infinite!)

   for( k = 0; k < n; k++) x[k] = 2*rng() - 1; // Generate a trial vector
                                                // of independent uniforms
                                                // in [-1,1].
   ssq = 0;  // ssq means "sum of squares"
   for( k = 0; k < n; k++ ) ssq+= x[k]*x[k];
   if ( ssq <= 1.) break ; // You accepted and are done with the loop.
                           // Otherwise go back and do it again.
  }
```

The probability of accepting in a given trial is equal to the ratio of the volume (or area in 2D) of the ball to the cube that contains it. In 2D this is

$$\frac{area(\text{disk})}{area(\text{square})} = \frac{\pi}{4} \approx .79 \ ,$$

which is pretty healthy. In 3D it is

$$\frac{vol(\text{ball})}{vol(\text{cube})} = \frac{\frac{4\pi}{3}}{8} \approx .52 \ ,$$

Table 9.1 shows what happens as the dimension increases. By the time the dimension reaches $n = 10$, the expected number of trials to get a success is about $1/.0025 = 400$, which is slow but not entirely impractical. For dimension $n = 40$, the expected number of trials has grown to about 3×10^{20}, which is entirely impractical. Monte Carlo simulations in more than 40 dimensions are common. The last column of the table shows that the acceptance probability goes to zero faster than any exponential of the form e^{-cn}, because the numbers that would be c, listed in the table, increase with n.

9.3.7 Histograms and testing

Any piece of scientific software is presumed wrong until it proves itself correct in tests. We can test a one dimensional sampler using a *histogram*. Divide the x axis into neighboring *bins* of length Δx centered about bin centers $x_j = j\Delta x$. The corresponding bins are $B_j = [x_j - \frac{\Delta x}{2}, x_j + \frac{\Delta x}{2}]$. With n samples, the *bin counts* are[11] $N_j = \#\{X_k \in B_j, 1 \leq k \leq n\}$. The probability that a given sample lands in B_j is $\Pr(B_j) = \int_{x \in B_j} f(x)dx \approx \Delta x f(x_j)$. The expected bin count is $E[N_j] \approx n\Delta x f(x_j)$, and the standard deviation (See Section 9.4) is

[11] Here $\#\{\cdots\}$ means the number of elements in the set $\{\cdots\}$.

Table 9.1: Acceptence fractions for producing a random point in the unit ball in n dimensions by rejection.

dimension	*vol*(ball)	*vol*(cube)	ratio	$-ln$(ratio)/dim
2	π	4	.79	.12
3	$4\pi/3$	8	.52	.22
4	$\pi^2/2$	16	.31	.29
10	$2\pi^{n/2}/(n\Gamma(n/2))$	2^n	.0025	.60
20	$2\pi^{n/2}/(n\Gamma(n/2))$	2^n	2.5×10^{-8}	.88
40	$2\pi^{n/2}/(n\Gamma(n/2))$	2^n	3.3×10^{-21}	1.2

$\sigma_{N_j} \approx \sqrt{n\Delta x}\sqrt{f(x_j)}$. We generate the samples and plot the N_j and $E[N_j]$ on the same plot. If $E[N_j] \gg \sigma_{N_j}$, then the two curves should be relatively close. This condition is

$$\frac{1}{\sqrt{n\Delta x}} \ll \sqrt{f(x_j)}\ .$$

In particular, if f is of order one, $\Delta x = .01$, and $n = 10^6$, we should have reasonable agreement if the sampler is correct. If Δx is too large, the approximation $\int_{x\in B_j} f(x)dx \approx \Delta x f(x_j)$ will not hold. If Δx is too small, the histogram will not be accurate.

It is harder to test higher dimensional random variables. We can test two and possibly three dimensional random variables using multidimensional histograms. We can test that various one dimensional functions of the random X have the right distributions. For example, the distributions of $R^2 = \sum X_k^2 = \|X\|_{l^2}^2$ and $Y = \sum a_k X_k = a \cdot X$ are easy to figure out if X is uniformly distributed in the ball.

9.4 Error bars

It is relatively easy to estimate the order of magnitude of the error in most Monte Carlo computations. Monte Carlo computations are likely to have large errors (see Chapter 2). Therefore, all Monte Carlo computations (except possibly those for senior management) should report error estimates.

Suppose X is a scalar random variable and we approximate $A = E[X]$ by

$$\widehat{A}_n = \frac{1}{n}\sum_{k=1}^{n} X_k\ .$$

The central limit theorem states that

$$R_n = \widehat{A}_n - A \approx \sigma_n Z\ , \tag{9.19}$$

where σ_n is the *standard deviation* of $\widehat{A}_n$ and $Z \sim \mathcal{N}(0,1)$. A simple calculation shows that $\sigma_n = \frac{1}{\sqrt{n}}\sqrt{\sigma^2}$, where $\sigma^2 = \mathrm{var}(X) = E[(X-A)^2]$. We estimate σ^2

using[12]

$$\widehat{\sigma_n^2} = \frac{1}{n}\sum_{k=1}^{n}\left(X_k - \widehat{A}_n\right)^2 \ , \tag{9.20}$$

then take

$$\widehat{\sigma}_n = \frac{1}{\sqrt{n}}\sqrt{\widehat{\sigma_n^2}} \ .$$

Since Z in (9.19) will be of order one, R_n will be of order $\widehat{\sigma}_n$.

We typically report Monte carlo data in the form $A = \widehat{A}_n \pm \widehat{\sigma}_n$. Graphically, we plot $\widehat{A}_n$ as a circle (or some other symbol) and $\widehat{\sigma}_n$ using a bar of length $2\widehat{\sigma}_n$ with $\widehat{A}_n$ in the center. This is the *error bar*.

We can think of the error bar as the interval $\left[\widehat{A}_n - \widehat{\sigma}_n, \widehat{A}_n + \widehat{\sigma}_n\right]$. More generally, we can consider k standard deviation error bars $\left[\widehat{A}_n - k\widehat{\sigma}_n, \widehat{A}_n + k\widehat{\sigma}_n\right]$. In statistics, these intervals are called *confidence intervals* and the confidence is the probability that A is within the conficence interval. The central limit theorem (and numerical computations of integrals of gaussians) tells us that one standard deviation error bars have have confidence

$$\Pr\left(A \in \left[\widehat{A}_n - \widehat{\sigma}_n, \widehat{A}_n + \widehat{\sigma}_n\right]\right) \approx 66\% \ ,$$

and and two standard deviation error bars have

$$\Pr\left(A \in \left[\widehat{A}_n - 2\widehat{\sigma}_n, \widehat{A}_n + 2\widehat{\sigma}_n\right]\right) \approx 95\% \ ,$$

It is the custom in Monte Carlo practice to plot and report one standard deviation error bars. This requires the consumer to understand that the exact answer is outside the error bar about a third of the time. Plotting two or three standard deviation error bars would be safer but would give an inaccurate picture of the probable error size.

9.5 Software: performance issues

Monte Carlo methods raises many performance issues. Naive coding following the text can lead to poor performance. Two significant factors are frequent branching and frequent procedure calls.

9.6 Resources and further reading

There are many good books on the probability background for Monte Carlo, the book by Sheldon Ross at the basic level, and the book by Sam Karlin and Gregory Taylor for more the ambitious. Good books on Monte Carlo include

[12] Sometimes (9.20) is given with $n-1$ rather than n in the denominator. This can be a serious issue in practical statistics with small datasets. But Monte Carlo datasets should be large enough that the difference between n and $n-1$ is irrelevent.

the still surprisingly useful book by Hammersley and Handscomb, the physicists' book (which gets the math right) by Mal Kalos and Paula Whitlock, and the broader book by George Fishman. I also am preparing a book on Monte Carlo, whth many parts already posted.

Markov Chain Monte Carlo, or *MCMC*, is the most important topic left out here. Most multivariate random variables not discussed here cannot be sampled in a practical way by the *direct* sampling methods, but do have indirect *dynamic* samplers. The ability to sample essentially arbitrary distributions has led to an explosion of applications in statistics (sampling Bayesian posterior distributions, Monte Carlo calibration of statistical tests, etc.), and operations research (improved rare event sampling, etc.).

Choosing a good random number generator is important yet subtle. The native C/C++ function `rand()` is suitable only for the simplest applications because it cycles after only a billion samples. The function `random()` is much better. The random number generators in Matlab are good, which cannot be said for the generators in other scientific computing and visualization packages. Joseph Marsaglia has a web site with the latest and best random number generators.

9.7 Exercises

1. What is wrong with the following piece of code?

```
for ( k = 0; k < n; k++ ) {
   setSeed(s);
   U[k] = rng();
  }
```

2. Calculate the distribution function for an exponential random variable with rate constant λ. Show that the sampler using the distribution function given in Section 9.3.3 is equivalent to the one given in Section 9.3.2. Note that if U is a standard uniform, then $1-U$ also is standard uniform.

3. If S_1 and S_2 are independent standard exponentials, then $T = S_1 + S_2$ has PDF $f(t) = te^{-t}$.

   (a) Write a simple sampler of T that generates S_1 and S_2 then takes $T = S_1 + S_2$.

   (b) Write a simpler sampler of T that uses rejection from an exponential trial. The trial density must have $\lambda < 1$. Why? Look for a value of λ that gives reasonable efficiency. Can you find the optimal λ?

   (c) For each sampler, use the histogram method to verify its correctness.

   (d) Program the Box Muller algorithm and verify the results using the histogram method.

4. A *Poisson* random walk has a position, $X(t)$ that starts with $X(0) = 0$. At each time T_k of a Poisson process with rate λ, the position moves (jumps) by a $\mathcal{N}(0, \sigma^2)$, which means that $X(T_k + 0) = X(T_k - 0) + \sigma Z_k$ with iid standard normal Z_k. Write a program to simulate the Poisson random walk and determine $A = \Pr(X(T) > B)$. Use (but do not hard wire) two parameter sets:

   (a) $T = 1$, $\lambda = 4$, $\sigma = .5$, and $B = 1$.

   (b) $T = 1$, $\lambda = 20$, $\sigma = .2$, and $B = 2$.

   Use standard error bars. In each case, choose a sample size, n, so that you calculate the answer to 5% relative accuracy and report the sample size needed for this.

Index

Index